Software Engineering Essentials

Volume I: The Development Process

FOURTH EDITION

Become a Certified Software Engineer

The *CSDA credential* is intended for graduating software engineers and entry-level software professionals.

The *CSDP credential* is intended for midcareer software professionals looking to advance in their careers

---

The IEEE Computer Society (The world's largest organization of computer professionals) has launched an exam-based process for certifying software engineers as software engineering professionals.

This certificate establishes that the certificate holder is capable of using software engineering methods, tools, and techniques to develop and build software systems and, in addition, can fulfill the roles of:

- **Software project manager**
- **Software architect and requirements analyst**
- **Software designer**
- **Software configuration manager**
- **Software quality-assurance engineer**
- **V&V engineer**
- **Software test lead**
- **And so forth**

The author team of Drs. Richard Hall Thayer and Merlin Dorfman has written a three-volume set of study guides to assist the potential certificate holder to pass the CSDP exam (you are currently reading Volume I).

In addition, Dr. Thayer has developed a self-teaching, multimedia, CD training course, with both audio and visual component as an addition study guide in passing the certificate exam.

For more information go to www.CSDP-Training.com

Software Engineering Essentials

Volume I: The Development Process

FOURTH EDITION

A multi- text software engineering course or courses (based on the 2013 IEEE SWEBOK) for undergraduate and graduate university students

A self-teaching IEEE CSDP/CADA certificate exam training course based on the Computer Society's CSDP exam specifications

Edited & Written by:

Richard Hall Thayer and Merlin Dorfman

Published by:
Software Management Training Press
Carmichael, California

2013

Carmichael, California

Copyright Notices

- SWEBOK *Copyright and Reprint Permissions:* This document may be copied, in whole or in part, in any form or by any means, as is, or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy. Any other use or distribution of this document is prohibited without the prior express permission of IEEE.
- *Understanding Software Requirements*, by Stuart R. Faulk (Based on "Software Requirements: A Tutorial," by Stuart R. Faulk, which appeared in R.H. Thayer and M. Dorfman (editors), *Software Requirements Engineering*, 2nd edition, IEEE Computer Society Press, Los Alamitos, CA, © IEEE 1997).
- *Software Design: Introduction and Overview* by David Budgen (Based on "Software Design: An Introduction," by David Budgen, which appeared in R.H. Thayer and M. Dorfman (editors), *Software Engineering,* Volume I, 3rd edition, IEEE Computer Society Press, Los Alamitos, CA, © IEEE 2005).
- *Welcome to Software Construction* by Steve McConnell (This paper is an extract from McConnell's book, *Code Complete: A practical handbook of software construction,* 2nd ed., Microsoft Press, Redmond, WA, 2004. Used with permission of Microsoft, Inc. Used with permission of the author). (This book is recommended by the IEEE Computer Society as a reference book for the CSDP exam).
- *Software Engineering Maintenance: An Introduction* by Keith H. Bennett (Based on "Software Maintenance: A Tutorial," by Keith H. Bennett, which appeared in R.H. Thayer and M. Dorfman (editors), *Software Engineering*, Volume I, 3rd edition, IEEE Computer Society Press, Los Alamitos, CA, ©2007 IEEE).
- *Wikipedia* is a free web based encyclopedia enabling multiple users to freely add and edit online content. Definitions cited on Wikipedia and their related sources have been verified by the authors and other peer reviewers. Readers who would like to verify a source or a reference should search the subject on Google, and read the technical report found under Wikipedia.
- *Portrait of Maxwell* by Laura Marshall Photography, San Jose, CA 2007. Permission to copy granted 2009.

ISBN-13: 978-0-9852707-0-4

ISBN-10: 0-9852707-0-5

Acknowledgments

No successful endeavor has ever been done by one person alone. We would like to thank the people and organizations that supported us in this effort:

First we would like to thank our wives – Mildred Thayer and Sandy Dorfman – for their high degree of tolerance in putting up with us working many hours on this manuscript.

We would also like to thank the writers, proofreaders, and subject matter experts who wrote our overview papers on the various software engineering processes:

Friedrich L. Bauer, Keith H. Bennett, Antonia Bertolino, J. Glenn Brookshear, David Budgen, Richard E. Fairley, Stuart R. Faulk, Alfonso Fuggetta, Hassan Gomaa, Don Gotterbarn, Anne Matte Jensen Hass, Jill Macari, Eda Marchetti, Steven McConnell, National Aeronautics and Space Administration (NASA), Jed Scully, Laura Sfardini, Ian Sommerville, Steve Tockey, Leonard Tripp, and Gerard Voland.

In addition, we would like to thank: Steve Tockey for providing us with the current CSDP exam specifications in order to maximize the usefulness of our study guide, Ellen Sander for helping with proofing and copy editing our manuscript to make it ready for printing, Melville (Mel) Piercey of Copy Plus for providing printing and other graphic support, and Jim Tozza for giving us technical support to keep our computer equipment running. Dr. Thayer would like to give an extra special thanks to Big John Nelson and Iain Low of the *Technical Progress Limited* (TPL) of Cumbernauld, Scotland, for providing computer equipment and support when Dr. Thayer was a resident at the University of Stirling, during the 2000s.

And finally, Dr. Thayer would like to thank his little dog Maxwell (a.k.a. Max, Maxito, or Speedy) who kept him company in the evening hours when everybody else had gone to bed.

A happy Max says that:

This is a Terrrrrrific Book

And be sure and also get Volumes II and III

A Detailed Guide to the IEEE SWEBOK and the IEEE CSDP/CSDA Exam

A Three Volume Set

This is volume I of a three-volume document (1) to provide a more detailed understanding of the SWEBOK knowledge areas (KAs) and (2) to use in preparing for the exams for the IEEE Software Engineering Certificates (called the Certified Software Development Associate and the Certified Software Development Professional, more commonly referred to as the CSDA and the CSDP). The *CSDA credential* is intended for graduating software engineers and entry-level software professionals. The *CSDP credential* is intended for mid-career software professionals looking to advance in their careers.

This study was partitioned into three volumes: (1) because of its overall size (over 700 pages), (2) to provide a university-level textbook for an IEEE SWEBOK-based software engineering course or courses, and/or (3) to allow exam takers to buy and study only what they need in order to take and pass the CSDA/CSDP exam.

The Table of Contents for Volume I follows. This document also includes the Table of Contents for Volume II and Volume III. This information is provided so that interested software engineeering exam takers can find additonal information about the test and dermine if the additional volumes are necessary. To aid in making this decision, the information contained in Table 1 (which follows) is provided.

Table 1: The differences between the CSDP and the CSDA exams

Contains:	Volume I	Volume II	Volume III
Chapters in the study guides	1 through 5	6 through 11	12 through 16
Page counts in the study guides	260	256	204
Percent coverage of exam for CSDP	47%	35%	18%
Percent coverage of exam for CSDA	39%	24%	37%

The new CSDP/CSDA exam specifications are much more detailed than the earlier exams and contain 15 KAs that need to be defined and explained (plus a 16th KA added by the authors).

Since the exam is based on the IEEE SWEBOK 2013, this guide books can also serve as a textbooks for university-level software engineering courses.

The CSDA and the CSDP exams are similar and follow the same exam specification. The biggest difference is that the CSDA exam places a greater emphasis (and more questions) on the KAs of *computer economics, science, engineering, and mathematics* and therefore contains less emphasis on the other KAs. However, this new reference guide can be used to study for both exams.

A Note to Our Readers

One of the advantages of using a "print-on-demand" or an e-mail publishing services is the ability to make changes to the manuscript relatively easily when errors or improvements are identified.

The authors encourage you to identify and send potential errors or suggested improvement to either Thayer or Dorfman at the below e-mail addresses. We don't guarantee to make all the changes identified, but we do promise to seriously consider all recommendations.

Note however that some things can't be changed. For example, the outline or contents of the exam specification, which are listed on the first page of all of the chapters labeled "n.2," are controlled by the Computer Society's *CSDP Certification Committee* and cannot be changed by us. Also we would have no real control over the papers written by the contributing software engineering experts (however we would notify them of your concerns).

These three software-engineering books were republished as "textbooks/guide book" in August 2013. One of the differences between a "guide book" and a "textbook" is that a textbook contains an "index" and a guide book usually does not. The index for this book is current under development and will be aided the textbook in the next printing. If you would like to have a copy of this index to use with this volume, please email Dr. Thayer, providing the volume number, and he will send you an electronic copy as soon as it is finished.

Thank you for listening.

Richard Hall Thayer
thayer@csus.edu

Merlin Dorfman
dorfman@computer.org

Disclaimer

While the authors have more than 80 years of system and software engineering experience between us, we are not experts in all aspects of this very large discipline. We have made extensive use of material written by subject matter experts and have cross-checked the material with other sources to confirm its accuracy.

Every effort has been made to make these *reference books* as complete and accurate as possible. However, mistakes may remain, both typographical and in content. Furthermore, while the books are current as of the date of publication of the SWEBOK and the CSDP and CSDA exam specifications, the state of the art advances on a daily basis. The reader should use his/her education and experience to supplement these *textbooks/guide books*.

Table of Contents

Volume I: The Development Process

Volume II: The Supporting Processes

Volume III: The Foundations Documents

Foreword

Software is pervasive in modern society. Problems with software quality are no longer just an inconvenience and an expense—they can impact the health and welfare of individuals and of society as a whole. Thus, it is vitally important that those of us involved in software development do all we can to ensure that the software we produce meets the users' needs—it does what it is intended to do, operates correctly, and doesn't do things that it shouldn't. Additionally, the survival of developer organizations requires that the software be produced quickly and economically.

More than 50 years ago, the IEEE Computer Society established itself as the leading association for computing professionals worldwide. Today there are nearly 85,000 members in over 140 countries. The Computer Society strives to be the leading provider of technical information and services to the world's computing professionals.

The Computer Society has always been instrumental in advancing the profession of software engineering. The Society's focus on advancing this important profession can be seen from the introduction of the *IEEE Transactions on Software Engineering* in 1975, to the introduction of the *SWEBOK Guide* in 2004, and has led to the development of formal software certification programs.

After more than three years of extensive research in the field among professionals, employers, and their customers, I initiated and launched the first IEEE Computer Society certification program in 2002—the Certified Software Development Professional (CSDP), intended for mid-level software engineering professionals.

After my term of office had passed, the Computer Society launched a second certificate program to satisfy a request for the computing industry to provide a means of evaluating entry level computer engineers prior to their being hired. This new certificate is entitled the Certified Software Development Associate (CSDA).

In 2008, both certifications were recognized as the first certifications to conform to the ISO-IEC 24773 Standard (*Software engineering—Certification of software engineering professionals—Comparison framework*), making them internationally portable. This development truly positions the Society as an international credentialing body with certification programs that are the benchmark standards in the field of software engineering.

The CSDP examination was designed to measure an individual's mastery of the fundamental knowledge required to perform the functions of an experienced software engineer. The CSDP also supports the Computer Society's position that the disciplined development to high-quality software requires a good development process and applicable software engineering standards.

Why should an individual software engineer be interested in becoming certified? The Computer Society lists these reasons [http://www.computer.org/portal/web/certification/home]:

In a world where software is pervasive, the need for skilled, competent, software development professionals is greater than ever.

- Graduates: Bridge the gap between your education and work requirements and verify your understanding of fundamental software development practices
- Professionals: Confirm your proficiency in established software development practices and demonstrate your commitment and professionalism

- Employers: Standardize your software development practices and protect your investment in a competent and proficient workforce.

The CSDP and CSDA exams cover a wide range and extensive depth of material as indicated in the Preface below. Many practitioners will be familiar with much of the required knowledge; few will have all, or even enough to pass the exam, "in their heads" or readily at hand. Reviewing college textbooks will refresh the candidate on the fundamentals, but much of the "state of the practice" is documented in journal articles, conference proceedings, web pages, and a plethora of books that an individual would be hard pressed to afford, much less read, and understand. How, then, to prepare efficiently and effectively to take a certification exam? The guide books by Thayer and Dorfman meet this need, covering all aspects of the exam topics in an affordable and readily-understood form. And even if you are not ready to take the leap and go for certification quite yet, the material in the guide books will round out your knowledge of the discipline and help you improve your professional performance.

Leonard L. Tripp
1999 President of the IEEE Computer Society

Honorary Foreword

To explain the origin of the term "Software Engineering," I submit the following story[1]

In the mid-1960s, there was increasing concern in scientific quarters of the Western world that the tempestuous development of computer hardware was not matched by appropriate progress in software. The software situation looked more turbulent. Operating systems had just been the latest rage, but they showed unexpected weaknesses. The uneasiness had been lined out in the NATO Science Committee by its U.S. representative, Dr. I.I. Rabi, the Nobel laureate and famous, as well as influential, physicist. In 1967, the Science Committee set up the Study Group on Computer Science, with members from several countries, to analyze the situation. The German authorities nominated me for this team. The study group was given the task of "assessing the entire field of computer science," with particular elaboration on the Science Committee's consideration of "organizing a conference and, perhaps, at a later date, setting up ... an International Institute of Computer Science."

The study group, concentrating its deliberations on actions that would merit an international rather than a national effort, discussed all sorts of promising scientific projects. However, it was rather inconclusive on the relation of these themes to the critical observations mentioned above, which had guided the Science Committee. Perhaps not all members of the study group had been properly informed about the rationale of its existence. In a sudden mood of anger, I made the remark, "The whole trouble comes from the fact that there is so much tinkering with software. It is not made in a clean fabrication process," and when I found out that this remark was shocking to some of my scientific colleagues, I elaborated the idea with the provocative saying, "What we need is *software engineering.*"

This remark had the effect that the expression "software engineering," which seemed to some to be a contradiction in terms, stuck in the minds of the members of the group. In the end, the study group recommended in late 1967 the holding of a Working Conference on Software Engineering, and I was made chairman. I had not only the task of organizing the meeting (which was held from October 7 to October 10, 1968, in Garmisch, Germany), but I had to set up a scientific program for a subject that was suddenly defined by my provocative remark. I enjoyed the help of my co-chairmen, L. Bolliet from France, and H. J. Helms from Denmark, and in particular the invaluable support of the program committee members, A. J. Perlis and B. Randell in the section on design, P. Naur and J. N. Buxton in the section on production, and K. Samuelson, B. Galler, and D. Gries in the section on service.

Among the 50 or so participants, E. W. Dijkstra was dominant. He actually made not only cynical remarks like "the dissemination of error-loaded software is frightening" and "it is not clear that the people who manufacture software are to be blamed. I think manufacturers deserve better, more understanding users." He also said already at this early date, "Whether the correctness of a piece of software can be guaranteed or not depends greatly on the structure of the thing made," and he had very fittingly named his paper "Complexity Controlled by Hierarchical Ordering of Function and Variability," introducing a theme that followed his life over the next 20 years.

---

1. Dr. Bauer originally wrote this paper as an introduction to a 1993 IEEE tutorial, R.H. Thayer, and A.D. McGettrick (eds.), *Software Engineering: A European Perspective*, IEEE Computer Society Press, Los Alamitos, CA, 1993.

Some of his words have become proverbs in computing, like "testing is a very inefficient way of convincing oneself of the correctness of a program."

With the wide distribution of the reports on the Garmisch Conference and on a follow-up conference in Rome, from October 27 to 31, 1969, it emerged that not only the phrase *software engineering,* but also the idea behind this became fashionable. Chairs were created, institutes were established (although the one which the NATO Science Committee had proposed did not come about because of reluctance on the part of Great Britain to have it organized on the European continent), and a great number of conferences were held.

The tutorial nature of the papers in this book is intended to offer readers an easy introduction to the topics and indeed to the attempts that have been made in recent years to provide them with the *tools,* both in a handcraft and intellectual sense, which allow them now to honestly call themselves software *engineers.*

Friedrich L. Bauer
Professor Emeritus
Technical University of Munich, Germany

Preface

These software engineering books serves two separate but connected audiences and roles:

1. **Software engineers who wish to study for and pass either or both of the IEEE Computer Society's software engineering certification exams.**

   The Certified Software Development Professional (CSDP) and is awarded to software engineers who have 5 to 7 years of software development experience and pass the CSDP exam. This certification was instituted in 2001 and establishes that the certificate holder is a competent software engineer in most areas of software engineering such as:

   - Software project manager
   - Software developer
   - Software configuration manager
   - Software quality-assurance expert
   - Software test lead
   - And so forth

   The other certificate is for recent software engineering graduates or self-taught software engineers and is designated Certified Software Development Associate (CDSA). The CSDA also requires passing an exam, but does not require any professional experience.

2. **University students who are taking (or reading) a BS or MS degree in software engineering, or practicing software engineers who want to update their knowledge.**

   This book was originally written as a guide to help software engineers take and pass the IEEE CSDP exam. However several reviewers commented that this book would also make a good university text book for an undergraduate or graduate course in software engineering. So the original books were modified to be applicable to both tasks.

The SWEBOK (Software Engineering Body of Knowledge) is a major milestone in the development and publicity of software engineering technology. However it needs to be noted that SWEBOK was NOT developed as a software engineering tutorial or textbook. *The SWEBOK is intended to catalog software engineering concepts, not teach them.*

The new, three-volume, fourth edition, *Software Engineering Essentials*, by Drs. Richard Hall Thayer and Merlin Dorfman attempts to fill this void. This new software engineering text expands on and replaces the earlier two-volume, third-edition, *Software Engineering* books which was also written by Thayer and Dorfman and published by the IEEE Computer Society Press [2006].

These new Volumes I and II offer a complete and detailed overview of software engineering as defined in IEEE SWEBOK 2013. These books provide a thorough analysis of *software development* in requirements analysis, design, coding, testing, and maintenance, plus the *supporting processes* of configuration management, quality assurance, verification and validation, and reviews and audits.

To keep up with evolution of the software industry (as expressed through evolution of the SWEBOK Guide, CSDP/CSDA, and the curriculum guidelines) a third volume in the Software Engineering series is needed. This third volume contains:

- *Software Engineering Measurements*
- *Software Engineering Economics*
- *Computer Foundations*
- *Mathematics Foundations*
- *Engineering Foundations*

This three-volume, *Software Engineering* Essentials series, provides an overview snapshot of the software state of the practice in a form that is a lot easier to digest than the SWEBOK Guide. The three-volume set is also a valuable reference (useful well beyond undergraduate and graduate software engineering university programs) that provides a concise survey of the depth and breadth of software engineering.

These new KAs exist so that software engineers can demonstrate a mastery of scientific technology and engineering. This is in answer to the criticism of software engineering that it does not contain enough engineering to qualify it as an engineering discipline.

1. History

In 2000, the president of the Computer Society, Mr. Leonard L. Tripp, asked Dr. Richard Hall Thayer to develop a reference/text and a three-day CSDP Software Engineering course to aid software engineers in refreshing their knowledge of software engineering. Dr. Thayer is a Fellow of the IEEE, a member of the Computer Society's Golden Core, and a Certified Software Development Professional. Thayer teamed with Dr. Merlin Dorfman (Fellow of the AIAA and registered Professional Engineer) to develop these reference books. The first result was a book titled *Software Engineering,* 2nd edition, in two volumes. (Thayer and Dorfman also wrote the first edition in 1997; however it preceded the CSDP program.) The third edition was written in 2005 to update and improve the contents. In 2009, the exam was updated and made broader (containing more knowledge areas) and more difficult. Therefore, the CSDP exam reference needed to be rewritten yet again.

In 2004, the IEEE Computer Society initiated a reference book on software engineering to provide an overview of the discipline of software engineering. This book is entitled Software Engineering Body of Knowledge (SWEBOK). SWEBOK parallels the CSDP exam specifications. The SWEBOK is being updated for 2013 and is now the driving force behind the CSDP exams. The primary purpose of the current revision of the SWEBOK Guide is to add a Knowledge Area (KA) on professional practices—a subject currently covered by the CSDP exams—and to add "foundation" KAs on high-tech subject is technology and science.

To achieve alignment with the CSDP and to maintain the currency of the SWEBOK Guide, the IEEE Computer Society's Professional Practices Committee agreed in 2008 to the following changes in the CSDP exam:

- Add four new education KAs: *Engineering Economy Foundations, Computing Foundations, Mathematical Foundations,* and *Engineering Foundations*
- Remove three *Related Disciplines of Software Engineering (Chapter 12, SWEBOK 2004): Computer Science, Mathematics, and Software Ergonomics*
- Add material about *Human-Computer Interfaces* in the *Software Design and Software Testing KA*

- *Remove the Software Tools* section from SWEBOK, *Software Engineering Tools and Methods,* and distribute the material to the other KAs
- Rename the *Software Engineering Tools and Methods* KA to *Software Engineering Methods* KA in SWEBOK 2013 to focus on methods that affect more than one KA

  For additional information see http://www.computer.org/portal/web/swebok

In 2010, the Computer Society launched an additional initiative to set up a software development certificate for recent university graduates and other entry-level software engineers or computer scientists. This certificate was called the Certified Software Development Associate (CSDA). The CSDA credential is intended for graduating software engineers and entry-level software professionals and serves to bridge the gap between educational experience and real-world work requirements.

The CSDP and CSDA exams are similar and are based on the same exam specification. However, the CSDA exam places more emphasis on the basic knowledge areas of computer science and engineering.

2. The Book's Contents

In its role as a supporting text to the IEEE SWEBOK, this reference book greatly expands the SWEBOK outline to provide greater detail to the SWEBOK engineering concepts and as a result should make an above average university software engineering textbook or textbooks. As an example, this text greatly expands the coverage of the software engineering project management KA to provide the detail necessary to (1) properly manage a large-scale software project or (2) to study for a software engineering project management course.

The new CSDP/CSDA exam specifications (which are based on the SWBOK 2013) are much more detailed than the earlier CSDP 2004 exams specifications. The new specifications contain 15 KAs. The CSDA exam is similar to the CSDP exam and uses the same exam specifications. The biggest difference is that the CSDA exam places a greater emphasis (and more questions) on the KAs of *computer economics, science, engineering,* and *mathematics* (see Table 1 earlier) and therefore, less emphasis (and questions) on the other KAs.

3. What Makes Our Book Unique?

This text makes use of the broad coverage of SWEBOK to ensure that all possible elements of the software engineering discipline are covered. We also asked *notable* software engineering authors to provide overview papers to provide a general look at some of the software engineering knowledge areas to help the student tie things together. By using the new print on demand (POD) business model to print our books and our decision to divide the extensive material into three parts we have provided one of the less expensive texts of it size and scope.

This is the fourth edition of this software engineering reference book and, in many ways, a better book than the earlier editions for upgrading a professional's software engineering knowledge.

Each chapter of the reference is divided into two parts. Part 1 consists of one or more papers written as an "overview tutorial" on one of the 16 KAs of the SWEBOK and the exam specifications. These authors are experts in their particular area and in many cases are also the authors of reference books recommended by the IEEE Computer Society to potential certification exam takers. Part 2 is an analysis of the certification exam specifications for that KA (written by the

Drs. Thayer and Dorfman). Part 2 was based on the exam specifications that were furnished to the *Guides* authors by the Computer Society committee who wrote the exam questions. Note that the questions themselves have not been and will not be released to *guide book authors.*

The exam specification outlines 15 software engineering knowledge areas (KAs). Our book covers 16 KAs because we split one area into two—the Software Engineering Management KA was separated into Software Engineering Project Management KA and Software Measurement and Metrics Foundation KA. We have recommended to the Computer Society that they do the same for the next SWEBOK.

Richard Hall Thayer, PhD, CSDP
Emeritus Professor of Software Engineering
California State University, Sacramento

Merlin Dorfman, PhD, PE
Quality Systems Staff Engineer (Retired)
Cisco Systems, Inc.

Chapter 1.1

Understanding Software Requirements[2]

Stuart R. Faulk
Department of Computer Science
University of Oregon
Eugene, Oregon

"The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements . . . No other part of the work so cripples the resulting system if done wrong. No other part is as difficult to rectify later" [Brooks 87].

1. Introduction

Deciding precisely what to build and then documenting the result is the goal of the requirements phase of software development. For many developers of large, complex software systems, requirements are their biggest software engineering problem. While there is considerable disagreement regarding how to solve the problem, few would disagree with Brook's assessment that no other part of a development is as difficult to do well or as disastrous in result when done poorly.

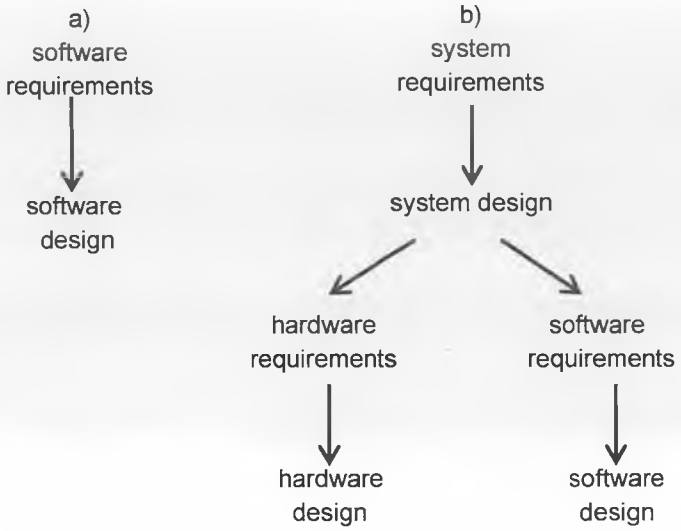


Figure 1: System vs. software requirements

The purpose of this tutorial is to help the reader understand why the apparently simple notion of "deciding what to build" is so difficult in practice, where the state of the art does and does not address these difficulties, and what hopes we have for doing better in the future. This paper does *not* survey the literature, but rather seeks to provide the reader with an understanding of the

2. Based on "Software Requirements: A Tutorial," by Stuart R. Faulk, which appeared in R.H. Thayer and M. Dorfman (editors), *Software Requirements Engineering*, 2nd edition, IEEE Computer Society Press, Los Alamitos, CA, 1997.

underlying issues. There are currently many more approaches to requirements than one can cover in a short paper. This diversity is the product of different views about which of the many problems in requirements are pivotal and of different assumptions concerning the desirable characteristics of a solution. We begin with basic terminology and some historical data about the requirements problem. We examine the goals of the requirements phase and the problems that can arise while attempting to meet those goals.

As in Brooks' article, much of the discussion is motivated by the distinction between the difficulties inherent in what one is trying to accomplish (the "essential" difficulties) and those one creates through inadequate practice ("accidental" difficulties) [Brooks 87]. We discuss how a disciplined software engineering process helps address many of the accidental difficulties and why the focus of such a disciplined process is on producing a written specification of the detailed technical requirements. We examine current technical approaches to requirements in terms of the specific problems each approach seeks to address. Finally, we examine technical trends and discuss where significant advances are likely to occur in the future.

2. Requirements and the Software Life Cycle

A variety of software life–cycle models have been proposed with an equal variety of terminology. While differing in the detailed decomposition of the steps (e.g., prototyping models) or in the surrounding management and control structure (e.g., to manage risk), there is general agreement on the core elements of the model. Figure 2 presents a version of the model that illustrates the relationship between the software development stages and the related testing and acceptance phases.

When software is created in the context of a larger hardware and software system, system requirements are defined first followed by system design. System design includes decisions about which parts of the system requirements will be allocated to hardware and which to software. For software-only systems, the life-cycle model begins with analysis of the software requirements. From this point on, the role of software requirements in the development model is same whether or not the software is part of a larger system, as shown in Figure 2. For this reason, the remainder of our discussion does not distinguish whether or not software is developed as part of a larger system. For an overview of system versus software issues, the reader is referred to Dorfman and Thayer's survey [Thayer 90].

In a large system development, the software requirements specification may play a variety of roles:

- For customers, the requirements typically document what should be delivered and may provide the contractual basis for the development.
- For managers the requirements may provide the basis for scheduling and a yardstick for measuring progress.
- For the software designers, the requirements may provide the "design–to" specification.
- For coders, the requirements define the range of acceptable implementations and are the final authority on the outputs that must be produced.
- For quality assurance personnel, the requirements represent the basis for validation, test planning, and verification.

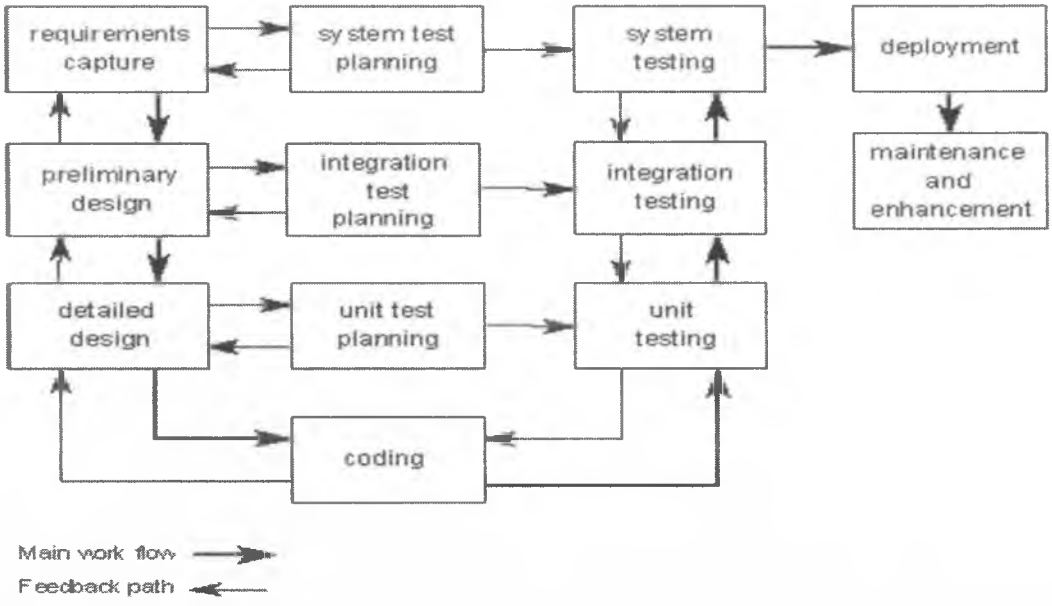


Figure 2: A conventional life-cycle Model

Such diverse groups as marketing and governmental regulators may also use the requirements. These groups and any others with an interest in the outcome of system development are collectively referred to as the system's *stakeholders*.

It is common practice (e.g., see [Thayer 90]) to classify software requirements as "functional" or "non-functional." While definitions vary somewhat in detail, "functional" typically refers to requirements defining the acceptable mappings between system input values and corresponding output values. "Non-functional" then refers to all other constraints including, but not limited to, performance, dependability, maintainability, reusability, and safety.

While widely used, the classification of requirements as "functional" and "non-functional" is confusing in its terminology and of little help in understanding common properties of different kinds of requirements. The word "function" is one of the most overloaded in computer science and its only rigorous meaning, as that of a mathematical function, is not what is meant in this context. The classification of requirements as functional and non–functional offers little help in understanding common attributes of different types of requirements since it partitions classes of requirements with markedly similar qualities (e.g., output values and output deadlines) while grouping others that have common only what they are not (e.g., output deadlines and maintainability goals).

A more useful distinction is between what can be described as "behavioral requirements" and "developmental quality attributes" with the following definitions [Bass 03]:

- *Behavioral requirements* - Behavioral requirements include any and all information necessary to determine if the run–time behavior of a given implementation is acceptable. The behavioral requirements define all constraints on the system outputs (e.g., value, accura-

cy, and timing) and the resulting system state for all possible inputs and the current system state. By this definition, security, safety, performance, timing, and fault–tolerance are all behavioral requirements.

- *Developmental quality attributes* - Developmental quality attributes include any constraints on the attributes of the system's static construction. These include properties like testability, changeability, maintainability, and reusability.

Behavioral requirements have in common that they are properties of the run–time behavior of the system and can (at least in principle) be validated objectively by observing the behavior of the running system, independent of its method of implementation. In contrast, developmental quality attributes are properties of the system's static structures (e.g., modularization) or representation. Developmental quality attributes have in common that they are functions of the development process and methods of construction. Assessment of developmental quality attributes is necessarily relativistic—for example, we do not say that a design is or is not maintainable but that one design is more maintainable than another.

In addition, there may be constraints on the development process itself, for example, that the software must reuse certain legacy code, be developed on a particular platform, or be written in a specific language. Such requirements may be collectively referred to as *process requirements* [SWEBOK 04]. Process requirements are often imposed by regulatory agencies or internal company standards.

3. A Big Problem

Requirements problems are persistent, pervasive, and costly. Evidence is most readily available for the large software systems developed for the U.S. Government since the results are a matter of public record. As soon as software became a significant part of such systems, developers identified requirements as a major source of problems. For example, developers of the early Ballistic Missile Defense System noted that:

> *In nearly every software project that fails to meet performance and cost goals, requirements inadequacies play a major and expensive role in project failure* [Alford 79].

Nor has the problem been mitigated over the intervening years. A study of problems in mission-critical defense systems identified requirements as a major problem source in two-thirds of the systems examined [GAO 92]. This is consistent with results of a survey of large aerospace firms that identified requirements as the most critical software development problem [Faulk 92]. Likewise, studies by Lutz identified functional and interface requirements as the major source of safety-related software errors in NASA's Voyager and Galileo spacecraft [Lutz 92]. The Government Accounting Office (GAO) again identified requirements as a major issue in defense acquisition [GAO 04]. Requirements errors have also been cited as a major cause in the very public losses of the Mars Climate Orbiter and Mars Polar Lander spacecraft [Bahill 05].

Results of industry studies described by Boehm, and since replicated a number of times, showed that requirements errors are the most costly [Boehm 81]. These studies all produced the same basic result: the earlier in the development process an error occurs and the later the error is detected, the more expensive it is to correct. Moreover, the relative cost rises quickly. As shown in Figure 3, an error that costs a dollar to fix in the requirements phase may cost 100 to 200 dollars to fix if it is not corrected until the system is fielded or in the maintenance phase.

The costs of such failures can be enormous. For example, a 1992 GAO report noted that one system, the Cheyenne Mountain Upgrade, would be delivered eight years late and exceed budget by $600 million with less capability than originally planned, largely due to requirements-related problems. Recently, requirements problems have been cited in cost overruns projected for the 2010 Census of up to $2 billion [GAO 08]. Broader GAO reviews (e.g., of troubled weapons programs) suggest that such problems are the norm rather than the exception [GAO 10]. While data from private industry is less readily available, there is little reason to believe that the situation in that sector is significantly different.

Table 1: Relative cost to repair a requirements error

Stage	Relative Repair Cost
Requirements	1-2
Design	~ 5
Coding	~ 10
Unit test	~ 20
System test	~ 50
Maintenance	~ 200

In spite of advances in software engineering methodology and tool support, the requirements problem has not diminished. This does not mean that the apparent progress in software engineering is illusory. While the features of the problem have not changed, the applications have grown significantly in capability, scale, and complexity. A reasonable conclusion is that the growing ambitiousness of our software systems has outpaced the gains in requirements technology; at least as such technology is applied in practice.

4. Why are Requirements so Hard?

It is generally agreed that the goal of the requirements phase is to establish and specify precisely what the software must do without describing how to do it. So simple seems this basic intent that it is not at all evident why it is so difficult to accomplish in practice. If what we want to achieve is so clear, why is it so hard? To understand this, we must examine more closely the goals of the requirements phase, where errors originate, and why the nature of the task leads to some inherent difficulties.

Most authors agree in principle that requirements should specify "what" rather than "how." In other words, the goal of requirements is to understand and specify the *problem* to be solved rather than the *solution*. For example, the requirements for an automated teller system should talk about customer accounts, deposits, and withdrawals rather than software algorithms and data structures. The most basic reason for this is that a specification in terms of the problem captures the actual requirements without over-constraining the subsequent design or its implementation. Furthermore, solutions in software terms are typically more complex, more difficult to change, and harder to understand (particularly for the customer) than a specification of the problem.

Unfortunately, distinguishing "what" from "how" in itself represents a dilemma. As Davis among others points out, the distinction between what and how is necessarily a function of perspective [Davis 88]. A specification at any chosen level of system decomposition can be viewed as describing the "what" for the next level. Thus, customer needs may define the "what" while the decomposition into hardware and software specifies the corresponding "how." Subsequently, the behavioral requirements allocated to a software component define its "what," the software design determines the "how," and so on. In other words, one person's design becomes the next person's requirements.

The upshot is that requirements cannot be effectively discussed at all without prior agreement on which system one is talking about and at what level of decomposition. One must agree on what constitutes the *problem space* and what constitutes the *solution space*—the analysis and specification of requirements then properly belong in the problem space.

In discussing requirements problems one must also distinguish the development of large, complex systems from smaller efforts (e.g., developments by a single or small team of programmers). Large system developments are multi–person efforts. They are developed by teams of tens to thousands of programmers. The programmers work in the context of an organization that typically includes management, systems engineering, marketing, accounting, and quality assurance. The organization itself must operate within the context of outside stakeholders who are also interested in the software product, including the customer, regulatory agencies, and suppliers.

Even in cases in which only one system is intended, large systems inevitably become multi–version as well. Software evolves as it is being developed, tested, and even fielded. Customers better understand what they want and developers better understand what they can and cannot do within the constraints of cost and schedule, all while circumstances surrounding development change. The results are changes in the software requirements and, ultimately, the software itself. In effect, several versions of a given program are produced, if only incrementally. Such unplanned changes occur in addition to the expected variations of planned improvements.

The multi-person, multi-version nature of large system development introduces problems that are both quantitatively and qualitatively different from those found in smaller developments. For example, scale introduces the need for administration and control functions with the attendant management issues that do not exist in small projects. The quantitative effects of increased complexity in communication with an increased number of workers are well documented by Brooks [Brooks 95]. The effort required for communication and other overhead tasks such as documentation or configuration management tends to rise exponentially with the size and complexity of the system. The following discussion is written within this large system development context since that is where the worst problems occur and where the most help is needed.

Given the context of multi–person, multi–version development, our basic goal of specifying what the software must do is decomposed into the following subgoals:

- Understand precisely what is required of the software.
- Communicate the understanding of what is required to each party involved in the project development.
- Control the software production to ensure that the final system satisfies the requirements (including managing the effects of changes).

It follows that the source of most requirements errors lies in the failure to adequately accomplish one of these goals, i.e.:

- The developers failed to understand what was required of the software by the customer, end user, or other parties with a stake in the final product.
- The developers did not adequately capture the requirements or subsequently communicate the requirements effectively to other parties involved in the development.
- The developers did not effectively manage the effects of changing requirements or ensure the conformance of downstream development steps, including design, code, integration, test, or maintenance, to the system requirements.

The end result of such failures is a software system that does not perform as desired or expected, a development that exceeds budget and schedule, or, all too frequently, failure to deliver any working software at all.

4.1 Essential Difficulties

Even our more detailed goals appear reasonably straightforward. Why then, do so many development efforts fail to achieve the desired goals? The short answer is that the *mutual* satisfaction of these goals, in practice, is inherently difficult. To understand why, it is useful to reflect on some points raised by Brooks as to why software engineering is hard and on the distinction he makes between essential difficulties – those inherent in the problem, and the accidental difficulties -- those introduced through imperfect practice [Brooks 87]. For though requirements are inherently difficult, there is no doubt that these difficulties are many times multiplied by the inadequacies of current practice.

The following essential difficulties attend each (in some cases all) of the requirements goals:

- *Comprehension.* People do not know what they want. This does not mean that people do not have a general idea of what the software is for. Rather, they do not begin with a precise and detailed understanding of what functions belong in the software, what the output must be for every possible input, how long each operation should take, how one decision will affect another, and so on. Indeed, unless the new system is simply a reconstruction of an old one, such a detailed understanding at the outset is unachievable. Many decisions about the system behavior will depend on other decisions yet unmade, and expectations will change as the problem (and attendant costs of alternative solutions) is better understood. Nonetheless, it is a precise and richly detailed understanding of expected behavior that is needed to create effective designs and to develop correct code.
- *Communication.* Software requirements are difficult to communicate effectively. As Brooks points out, the conceptual structures of software systems are complex, arbitrary, and difficult to visualize. The large software systems now being built are among the most complex structures ever attempted. That complexity is arbitrary in the sense that it is an artifact of people's decisions and prior construction rather than a reflection of fundamental properties (as in, for example, the case of physical laws). To make matters worse, many of the conceptual structures in software have no readily comprehensible physical analogue so they are difficult to visualize.

  In practice, comprehension suffers under all of these constraints. We work best with regular, predictable structures, can comprehend only a very limited amount of infor-

mation at one time, and understand large amounts of information best when we can visualize it. Thus, the task of capturing and conveying software requirements is inherently difficult.

The inherent difficulty of communication is compounded by the diversity of purposes and audiences for a requirements specification. Ideally, a technical specification is written for a particular audience. The brevity and comprehensibility of the document depend on assumptions about common technical background and use of language. Such commonality typically does not hold for the many diverse groups (e.g., customers, systems engineers, managers) that must use a software requirements specification.

- *Control.* Inherent difficulties attend control of software development as well. The arbitrary and invisible nature of software makes it difficult to anticipate which requirements will be met easily and which will decimate the project's budget and schedule if, indeed, they can be fulfilled at all. The low fidelity of software planning has become a cliché, yet the requirements are often the best available basis for planning or for tracking to a plan.

  This situation is made incalculably worse by software's inherent malleability. Of all the problems bedeviling software mangers, few evoke such passion as the difficulties of dealing with arbitrary requirements changes. For most systems, such changes remain a fact of life even after delivery. The continuous changes make it difficult to develop stable specifications, plan effectively, or control cost and schedule. For many industrial developers, change management is the most critical problem with regard to requirements.

- *Inseparable concerns.* In seeking solutions to the foregoing problems, we are faced with the additional difficulty that the issues cannot easily be separated and dealt with piecemeal. For example, developers have attempted to address the problem of changing requirements by baselining and freezing requirements before design begins. This proves impractical because of the comprehension problem—the customer may not fully know what he wants until he sees it. Similarly, the diversity of purposes and audiences is often addressed by writing a different specification for each. Thus, there may be a system specification, a set of requirements delivered to a customer, a distinct set of technical requirements written for the internal consumption of the software developers, and so on. However, this solution vastly increases project complexity, provides an open avenue for inconsistencies, and multiplies the difficulties of managing changes.

  These issues represent only a sample of the inherent dependencies between different facets of the requirements problem. The many distinct parties with an interest in a system's requirements, the many different roles the requirements play, and the interlocking nature of software's conceptual structures all introduce dependencies among concerns and impose conflicting constraints on any potential solution.

  The implications are twofold. First, we are constrained in the application of our most effective strategy for dealing with complex problems—divide and conquer. If a problem is considered in isolation, the solution is likely to aggravate other difficulties. Effective solutions to most requirements difficulties must simultaneously address more than one problem. Second, developing practical solutions requires making difficult tradeoffs. Where different problems have conflicting constraints, compromises must be made. Because the tradeoffs result in different gains or losses to the different parties involved, ef-

fective compromise requires negotiation. These issues are considered in more detail when we discuss the properties of a good requirements specification.

4.2 Accidental Difficulties

While there is no doubt that software requirements are inherently difficult to do well, there is equally no doubt that common practice unnecessarily exacerbates the difficulty. We use the term "accidental" in contrast to "essential," not to imply that the difficulties arise by chance, but that they are the product of common failings in management, elicitation, specification, or use of requirements. It is these failings that are most easily addressed by improved practice.

- *Written as an afterthought.* It remains common practice that requirements documentation is developed only after the software has been written. For many projects, the temptation to rush into implementation before the requirements are adequately understood proves irresistible. This is understandable. Developers often feel like they are not really doing anything when they are not writing code; managers are concerned about schedule when there is no visible progress toward project implementation. Then too, the intangible nature of the product mitigates toward early implementation. Developing the system is an obvious way to better understand what is needed and make visible the actual behavior of the product. The result is that requirements specifications are written as an afterthought (if at all). They are not created to guide the developers and testers, but are instead treated as a necessary evil to satisfy contractual demands.

  Such after-the-fact documentation inevitably violates the principle of defining "what" the system must do rather than "how," since it is a specification of the code as written. Because it is produced after the fact, it is not planned or managed as an essential part of the development but rather is thrown together. In fact, it is not even available in time to guide project implementation or to manage production.

- *Confused in purpose.* Because there are so many potential audiences for a requirements specification with different points of view, the exact purpose of the document becomes confused. An early version is used to sell the product to the customer so it includes marketing hype extolling the product's virtues. As the only documentation of the system, it provides introductory, explanatory, and overview material. It is a contractual document so it is intentionally imprecise to allow the developer latitude in the delivered product or the customer latitude in making no–cost changes. It is the vehicle for communicating decisions about software to designers and coders, so it incorporates design and implementation details. The result is a document in which it is unclear which statements represents real requirements and which are more properly allocated to marketing, design, or other documentation. It is a document that attempts to be everything to everyone and ultimately serves no one well.

- *Not designed to be useful.* Often in the rush to implementation, little effort is expended on requirements. The requirements specification is not expected to be useful and, indeed, this turns out to be a self–fulfilling prophecy. Little effort is expended on designing it, writing it, checking it, or managing its creation and evolution. The most obvious result is poor organization. The specification is written in English prose and follows either the author's stream of consciousness or the order of execution [Heninger 80].

The resulting document is ineffective as a technical reference. It is unclear which statements represent actual requirements. It is unclear where to put or find particular requirements. There is no effective procedure for ensuring that the specification is consistent or complete. There is no systematic way to manage requirements changes. The specification is difficult to use and difficult to maintain. It quickly becomes out of date and loses whatever usefulness it might originally have had.

- *Lacks essential properties.* Lack of forethought, confusion of purpose, or lack of careful design and execution all lead to requirements that lack properties critical to good technical specifications. The requirements, if documented at all, are redundant, inconsistent, incomplete, imprecise, and inaccurate.

While the essential difficulties are inherent in the problem, the accidental difficulties result from a failure to gain or maintain intellectual control over what is to be built. While the presence of the essential difficulties means that there can be no "silver bullet" that will suddenly render requirements easy, we can remove at least the accidental difficulties through a well-thought-out, systematic, and disciplined development process. Such a disciplined process then provides a stable foundation for attacking the essential difficulties.

5. Role of a Disciplined Approach

The application of discipline in analyzing and specifying software requirements can address the accidental difficulties. While there is considerable agreement on the desirable qualities of a software development approach, development processes have not been standardized. Further, the context and qualities of development can differ such that no single process model will suit all developments. Nonetheless, it is useful to examine the characteristics of an idealized process and its products to understand weaknesses in current approaches and which current trends are promising. In general, a complete requirements approach will define:

- *Process:* The (partially ordered) sequence of activities, entrance and exit criteria for each activity, which work products are produced in each activity, and what skill sets are needed to do the work.

- *Products:* The work products to be produced and, for each product, the resources needed to produce it, the information it contains, the expected audience, and the acceptance criteria the product must satisfy.

Conceptually, the requirements phase consists of two distinct but overlapping activities corresponding to the first two goals for requirements previously enumerated:

1. *Problem analysis:* The goal of problem analysis is to understand precisely what problem is to be solved. It includes identifying the system's stakeholders and eliciting their requirements. It also includes deciding the exact purpose of the system, who will use it, the constraints on acceptable solutions, and the possible tradeoffs between conflicting constraints.

2. *Requirements specification:* The goal of requirements specification is to capture the results of problem analysis in a transferable form. The products of this activity typically include a written specification of precisely what is to be built in the form of a Software Requirements Specification (SRS). The SRS captures the decisions made during problem analysis and characterizes the set of acceptable solutions to the problem.

In practice, the distinction between these activities is conceptual rather than temporal. Where both are needed, the developer typically switches back and forth between analysis of the problem and documentation of the results. When problems are well understood, the analysis phase may be virtually non–existent. When the system model and documentation are standardized or based on existing specifications, the documentation paradigm may guide the analysis [Hester 81].

5.1 Problem Analysis

Problem analysis lies at the boundary between human concerns and the realization of some software system that seeks to address those concerns. It is necessarily informal in the sense that there is no effective, closed-end procedure that will guarantee success. It is a process of acquiring, collating, and structuring information, through which one attempts to understand all the various parts of a problem and their relationships.

Problem analysis may be further divided into two closely related sub-activities: *requirements elicitation* and *requirements modeling and analysis*. Requirements elicitation focuses on the human side of problem analysis. It seeks to answer the question "What are the behavioral and developmental qualities of an acceptable system?" Modeling and analysis supports elicitation by capturing the answers to this question in a form that allows the stakeholders to understand, communicate, and reason about the results.

5.2 Requirements Elicitation

As our discussion of the essential difficulties suggests, understanding what constitutes an "acceptable system" to its stakeholders can be a daunting task. People do not really know what they want in sufficient detail. Moreover, different people or types of stakeholders often have different and incompatible views of the problem, the purposes for developing the system, and what it should accomplish. In fact, since the scope of the system may be undetermined, it may not even be clear who the stakeholders are.

The purpose of a disciplined elicitation process is to systematically remove the uncertainty from problem understanding, resolve conflicting views, and arrive at a set of behavioral and developmental requirements that the stakeholders will agree to. To do so, the process must answer the following questions:

- What are the system boundaries?
- What is the rationale for creating the system? What are the current problems and what are the goals for the proposed system?
- What are the constraints on acceptable solutions?
- Who are the stakeholders?
- What are the different stakeholders' views of the problem and the system requirements?
- Where does the understanding differ or requirements conflict and how can those conflicts be resolved?

Developments differ in the extent to which the process must address such questions. For example, for a project with a single customer, it may be unnecessary to expend any effort establishing who the stakeholders are or managing stakeholder conflicts. Thus, the activities necessary to answering these questions are incorporated into the elicitation process as needed.

Establish system boundaries: The purpose of this activity is to establish where system concerns properly begin and end. In practice, this means characterizing the system's external interfaces. It delimits and defines how the software interacts with users or with other systems (software or hardware).

In addition, establishing the system boundaries sets boundaries on the elicitation process itself. By defining what is inside the system and what is outside, the scope of inquiry about the problem and the system requirements is bounded. By identifying which concerns properly belong to the software it helps establish who the stakeholders are and which views or concerns are relevant. By establishing bounds on which persons and issues are relevant. it helps determine when elicitation is done.

Rationale and goal understanding: Fully understanding the problem requires understanding the rationale - why the system is being built in the first place. Understanding the rationale can be necessary for establishing system requirements and for maintaining consistency as real-world objectives or constraints change over time.

The rationale encompasses both the problems with any current system (automated or manual) and the objectives for the new system. System objectives may be codified in the form of goals where a *goal* characterizes "an objective the system under consideration should achieve" [Lamsweerde 01].

Goals provide a link between broader concerns like business objectives and the requirements that instantiate those concerns in the software context. Defining goals and providing traceability to the software requirements supports managing requirements changes as business objectives mature. Likewise, understanding the overall system goals and their relative priorities provides a basis for choosing among likely alternatives and resolving conflicting requirements. Specific approaches to goal-based requirements are discussed in the subsequent section on the state of practice.

Stakeholder identification: fully understanding the problem necessitates identifying all of the system's stakeholders, then understanding their interest in the system. In stakeholder identification, it is important to include both the individuals (and organizations) who stand to lose, as well as those who stand to gain, from development success or failure [Gause 89].

For many large developments it is not immediately obvious who all the stakeholders are, even to the stakeholders themselves. Further, the set of stakeholders may change as requirements evolve, system boundaries change, or the individual filling those organizational roles are replaced.

Since different stakeholders will have different attributes, concerns, and views of the system, identifying them is a necessary step toward selecting appropriate elicitation methods, gathering a complete set of requirements, establishing priorities, and negotiating conflicts.

Elicitation: The core of requirements elicitation is the process of working with the stakeholders to obtain their understanding of the problem, goals, and system requirements. Since different classes of stakeholders typically have different perspectives about the problem. have different cultures, and communicate in different languages, a number of different elicitation methods may have to be used as part of an effective elicitation process. Determining which methods to use, incorporating them in the requirements process, and synthesizing the results are the concerns of effective practice (e.g., [Lauesen 02]).

Requirements Negotiation: Different stakeholders necessarily have different perspectives on the system requirements. For most real developments, there is no single set of requirements waiting to be discovered. Rather, there are many potential manifestations of stakeholder desires that lead to different, and often conflicting, sets of requirements.

Before development can proceed to implementation, there must be agreement on a single, consistent set of requirements. Modeling and analyzing the requirements can help identify where conflicts occur but does not resolve them. This almost always requires tradeoffs and compromises between conflicting goals. It follows that arriving at agreement requires an effective process for negotiating requirements tradeoffs among stakeholders (e.g., [Boehm 94]).

5.3 Requirements Modeling and Analysis

The inherent difficulties of software complexity and invisibility are typically addressed by developing one or more abstract models. "Model," in this sense, means a representation of some aspect of the software system, the system's context, or both. It is abstract in that it represents certain information (entities and relationships) about the system while omitting others.

The use of models can help make the intangible objects and relationships in a software system visible. For example, a behavioral model might show the required system transitions and the observable behavior in response to user inputs. Such models aid elicitation and understanding by providing a transferable representation of the problem or system requirements. The use of models also reduces complexity by allowing the user to focus on and reason about a limited, related set of information at one time.

That said, not all models or modeling languages are equal. In some cases, "abstract" is interpreted to mean vague, not well defined, or inaccurate. To support reasoning about a system, any model should have the property that anything that is true of the model is also true of the system it represents. One can then manipulate the model to achieve particular developmental goals with the understanding that corresponding transformations to the system will yield corresponding real-world properties. In many cases, modeling languages (e.g., UML) lack sufficiently well-defined semantics to achieve this property. The result is a model that is open to conflicting interpretations.

In addition to supporting problem understanding, the creation of models can support various kinds of analysis. Where models provide a formal syntax and semantics, they may support analysis for properties like consistency and completeness, as well as reasoning about requirements like safety properties. Such analyses can help identify missing requirements, inconsistencies, and requirements conflicts during elicitation. While informal models may not support formal reasoning, they can be useful aids for visualizing and reasoning about system requirements, as long as their limitations are understood.

5.4 Requirements Specification

For substantial developments, the effectiveness of the requirements effort depends on how well the SRS captures the results of analysis and how useable the specification is. There is little benefit to developing a thorough understanding of the problem if that understanding is not effectively communicated to customers, designers, implementers, testers, and other stakeholders. The larger and more complex the system, the more important a good specification becomes. This is a direct result of the many roles the SRS plays in a multi–person, multi–version development [Parnas 86]:

1. The SRS is the primary vehicle for agreement between the developer and customer on exactly what is to be built. It is the document that is reviewed by the customer or his representative and often is the basis for judging fulfillment of contractual obligations.
2. The SRS records the results of problem analysis. It is the basis for determining where the requirements are complete and where additional analysis is necessary. Documenting the results of analysis allows questions about the problem to be answered only once during development.
3. The SRS defines what properties the system must have and the constraints on its design and implementation. It defines where there is, and is not, design freedom. It helps ensure that requirements decisions are made explicitly during the requirements phase, not implicitly during the design or programming phases.
4. The SRS is the basis for estimating cost and schedule. It is management's primary tool for tracking development progress and ascertaining what remains to be done.
5. The SRS is the basis for test plan development. It is the tester's chief tool for determining acceptable software behavior.
6. The SRS provides the standard definition of expected behavior for the system's maintainers and is used to record engineering changes.

For a disciplined software development, the SRS is both the primary technical specification of the software and the primary control document. This is an inevitable result of the complexity of large systems and the need to coordinate multi–person development teams. To ensure that the right system is built, one must first understand the problem. To ensure agreement on what is to be built and the criteria for success, the results of that understanding must be recorded. The goal of a systematic requirements process is thus the development of a set of specifications that effectively communicate the results of analysis. The SRS is the primary vehicle for communicating requirements between the developers, managers, and customers, so the document is designed to be useful for that purpose. A useful document is maintained.

5.5 Requirements Process and Plan

Requirements' accidental difficulties are addressed through the careful analysis and specification of a disciplined process. Rather than developing the specification as an afterthought, requirements are understood and specified before development begins. One knows what one is building before attempting to build it. When requirements cannot be completely known in advance, the process systematically revisits the requirements process and downstream activities (e.g., iterative development).

The facts that requirements cannot be fully known in advance, and often change, are sometimes used as justification for expending little effort toward requirements planning. The thought is that the project will deal with requirements when and if they become manifest. Such an approach surrenders the notion of a controlled engineering process to chance.

By definition, as a system enters the coding phase every decision about the requirements necessarily gets made. The question is not whether any particular requirements decision will be made but when it will be made and by whom. By default, any decision that is not made earlier in the process will be made by the programmers. In many cases, the programmers have little

visibility into the business implications of such decisions or their effects on stakeholder goals. This is seldom a desirable outcome.

Being in control of the process means that requirements decisions, including postponing or not making decisions, are conscious choices. Each decision is made at the appropriate time by those with the knowledge and skills necessary to choose the best available alternative. This kind of control requires that the complex activities related to requirements be planned in advance.

While organizations that develop complex software systems should employ a disciplined requirements process, no one process will meet the needs of every organization. A company that is developing an application in which development cost and time to market are primary business drivers should not use the same process as an organization developing safety-critical aerospace software with a long life expectancy.

It follows that the requirements process is something that should be chosen or designed to fit the organizational and even developmental contexts. While every development will typically go through some form of elicitation, modeling, analysis, and specification, the emphasis on the different phases and products will differ from one situation to the next. Likewise, the choices among methods, technologies, notations, and tools will vary.

We then must create (build) or choose (buy) a process that satisfies the requirements. We must compare the process to the goals, verify its enactment, and so on. We must communicate that process to those who will enact it, manage it, or monitor it. We must validate the process against the goals, verify its enactment, and so on.

In a disciplined organization, this means that there must be a written specification that records decisions about the process and provides a baseline for enactment, tailoring, or process improvement. While treating a process as a product in this manner may seem alien, in fact many organizations that have embarked on systematic process improvement (e.g., [SEI 06]) have done all of this and more. Thinking about the process as a product helps ensure that adequate consideration is given to planning, budgeting for, and managing process development or improvement.

At the project level, the requirements process should be instantiated in the form of a *requirements plan* [Young 04]. The requirements plan makes the abstract requirements processes concrete by mapping activities to tasks, people to roles, and artifacts to deliverables. It describes who will do what and how. For example, it should describe which elicitation methods will be used to obtain which kinds of requirements information and which modeling methods will be used to capture that information.

The plan serves as the basis for team consensus on exactly what will be done, provides a yardstick for tracking progress, and serves as a guide to new personnel and other stakeholders. The exact plan contents should vary depending on the organization's process and the specific characteristics of the project. In general, however, it should answer the following kinds of questions for the reader:

- Roles and Responsibilities —Who is responsible for what?
- Project Background —What background information will help us understand this project?
- Requirements Process —What idealized requirements process will we follow?

- Mechanisms, methods, techniques — How will we elicit, identify, analyze, define, specify, prioritize, track, etc.?
- Quality assessment — What methods will be used to assess requirements qualities and what are the acceptance criteria for the products produced?
- Detailed schedule, milestones — How are the activities and artifacts mapped to the project schedule and milestones?
- Resources and references — Who or what resources can answer questions about the product or process?

The instantiation of a well-defined process in the project plan helps ensure that the process actually enacted by project personnel will be consistent with the organization's overall process goals. Observing and measuring the results then provides metrics for systematic process improvement.

The final key to implementing the plan is providing adequate resources. Historical data from a large set NASA projects shows that, in general, the projects that spent the least on developing requirements tended to have the highest cost overruns. Projects that spend 8% to 14% of the total project budget on acquiring and managing requirements reduced cost overruns by 50% ([NASA 05], [Young 06]).

6. Requirements for the Software Requirements Specification

The goals of the requirements process, the attendant difficulties, and the role of the requirements specification in a disciplined process determine the properties of a "good" requirements specification. These properties do not mandate any particular specification method but do describe characteristics an effective method should possess.

The semantic properties determine how effectively an SRS captures the software requirements. The packaging properties determine how useable the resulting specification is and illustrates the classification of properties of a good SRS (see Table 2). An SRS that satisfies the semantic properties of a good specification is:

- *Complete.* The SRS defines the set of acceptable implementations. It should contain all the information needed to write software that is acceptable to the customer and no more. Any implementation that satisfies every statement in the requirements is an acceptable product. Where information is not available before development begins, areas of incompleteness must be explicitly indicated [Parnas 86].
- *Implementation independent.* The SRS should be free of design and implementation decisions unless those decisions reflect actual requirements.
- *Unambiguous and consistent.* If the SRS is subject to conflicting interpretation, the different parties will not agree on what is to be built or whether the right software has been built. Every requirement should have only one possible interpretation. Similarly, no two statements of required behavior should conflict.
- *Precise.* The SRS should precisely define the required behavior. For each output, it should define the range of acceptable values for every input. The SRS should define any applicable timing constraints such as minimum and maximum acceptable delay.

Table 2: Semantic vs. packaging properties

SRS Semantic Properties	SRS Packaging Properties
Complete Implementation independent	Modifiable Readable
Unambiguous and consistent, precise and verifiable	Organized for reference and review

- *Verifiable.* A requirement is verifiable if it is possible to determine unambiguously whether a given implementation satisfies the requirement or not. For example, a behavioral requirement is verifiable if it is possible to determine, for any given test case (i.e., an input and an output), whether the output represents an acceptable behavior of the input and the system state.

An SRS that satisfies the packaging properties of a good specification is:

- *Modifiable.* The SRS must be organized for ease of change. Since no organization can be equally easy to change for all possible changes, the requirements analysis process must identify expected changes and the relative likelihood of their occurrence. The specification is then organized to limit the effect of likely changes.
- *Readable.* The SRS must be understandable by the parties that use it. It should clearly relate the elements of the problem space as understood by the customer to the observable behavior of the software.
- *Organized for reference and review.* The SRS is the primary technical specification of the software requirements. It is the repository for all the decisions made during analysis about what should be built. It is the document reviewed by the customer or his representatives. It is the primary arbitrator of disputes. As such, the document must be organized for quick and easy reference. It must be clear where each decision about the requirements belongs. It must be possible to answer specific questions about the requirements quickly and easily.

To address the difficulties associated with writing and using an SRS, a requirements approach must provide techniques addressing both semantic and packaging properties. It is also desirable that the conceptual structures of the approach treat the semantic and packaging properties as distinct concerns (i.e., as independently as possible). This allows one to change the presentation of the SRS without changing its meaning.

In aggregate, these properties of a good SRS represent an ideal. Some of the properties may be unachievable, particularly over the short term. For example, a common complaint is that one cannot develop complete requirements before design begins because the customer does not yet fully understand what he wants or is still making changes. Further, different SRS "requirements" mitigate toward conflicting solutions. A commonly cited example is the use of English prose to express requirements. English is readily understood but notoriously ambiguous and imprecise. Conversely, formal languages are precise and unambiguous, but can be difficult to read.

Although the ideal SRS may be unachievable, possessing a common understanding of what constitutes an ideal SRS is important [Parnas 86] because it:

- Provides a basis for standardizing an organization's processes and products,
- Provides a standard against which progress can be measured, and,
- Provides guidance - it helps developers to understand what needs to be done next and to know when they are finished.

Because it is so often true that (1) requirements cannot be fully understood before at least starting to build the system, and (2) a perfect SRS cannot be produced even when the requirements are understood, some approaches advocated in the literature do not even attempt to produce a definitive SRS. For example, some authors advocate going directly from a problem model to design or from a prototype implementation to the code. While such approaches may be effective on some developments, they are inconsistent with the notion of software development as an *engineering* discipline. The development of technical specifications is an essential part of a controlled engineering process. This does not mean that the SRS must be complete or perfect before anything else is done but that its development is a fundamental goal of the process as a whole. That we may currently lack the ability to write good specifications in some cases does not change the fact that it is useful and necessary to try.

7. State of the Practice

The past decade has brought a significant shift in requirements practice and the perception of the role of requirements in the development process. When the first version of this article was published, requirements analysis was generally treated as a distinct concern (e.g., [Davis 93]). There was the conceptual distinction that requirements should express an implementation-independent specification of what the software should do. However, it was also treated as a development phase that divided the software process into distinct and relatively independent parts. It is this sequencing relationship that is represented in the Waterfall model and its variations [e.g., Figure 1].

In this view, the requirements phase begins with requirements gathering, and ends with the delivery of some form of requirements specification to the software designers. While it is understood that the requirements activities and its products may be revisited in subsequent phases, it is assumed that the requirements specification can capture and communicate everything the developers need to know to design, implement, and maintain the software. In practice, this separation of concerns was embodied in the notion of the "requirements handoff" – a process milestone in which the requirements specification is baselined and control is passed to the software designers and coders.

The unstated assumption behind this model is that the dependencies between non-contiguous parts of the process do not require explicit understanding or management; that everything the stakeholders need to know can be captured through work products like the SRS and supporting traceability matrices. Thus, for example, the designers do not need to understand the source of particular requirements or the underlying business rationale to design good software architecture.

Over the past decade, a more holistic view of the software process has emerged. It has become clear that, for most complex software development, the decisions made in each phase of development may have significant implications across the life cycle and, indeed, across more

than one life cycle. Thus, controlling the downstream effects of development decisions requires explicit understanding and management of these dependencies. This requires a model of development that spans the software life cycle and, for some concerns, multiple life cycles.

In the remainder of this section we discuss the current state of practice, particularly as it embodies this broader, more interdisciplinary view of requirements.

7.1 Software Methodologies

Over the years, a number of analysis and specification methods have been developed as part of more comprehensive software engineering methods. The general trend has been for software engineering techniques to be applied first to coding problems (e.g., complexity, ease of change), then to similar problems occurring earlier and earlier in the life cycle. Thus the concepts of structured programming eventually led to structured design and analysis. Similarly, the concepts of object-oriented programming led to object-oriented design and analysis.

The benefits of this approach are that a common set of conceptual structures and notations can be used across the software life cycle. It is unnecessary to translate from one set of abstractions to another (until code is produced), avoiding translation errors, and inconsistencies between models. The drawback is that the same notations and structures must be used to represent concepts that we are trying to keep distinct. For example, the concept of *objects* is used to represent both entities in the problem domain (requirements) and entities in the implementation domain (code). This can make it difficult to distinguish requirements decisions from downstream concerns.

Since a number of the concepts used in current object-oriented approaches were introduced in Structured Analysis, and since Structured Analysis is still in use in use in some application domains, our discussion will treat both.

7.1.1 Structured Analysis (SA)

Following the introduction of structured programming as a means to gain intellectual control over increasingly complex programs, structured analysis evolved from functional decomposition as a means to gain intellectual control over system problems.

The basic assumption behind SA is that the accidental difficulties can be addressed by a systematic approach to problem analysis using [Svoboda 90]:

- A common conceptual model for describing all problems,
- A set of procedures suggesting the general direction of analysis and an ordering on the steps,
- A set of guidelines or heuristics supporting decisions about the problem and its specification, and
- A set of criteria for evaluating the quality of the product.

While functional decomposition is still a part of SA, the focus of the analysis shifts from the processing steps to the data being processed. The analyst views the problem as constructing a system to transform data. He analyzes the sources and destinations of the data, determines what data must be held in storage, what transformations are done on the data, and the form of the output.

Common to the SA approaches is the use of data flow diagrams and data dictionaries. Data flow diagrams provide a graphic representation of the movement of data through the system (typically represented as arcs) and the transformations on the data (typically represented as nodes). The data dictionary supports the data flow diagram by providing a repository for the definitions and descriptions of each data item on the diagrams. Required processing is captured in the definitions of the transformations. Associated with each transformation node is a specification of the processing the node does to transform the incoming data items to the outgoing data items. At the most detailed level, a transformation is defined using a textual specification called a "minispec." A minispec may be expressed in a number of different ways, including English prose, decision tables, or a procedure definition language (PDL).

SA approaches originally evolved for management information systems (MIS). Examples of widely used strategies include those described by DeMarco [1978] and Gane and Sarson [1979]. Later "Modern" structured analysis was introduced to provide more guidance in modeling systems as data flows, as exemplified by Yourdon [1989].

Structured analysis is based on the notion that there should be a systematic (and hopefully predictable) approach to analyzing a problem, decomposing it into parts, and describing the relationships between the parts. By providing a well-defined process, structured analysis seeks to address, at least in part, the accidental difficulties that result from ad hoc approaches and the definition of requirements as an afterthought. It seeks to address problems in comprehension and communication by using a common set of conceptual structures and a graphic representation of the specification in terms of those structures, based on the assumption that a decomposition, in terms of the data the system handles, will be clearer and less inclined to change than one based on the functions performed.

While structured analysis techniques have continued to evolve and have been widely used, there remain a number of common criticisms. When used in problem analysis, a common complaint is that structured analysis provides insufficient guidance. Analysts have difficulty deciding which parts of the problem to model as data, which parts to model as transformations, and which parts should be aggregated. While the gross steps of the process are reasonably well defined, there is only very general guidance (in the form of heuristics) as to what specific questions the analyst needs to answer next. Similarly, practitioners find it difficult to know when to stop decomposition and addition of detail. In fact, the basic structured analysis paradigm of modeling requirements as data flows and data transformations requires the analyst to make decisions about intermediate values (e.g., form and content of stored data and the details of internal transformations) that are not requirements. Particularly in the hands of less experienced practitioners, data flow models tend to incorporate a variety of detail that properly belongs to design or implementation.

Many of these difficulties result from the weak constraints imposed by the conceptual model. A goal of the developers of structured analysis was to create a very general approach to modeling systems; in fact, one that could be applied equally to model human enterprises, hardware applications, software applications of different kinds, and so on. Unfortunately, such generality can be achieved only by abstracting away any semantics that are not common to all types of systems potentially being modeled. The conceptual model itself can provide little guidance relevant to a particular system. Since the conceptual model applies equally to both requirements analysis and design analysis, its semantics provide no basis for distinguishing between the two. Similarly, such models can support only very weak syntactic criteria for assessing the quality of structured

analysis specifications. For example, the test for completeness and consistency in data flow diagrams is limited to determining that the transformations at each level are consistent in name and number with the data flows of the level above.

This does not mean one cannot develop data flow specifications that are easy to understand, communicate effectively with the user, or capture required behavior correctly. The large number of systems developed using structured analysis show that it is possible to do so. However, the weakness of the conceptual model means that a specification's quality depends largely on the experience, insight, and expertise of the analyst. The analyst must provide the necessary discipline because the model itself is relatively unconstrained.

Finally, structured analysis provides little support for producing an SRS meeting our quality criteria. Data flow diagrams are unsuitable for capturing mathematical relations or detailed specifications of value, timing, or accuracy. Therefore, detailed behavioral specifications are typically given in English or as pseudo–code segments in the minispec. These constructs provide little or no support for writing an SRS that is complete, implementation independent, unambiguous, consistent, precise, and verifiable. Further, the data flow diagrams and attendant dictionaries do not, by themselves, provide support for organizing an SRS to satisfy the packaging goals of readability, ease of reference and review, or reusability. In fact, for many of the published methods, there is no explicit process step, structure, or guidance for producing an SRS at all as a distinct development product.

7.1.2 Object-Oriented Analysis (OOA)

OOA has evolved from at least two significant sources: information modeling and object-oriented design. Each has contributed to current views of OOA, and the proponents of each emphasize somewhat different sets of concepts. OOA techniques differ from structured analysis in their approach to decomposing a problem into parts and in the methods for describing the relationships between the parts. In OOA, the analyst decomposes the problem into a set of interacting objects based on the entities and relationships extant in the problem domain. An object encapsulates a related set of data, processing, and state. (Thus, a significant distinction between object-oriented analysis and structured analysis is that OOA encapsulates both data and related processing together.)

The structural components of OOA (e.g., objects, classes, services, and aggregation) support a set of analytic principles. Of these structural components, two directly address requirements problems:

1. From information modeling comes the assumption that a problem is easiest to understand and communicate if the conceptual structures created during analysis map directly to entities and relationships in the problem domain. This principle is realized in OOA through the heuristic of representing problem domain objects and relationships of interest as OOA objects and relationships. Thus an OOA specification of a vehicle registration system might model vehicles, vehicle owners, vehicle titles, and so on as objects. The object paradigm is used to model both the problem and the relevant problem context.
2. From early work on modularization by Parnas [Parnas 72] and abstract data types, and by way of object-oriented programming and design, come the principles of information hiding and abstraction. The principle of information hiding guides one to limit access to information on which other parts of the system should not depend. In an OO specification

of requirements, this principle is applied to hide details of design and implementation. In OOA, behavior requirements are specified in terms of the data and services provided on the object interfaces; the object encapsulates how those services are implemented. The principle of abstraction says that only the relevant or essential information should be presented. Abstraction is implemented in OOA by defining object interfaces that provide access only to essential data or state information encapsulated by an object (conversely hiding the accidentals).

The principles and mechanisms of OOA provide a basis for attacking the essential difficulties of comprehension, communication, and control. The principles of problem domain modeling help guide the analyst in distinguishing requirements (what) from design (how). Where the objects and their relationships faithfully model entities and relationships in the problem, they are understandable by the customer and other domain experts; this supports early comprehension of the requirements.

The principles of information hiding and abstraction, with the attendant object structures, provide mechanisms useful for addressing the essential problems of control and communication. Objects provide the means to divide the requirements into distinct parts, abstract from details, and limit unnecessary dependencies between the parts. Object interfaces can be used to hide irrelevant detail and define abstractions providing only the essential information. This provides a basis for managing complexity and improving readability. Likewise objects provide a basis for constructing reusable requirements units of related functions and data.

The potential benefits of OOA are often diluted by the way the key principles are manifested in particular methods. While the objects and relations of OOA are intended to model essential aspects of the application domain, this goal is typically not supported by a corresponding conceptual model of the domain behavior. Object modeling mechanisms and techniques are intentionally generic rather than application specific. One result is insufficient guidance in developing appropriate object decompositions. OOA practitioners often have difficulty choosing appropriate objects and relationships.

In practice, the notion that one can develop the structure of a system, or a requirements specification, based on physical structure is often oversold. It is true that the elements of the physical world are usually stable (especially relative to software details) and that real-world-based models have intuitive appeal. It is not true; however, that everything that must be captured in requirements has a physical analog. An obvious example is shared state information. Further, many real world structures are themselves arbitrary and likely to change (e.g., where two hardware functions are put on one physical platform to reduce cost). While the notion of basing requirements structure on physical structure is a useful heuristic, more is needed to develop a complete and consistent requirements specification.

A further difficulty is that the notations and semantics of OOA methods are typically based on the conceptual structures of software rather than those of the problem domain the analyst seeks to model. Symptomatic of this problem is that analysts find themselves debating about object language features and their properties rather than about the properties of the problem. An example is the use of message passing, complete with message passing protocols, where one object uses information defined in another. In the problem domain it is often irrelevant whether information is actively solicited or passively received. In fact there may be no notion of messages or transmission at all. Nonetheless one finds analysts debating about which object should

initiate a request and the resulting anomaly of passive entities modeled as active. For example, to get information from a book one might request that the book "read itself" and "send" the requested information in a message. To control an aircraft the pilot might "use his hands and feet to 'send messages' to the aircraft controls which in turn send messages to the aircraft control surfaces to modify themselves" [Davis 93]. Such decisions are about OOA mechanisms or design, not about the problem domain or requirements.

As mentioned in the previous section, where the decomposition into objects is driven only by use cases, the result is effectively a functional specification in object guise. The problems with such specifications are well understood [Parnas 72], in particular, being difficult to understand, change, or maintain.

A more serious complaint is that most OOA methods inadequately address our goal of developing a good SRS. Most OOA approaches in the literature provide only informal specification mechanisms, relying on refinement of the OO model in design and implementation to add detail and precision. There is no formal basis for determining if a specification is complete, consistent, or verifiable. Further, the approach does not directly address the issues of developing the SRS as a reference document. The focus is on problem analysis rather than specification. If the SRS is addressed at all, the assumption is that the principles applied to problem understanding and modeling are sufficient, when results are written down, to produce a good specification. Experience suggests otherwise. As we have discussed, there are inherently tradeoffs that must be made to develop a specification that meets the need of any particular project. Making effective tradeoffs requires a disciplined and thoughtful approach to the SRS itself, not just the problem. Thus, while OOA provides the means to address packaging issues, there is typically little methodological emphasis on issues like modifiability or organization of a specification for reference and review.

7.2 Use Cases

Usage scenarios or *use cases* have been widely adopted as a method for specifying required system behavior from the user's point of view. Use cases are sometimes deployed as the primary focus of elicitation and problem modeling [Schneider 98]. Use cases are also frequently employed as a first step in many object-oriented approaches (e.g., [Jacobsen 92], [Kruchten 99]). Despite their prevalence in object-oriented development, there is nothing intrinsically object-oriented about use cases and they are applied in other contexts. For these reasons, we will treat them separately.

Briefly, a use case describes a set of possible sequences of interactions between the system and a user seeking to accomplish a particular goal. Uses cases are intended capture a user-centric view of the required system behavior — i.e., how the system should respond to different user inputs to accomplish specific tasks like checking the balance on an account or adding an item to an on-line shopping cart.

While many approaches attempt to structure use cases by providing standard formats or templates (e.g., [Cockburn 00]), use cases are ultimately an informal, natural-language specification. A use-case template captures the user's (or *actor's*) interaction with the system as a sequence of natural-language statements that alternate between describing user inputs (e.g., the customer clicks the *checkout* button") and system responses ("the page displays the contents of the customers shopping cart").

Because use cases directly capture interaction with the system in terms of the user's problem domain (e.g., work tasks), they are usually easy for non-technical stakeholders to read, understand, review, and even assist in creating. While writing good use cases requires expertise, there is a relatively natural transition from a description of what a user wants the system to do, to a specification of how the system might support that task in a use case. Similarly, marketing or business goals for a system (e.g., what new things the system will allow users to do) are often straightforwardly represented as use cases [Lee 99].

While there is evidence that use cases can be an effective informal modeling technique, they lack many of the properties necessary to a technical requirements specification:

- *Unambiguous and consistent*: Use cases necessarily have all the limitations of any natural language specification. They are inherently ambiguous and open to inconsistent interpretation by stakeholders or developers.
- *Modifiable*: Individually, use cases are relatively easy to modify, particularly where standard templates are used. Collectively, where there are a large number of use cases, it can become very difficult to find or identify all of the use cases relating to a particular change.
- *Organized for reference and review*: Where the number of use cases becomes large, it also becomes difficult to find specific use cases or specific information. There is generally no organizing principle that accurately characterizes exactly where to put or find a given piece of information among the set of use cases. Similarly, it can be difficult for reviewers to find key information or assess basic properties like consistency.
- *Complete*: Since use cases represent specific paths through the system behavior, it is usually impossible or impractical to write a complete set of use cases. The problem is analogous to trying to write a complete set of test cases. While the level of abstraction is higher, in general, the number of possible scenarios is very large and there is no way to check if the set of use cases is complete, or to identify which ones might be missing.

There are also more important senses in which use cases are typically incomplete. Traditionally, use cases represent only users' interactions with the system. It follows that a specification written only in terms of use cases is an entirely functional specification. Other viewpoints as well as critical quality requirements are not addressed. Such an approach recapitulates the deficiencies of functional decomposition and discards decades of progress in software engineering. While there have been some efforts to modify use cases to represent quality requirements, (e.g., [Bass 03]) such approaches remain a work in progress.

These limitations suggest that use cases are more appropriate for informal business- or mission-oriented requirements capture. In many organizations there are two distinct audiences for the requirements: one audience that is versed in the organizational goals and problem domain and a second audience that is versed in technical goals and the solution domain. For businesses, the first audience typically includes customers, marketing, product management, and others on the business side of the organization. The second audience includes architects, coders, and others on the development side of the organization.

Because these two audiences tend to speak different languages and have different interests in the product, it is difficult to write any single specification that is suitable to both. In such cases, it often makes sense to create two distinct documents, one owned by the business side and a second

owned by the technical side. The goal in dividing the specification is to create a clear allocation of purpose, responsibility, and ownership.

The purpose of the business-oriented document is to capture the rationale for building the system. It includes the business case, solution approach, and the mapping between them. This document may be described as the, *Market Requirements Document (MRD), Business Requirements Document (BRD)* or, *Concept of Operation Document (ConOps)*. It should communicate the results of problem analysis and characterize the set of acceptable solutions to customers, managers, and others responsible for why the system is being developed. Because its purpose is to capture rationale, it is organized to "tell a story" [Fairley 97] rather than as a reference document.

The technical specifications are then captured in an SRS. By tracing requirements in the SRS to the BRD or similar document, one captures the origin and rationale for the technical requirements while maintaining the desirable properties of an SRS.

Use cases are a natural fit for the audience and purpose of a document like the ConOps or BRD. Use cases are written in terms of the problem domain and in a language that is accessible to those familiar with the problem domain. The format and organization is consistent with the objective that the document should "tell a story" and provides a vehicle for linking the system behavior to user tasks. While this comes at the expense of some redundancy in that the same requirements must be expressed in more than place, the benefits typically outweigh any issues in maintaining consistency.

7.3 Linking Requirements to Architecture

While a detailed discussion of software architecture is beyond the scope of this paper, one must have a clear understanding of the effect of architecture on important system qualities to understand the relationships between architectural design decisions and the requirements process.

We use the term *software architecture* to denote the structures of the system comprising a set of components, relations, and interfaces. For example, the *class structure* could refer to the set of classes in the system, the class interfaces, and the inheritance or instance relation. The *process structure* could refer to the organization of the system into processes or threads; interfaces are the inter-process operations (synchronization, communication), and the relations include exclusion and concurrency. By this definition, any software system can comprises of more than one architecture [Bass 03].

Architecture manifests the earliest set of design decisions. It is these decisions that enable or inhibit the system's quality attributes. These include essentially all of the system's developmental qualities (e.g., maintainability, reusability, etc.) and all of the system's behavioral qualities (e.g., performance, reliability, etc.) except functionality[3],

Inevitably, architectural design requires making tradeoffs among the system's quality attributes. For example, significantly increasing system security will tend to decrease performance and improving reliability will typically require longer development time.

3. Without going into detail, precisely the same functionality can be realized by any number of different architectural decompositions.

Since different stakeholders have different interests in system properties, the process of choosing among architectural design alternatives directly affects the extent to which the design will, or will not, satisfy their desires and goals. Since making good architectural design decisions requires making tradeoffs among the concerns of different stakeholders, the architect must understand the rationale for different quality requirements, as well as the relative priorities among stakeholder goals, and, ultimately, negotiate compromises. The architect must understand both the source and nature of the system's quality requirements.

The implication is that it is not sufficient to communicate black-box requirements; an effective process must also capture and communicate contextual information. This includes the purpose of different requirements, their relationships to organizational goals, and their importance to the system's diverse stakeholders.

Where an organization goes on to develop subsequent versions of the software or similar systems, the dependencies also extend downstream. The architectural design decisions embodied in the current system tend to influence subsequent business goals, requirements, and architectural structures. For example, how easy a system is to extend or modify the software in particular ways can significantly affect the ability to add specific features, address new customer needs, or target different markets.

These overlapping dependencies between developmental goals, requirements, and architectural design are captured in what Bass, et al., [Bass 03] calls the *architectural business cycle*. While our focus is on the role of requirements in that cycle, it expresses the key idea that there are important dependencies between the conceptually distinct activities of software development. Managing the implications of these dependencies requires explicit two-way communication between the business and technical parts of an organization. The activities and artifacts supporting this communication must be part of a disciplined process.

7.4 Elicitation Methods and Goal Modeling

Failing to understand what the stakeholders want leads to substantial rework [Boehm 88] or even rejection of a system. Because elicitation occurs at the beginning of development, errors in this step are the most expensive and difficult to correct later in the process. The importance of getting these early steps right has led to a wide range of efforts focused on understanding elicitation issues and supporting improved elicitation processes.

One significant result of these efforts has been a shift in the way researchers and practitioners view elicitation. While there were exceptions (e.g. [Gause 89]), the prevailing view in the past was that there existed some set of requirements characterizing the behavior of an ideal system. One could effectively elicit those requirements by asking a few key people, notably customers, and users, what the system should do.

For many of the reasons that we have discussed, this approach often proved ineffective. This reflects the fact that "what is wanted" is typically not well defined, fully understood, or even one thing. Rather, the perception of the problem, developmental goals, and requirements will vary from one stakeholder to the next, and even for a single stakeholder, over time. Any individual stakeholder's answers will yield a view that is neither complete nor precise. Views from multiple stakeholders tend to be inconsistent or conflicting.

The upshot is that the notion of an ideal system or set of requirements that can be "discovered" is a poor approximation of reality. Rather, there are many different perspectives on the

problem, partial views of solutions, and possible systems. The central challenge of elicitation is to obtain and reconcile these different perspectives to a single system definition that the stakeholders can live with.

Where, historically, this aspect of the requirements process received little attention, it has recently emerged as a distinct discipline in both practice and the literature. The understanding that elicitation must reconcile many different views from different kinds of stakeholders, and in different contexts, has stimulated research into the various facets of elicitation. This has, in turn, stimulated development of a number of elicitation methods targeted to different needs. An overview of the approaches is given in [Nuseibeh 00]; a more complete survey of different elicitation methods is given in [Lauesen 02].

7.4.1 Goal Modeling

An elicitation approach that integrates systematic modeling of objectives (e.g., business goals) with downstream requirements activities is *that of goal modeling* or *goal-oriented requirements*. A *goal* specifies some objective that the system should achieve [Lamsweerde 01]. The essential foci of goal-oriented requirements are:

1. To capture the stakeholder's objectives for the system in the problem context.
2. To systematically map those objectives to a detailed specification of the system requirements.

By beginning with goals, the approach seeks to capture each stakeholder's rationale for the system in the stakeholder's language and context. Thus, for example, business goals might be captured in terms of market opportunities and user needs in terms of ease of performing a work task. Expressing the system objectives using the stakeholder's perspective and language supports ease of understanding and elicitation. Integrating the different views of system goals provides an early opportunity for identifying and resolving conflicts [Robinson 89]. Subsequent refinement links rationale to specific system requirements. This supports two-way traceability and communication as goals or requirements evolve.

A relatively complete approach to requirements based on goals is the KAOS method by Lamsweerde *et al.* [Lamsweerde 09]. This work integrates goal-based elicitation with formal modeling and analysis. A formal language and tool support reasoning and the automated analysis of some completeness and consistency properties. Related publications include case studies of industrial experience (e.g., [Winter 01]). A good overview of goal-oriented requirements and set of references is given in [Lamsweerde 01].

7.5 "Agile" Methods

Much recent attention has been given to a set of development approaches that their authors characterize as "agile," for example, Extreme Programming [Beck 04], Scrum [Rising 02], or the Agile Unified Process [Ambler 02]. While there are differences among agile methods, they share a code-centered view of development – the view that the development effort should focus on the implementation rather than documentation (see the "agile manifesto"[4]).

The emphasis on code at the expense of documentation particularly pertains to the software requirements. Requirements documentation ranges from small amounts of informal documenta-

4. http://agilemanifesto.org

tion to using the code as the primary repository for all requirements and design decisions. This more extreme view is reflected in statements like: "The urge to write requirements documentation should be transformed into an urge to instead collaborate closely with your stakeholders and then create working software based on what they tell you."[5]

It should be clear that the software engineering philosophy behind these methods is at odds with what we have characterized as a "disciplined approach." To understand why this difference arises, it is necessary to examine the differences in methodological goals and the underlying assumptions the different approaches make about software development. By understanding the extent to which each approach's assumptions do or do not hold, the reader has a basis for choosing the approach best fitting a particular development situation.

Agile approaches seek to address the essential difficulties of *comprehension, communication,* and *control* by shortening the development cycle and bringing key stakeholders into the development loop. Many of the difficulties of traditional development processes (i.e., "waterfall" and its variations) arise from the temporal distance between project conceptualization and the delivery of any working software. In big projects, it may be months, or even years, between the time stakeholders begin describing their requirements and the time the developers can show them software that presumes to meet those requirements.

Because stakeholders typically do not know exactly what they want until they see it, this is often the point at which developers find out that what they have built is, in part or whole, not acceptable to the stakeholders. Because all of the work of design and implementation has been founded on incorrect requirements, fixing these errors is difficult and expensive. The result is a system that costs more than it should and delivers less than the stakeholders want.

Many of these problems can be avoided if it is possible to drastically shorten the development cycle. For agile methods, this cycle time is on the order of two to four weeks rather than months. Instead of eliciting all of the customer's requirements, the goal is to capture a small number of the most important ones (typically two or three). This small subset of requirements is then taken to code and validated with the customer. This cycle repeats until the customer is satisfied with the product. Little, if any, documentation is created or maintained. Rather, the code is the primary repository of the evolving set of requirements and design decisions.

With a short cycle time, the customer very quickly sees the expression of his requirements in the (partial) software. Errors and misunderstandings can be detected and corrected each cycle. Where errors occur, relatively little effort has been expended and the amount of rework may be limited to the length of the increment. Continuous communication between developers and the customer reduces the opportunity for misunderstanding. Because the developers are constantly integrating new requirements, requirements changes are addressed in the normal course of iterative development.

However, these benefits come at a substantial cost. Since only a small number of requirements can be considered at any time, there is no opportunity to understand the relationship of requirements to long-term goals, relationships between requirements, or the relationship between requirements and system structure:

- Because requirements are not gathered or considered in advance, it is not possible for the

5. http://www.agilemodeling.com/essays/agileRequirementsBestPractices.htm

designer to anticipate likely changes. There is constant rework as new requirements are added.

- Since only a very small subset of the requirements is examined at any one time, there is no mechanism to balance goals and make tradeoffs. Nor is there an opportunity to detect conflicting requirements before coding begins.
- Since the wide range of possible quality requirements that are whole-system properties (e.g., performance, safety, reliability, etc.) are not considered together, there is no opportunity to develop an architecture that balances such concerns. Similarly, constant restructuring (refactoring) makes it difficult to establish or maintain architectural properties.
- Constant interaction with the stakeholders is not just desirable, but essential. Without constant feedback validating the development, errors will accumulate over time, obviating the benefits of rapid increments.
- Because nothing is written down, progress depends on personnel who are intimately familiarity with the code. There is no mechanism to control the downstream effects of decisions on properties like maintainability or reusability.

Thus, realizing the benefits of agile methods depends on certain assumptions being true of the product, process, and people involved. It is a process that acts as if the development has neither a past nor a future, reacting only to immediate needs. Clearly there are many kinds of systems and development situations that are inconsistent with these assumptions, to name a few:

- Where there is limited availability or communication with stakeholders.
- Where stakeholders have conflicting views and requirements.
- Where there are critical behavioral and developmental properties that must be addressed by the architecture such as safety, reliability, or performance.
- Where requirements are relatively stable or predictable.
- Where there is a history of developing similar systems or the current system is a new version of a previous one.
- Where the development team is not co-located and frequent, high-bandwidth communication is not possible.
- Where the system is long lived and maintenance is a key concern - and so on.

In essence, agile approaches make an implicit assumption that the software requirements are relatively independent. It cannot be otherwise. If there are strong dependencies between requirements then the order in which requirements are addressed and design decisions are made significantly affects overall system properties including how easily the software can be changed to address subsequent requirements. These effects have been well understood for decades (e.g., [Parnas 76]). One obvious example is where requirements from different users conflict. Taking such requirements in arbitrary order (as opposed to considering them together) will result in an implementation that first meets one stakeholder's needs, then the other's, but never both.

It follows that there can be only limited circumstances in which the benefits of agile methods outweigh the costs and risks. The notion that most development efforts can abandon a disciplined

approach to requirements in favor of coding is not supportable. Unfortunately, many proponents of these methods do not make the underlying assumptions clear nor provide a balanced discussion of the limitations. Leaving this as an exercise for the reader may be good salesmanship but is poor software engineering. A somewhat more even-handed view can be found in [Boehm 02]. A more critical view that encompasses some of the issues of agile methods and XP is given in [Stephens 01].

7.6 Software Product-Lines

A view of development that spans multiple product cycles is that of software product-lines. Briefly, a software product line is a family of systems that share a significant number of common requirements, and are produced from a common set of reusable software assets. The reusable assets typically include a common software architecture, reusable, adaptable code modules, test cases, documentation, and so on.

Conventional software processes follow a "craftsman" production model – i.e., skilled individuals build each system by hand. Product-line development is more analogous to a manufacturing model where one builds a factory, then uses the factory to produce products. Software product lines are constructed by first creating a set of reusable assets, tools for deploying the assets (e.g., code generators), and a process for using the assets to produce members of the product line. Software systems are then created from the common assets.

Where applicable, software product-line approaches have been shown to significantly increase productivity (by as much as an order of magnitude), while decreasing cycle time and improving quality. Since code can be quickly created from reusable assets and validated with the customer, it provides the benefits of a rapid cycle time.

The approach, however, is applicable only where an organization is developing a number of reasonably similar systems. Refreshingly, the proponents of product-line approaches are careful not only to state the underlying assumptions (e.g., [Weiss 99]), but also to provide specific methods for assessing the costs and risks of applying a product-line approach to any particular application (also [Clements 01], [Pohl 05]).

The relevance of software product-lines to this discussion is that product line processes exemplify a disciplined approach to requirements that spans multiple software life cycles. Software product-lines work by amortizing the larger up-front development costs of the common asset base over the delivery of a number of similar software products. To create a reusable architecture and set of assets, the developers must understand not only the requirements for the next software system, but how those requirements are likely to vary over future instances of the product-line. In particular, which requirements should be the same across all members of the product-line (called *commonalities*) and which requirements are allowed to differ (called *variabilities*)?

This entails understanding both the current business objectives and how those objectives are likely to change over time. It also requires an understanding of the relationship of the requirements to the architecture, and how architectural design decisions will affect the future ability to build different versions of the product-line.

A variety of approaches to product-line requirements have been proposed and used. A significant difference from other requirements approaches has been a substantial body of work focusing on identifying and managing variabilities and the relationships between them (e.g.,

[Svahnber 05], [Pohl 05]). These works provide useful insight into disciplined approaches to managing requirements across multiple products and development cycles.

7.7 Practical Formal Methods

Like so many of the promising technologies in requirements, the application of formal methods is characterized by an essential dilemma. On one hand, formal specification techniques hold out the only real hope for producing specifications that are precise, unambiguous, and demonstrably complete or consistent. On the other, industrial practitioners widely view formal methods as impractical. Difficulty of use, inability to scale, readability, and cost are among the reasons cited. Thus, in spite of significant technical progress and a growing body of literature, the pace of adoption by industry has been extremely slow.

In spite of the technical and technology-transfer difficulties, increased formality is necessary. Only by placing behavioral specification on a mathematical basis will we be able to acquire sufficient intellectual control to develop complex systems with any assurance that they satisfy their intended purpose and provide necessary properties like safety. While it is not necessary to apply formal methods to all systems, or even all parts of critical systems, they are needed where it is necessary to establish correctness of the essential parts of critical systems (e.g., safety critical aspects). The solution is better formal methods - methods that are practical given the time, cost, and personnel constraints of industrial development.

Engineering models and the training to use them are *de rigueur* in every other discipline that builds large, complex, or safety–critical systems. Builders of a bridge or skyscraper who did not employ proven methods or mathematical models to predict reliability and safety would be held criminally negligent in the event of failure. It is only the relative youth of the software discipline that permits us to get away with less. But, we cannot expect great progress overnight. As Jackson [Jackson 94] notes, the field is sufficiently immature that "the prerequisites for a more mathematical approach are not in place." Further, many of those practicing our craft lack the background required of licensed engineers in other disciplines [Parnas 89]. Nonetheless, sufficient work has been done to show that more formal approaches are practical and effective in industry. The Naval Research Laboratory's (NRL) Software Cost Reduction (SCR) method and tools exemplify such an approach.

The Software Cost Reduction (SCR) Method: Where most of the techniques thus far discussed focus on problem analysis, the requirements work at the United States Naval Research Laboratory focused equally on issues of developing a good SRS [Heninger 80]. As part of an overall effort in validating software engineering methodologies the SCR project has developed rigorous approaches to requirements specification and documentation based on an underlying formal model.

The SCR approach uses formal, mathematically based specifications of acceptable system outputs to support development of a specification that is unambiguous, precise, and verifiable. It also provides techniques for checking a specification for a variety of completeness and consistency properties. The SCR approach introduced principles and techniques to support our SRS packaging goals including the principle of separation of concerns to aid readability and support ease of change. It includes the use of a standard structure for an SRS specification and the use of tabular specifications that improve readability and modifiability, and facilitate use of the specification for reference and review.

While other requirements approaches have stated similar objectives, the SCR project is unique in having applied software engineering principles to develop a standard SRS organization, a specification method, a review method [Parnas 85a], and notations consistent with those principles. The SCR project is also unique in making publicly available a complete, model SRS of a significant system [Alspaugh 92].

More recently, NRL has extended the SCR work to provide a suite of supporting tools. Since the approach is based on a formal model, the tools not only assist the developer in creating well-formed specification, the tools provide automated checking for the specification's completeness and consistency ([Heitmeyer 95a], [Heitmeyer 95b]). Likewise, the model can be used to support automated proofs of semantic properties like system safety properties [Heitmeyer 98] or fault tolerance [Jeffords 09] The work has also shown some of the promise of formal methods in supported automated test case generation [Gargantini 99] and even code generation [Rothamel 06].

While the SCR requirements approach is reasonably general, many of the specification techniques and models are targeted to real–time, embedded applications. More work needs to be done toward comparing the benefits of a practical formal methods to other types of systems.

8. Trends and Emerging Technology

There has been increasing agreement on the underlying problems in requirements as well as on the general characteristics of an effective requirements process. However, the overall trend has not been toward a common methodology, but toward a broadening of the concerns addressed and a proliferation of approaches.

These trends in requirements reflect more general trends in software engineering and software technology. As discussed in the section on processes, early life-cycle models tended to treat the conceptually distinct activities of software development, like requirements, design, and coding, as relatively independent phases. This reflected a desire to divide the development process into activities that addressed distinct concerns, with well-defined inputs and outputs.

With increasing application complexity and diversity of users, this paradigm has changed. More recent process models tend to reflect the view that the activities of the software life cycle are heavily interdependent and necessarily interleaved in time. Thus, for example, requirements activities may persist, if with diminishing effort, until the customer accepts the product. Where the software is developed in several versions, or part of a software product line, some requirements activities may continue across multiple delivery cycles ([Clements 01], [Faulk 01]).

At the same time, software has become increasingly ubiquitous. The types of applications along with the number and kinds of stakeholders have grown almost as fast as the size and complexity of the systems we build. One result has been an increasing diversity of development contexts and kinds of stakeholders.

Requirements research and practice have followed suit in broadening the scope of requirements activities and the diversity of methods. Thus, for example, we have seen new elicitation methods emerge to address different contexts and stakeholders. Likewise, requirements activities have been extended to encompass an organization's long-term goals and, in the case of software product lines, multiple developments or development cycles. We see these trends continuing in several areas of research and development:

Domain specificity: Requirements methods will provide improved support for understanding, specification, analysis, and usefulness by being tailored or created to address particular classes of problems.

Historically requirements approaches have been advanced as being equally useful to a wide variety of types of applications. For example, structured analysis methods based on conceptual models that were intended to be "universally applicable" (e.g., [Ross 77]); similar claims have been made for object-oriented approaches and notations like UML (e.g., [OMG 05]).

Such generality comes at the expense of ease of use and amount of work the analyst must do for any particular application. Where the underlying models have been tailored to a particular class of applications, the properties common to the class can be embedded in the model. The amount of work necessary to adapt the model to a specific instance of the class is relatively small. The more general the model, the more decisions that must be made, the more information that must be provided, and the more tailoring that must be done. This provides increased room for error and, since each analyst will approach the problem differently, makes solutions difficult to standardize. In particular, such generality precludes standardization of sufficiently rigorous models to support algorithmic analysis of properties like completeness and consistency.

Jackson [94] has expressed similar points. He points out that some of the characteristics separating real engineering disciplines from what is euphemistically described as "software engineering" are well-understood procedures, mathematical models, and standard designs specific to narrow classes of applications. Jackson points out the need for software methods based on the conceptual structures and mathematical models of behavior inherent in a given problem domain (e.g., publication, command and control, accounting, and so on). Such common underlying constructs can provide the engineer guidance in developing the specification for a particular system.

This trend is currently reflected in the proliferation of elicitation methods and models targeted to different development contexts. It is also evidenced in the trend toward tailoring the overall requirements processes [Young 06] to address the specific concerns of a project or organization. The trend toward better integration of requirements processes with business processes (e.g., [Middleton 05]) will also further the trend toward domain specificity to meet the needs of specific business areas.

Currently lacking are domain-specific approaches that encompass the artifacts, activities and roles comprising the entire requirements process. Some earlier work (e.g., [Prieto-Diaz 94], [Lam 97]) explored the potential of requirements reuse using domain-specific methods. Likewise, both product-line approaches and methods based on domain-specific modeling necessarily incorporate aspects of domain-specific requirements. For example, the use of the Embedded System Modeling Language (ESML) [Balasubramanian 07] on a family of embedded avionics applications [Karsai 02]. However, developing new requirements languages and semantics for specific domains remains a labor-intensive task. Progress in this area should see improved tool support (see the subsequent section on meta-engineering), new methods for modeling requirements in specific domains, and better guidance in adapting components to provide integrated processes.

Distributed Development: Another way in which the requirements problem has broadened (in a literal as well as figurative sense) is in the trend toward *distributed development*. We use the term "distributed development" to denote software projects where development teams and

activities are located in multiple geographic sites around the globe, particularly where sites are separated by time zones, cultures, and languages. While the early focus of globalization was on reduced cost, factors like increased access to talent and proximity to markets have continued to push the trend forward.

Distributed development has proven to have its own set of costs and risks, often requiring more effort and taking much longer than similar co-located projects [Mockus 01]. A key reason is the difficulty in achieving a common understanding of the requirements. In a cross-domain survey of industrial distributed developments, issues with misinterpreted, changing, and missing requirements ranked as the top three sources of error above all other development issues [Komi 05].

Experience suggests that distributed development is different from co-located projects (e.g., [Battin 01], Bradner 02]). These differences are manifestations of what Herbsleb characterizes as the key problem of distributed development, *coordination at a distance* [Herbsleb 07]. "Coordination," here, denotes the need to manage dependencies between people, tasks, and artifacts in a complex software development. In turn, difficulties in coordination are largely the result of difficulties in communicating effectively at a distance [Olson 2000], particularly where there are cultural, language, and organizational differences.

These differences suggest that new methods, models, and processes will be needed to manage requirements in distributed developments [Damian 07]. These will include new work in areas like cross-cultural requirements elicitation and communication. Likewise, new process models are needed for managing requirements elicitation, allocation, verification, and validation in a distributed project.

Personalization, Monitoring, and Adaptation: The trend toward broadening the scope of requirements engineering is evidenced in the areas of requirements personalization [Sutcliffe 06], requirements monitoring (e.g., [Fickas 95]), and real-time adaptation (e.g., [Robinson 05]). While these are three distinct areas of requirements research, they share a common concern for software *contextualization*: adapting software to a particular context such as user characteristics,

Contextualization extends the issues around changing requirements to a personal and real-time level. Personalized software is software that is produced to meet the requirements of small groups or even individuals. This can include software that is individually customized, software that the user can customize, or software that configures itself based on user preferences. Real-time adaptation is customization in response to changes over time. For example, software that changes behavior as the system moves through space (e.g., on a cell phone) or software that changes behavior depending on the time of day. Where the software itself does the adaptation, it must monitor parameters relating to the requirements (e.g., time of day or location) and change behavior accordingly.

While, historically, there have been many approaches to software customization and even personalization[6] these have not been systematically addressed as a type of requirements variation. Only recently have researchers begun to look at systematic approaches to understanding and managing contextual requirements.

6. The infamous Microsoft® "Clippy" being one.

Basically, contextualization embraces cases where requirements remain fluid even at run time. While we may continue to make tradeoffs between different stakeholders' requirements, we may also view the system as implementing more than one set of requirements at a time, switching between them depending on the context of use.

As more and more personal devices include increasingly powerful computing systems (or access to networks), the trend toward personalization and other forms of contextualization will grow. There is likewise a trend toward integrating the results of several requirements areas to address various dimension of the contextualization problem.

Personal Contextual - Requirements Engineering (PC-RE) [Sutcliff 06] addresses the issue that user goals tend to change with context. As the user moves through time and space, objectives and, hence, requirements change; PC-RE proposes a framework for relating changing goals, requirements, and modes of implementation.

Meta-Engineering: "Meta-engineering" refers to the engineering of engineering practices. All engineering disciplines include meta-engineering practices. An obvious example is that manufacturing necessarily includes processes for creating processes that will be used in a factory design to produce specific kinds of products.

Meta-engineering is an area in which software engineering excels [Faulk 10]. While creating "abstractions of abstractions" or designing "processes to design processes" may sound convoluted, it is precisely these kinds of capabilities that allow new methods, processes, and even tools to be created and introduced into practice at a pace commensurate with changing technology.

While not discussed in these terms, meta-engineering capabilities underlie some of the advances we have discussed in this paper. In particular, the ability to systematically create or adapt requirements processes to satisfy specific project constraints (i.e., the process requirements) is a meta-engineering activity. Likewise is the development of new methodologies like agile or product-line engineering.

Product-line engineering is a particularly instructive case since the product-line engineering process, itself, embeds a meta-engineering process. Whenever the domain engineers develop a set of product-line assets, it is also necessary to create a process for using those assets (common architecture, libraries of adaptable modules, etc.) to create any software product that is a member of the product line. Thus, any complete product-line process model includes a process for creating the application engineering process. Of course, the product-line process is itself a product of meta-engineering.

Improved meta-engineering capabilities will be necessary to much of the evolution of requirements practice. Facilitating the practice of defining new requirements processes for specific application domains requires providing systematic processes for producing new processes to satisfy specific developmental goals or constraints. Similar capabilities will be needed for fitting elicitation methods, modeling methods, and artifacts to specific needs.

The same argument can be made for tools. While we have not seen meta-engineering tools targeted specifically to requirements, meta-engineering tools exist in other disciplines. For example, there are already methods and "tool-building-tools" supporting product-line engineering [Kelly 08]. Such tools aim to create tools supporting application engineering based on a domain model. The output of the tool is a code generator that takes a specification of the requirements for member of the product line and generates the application code.

The potential for creating meta-engineering tools to support requirements modeling and analysis provides substantial opportunity for fruitful research.

9. Conclusions

Requirements are intrinsically hard to do well. Beyond the need for discipline, there are a host of essential difficulties that attend both the understanding of requirements and their specification. Further, many of the difficulties in requirements will not yield to technical solution alone. Addressing all of the essential difficulties requires the application of technical solutions in the context of human factors such as the ability to manage complexity or communicate to diverse audiences. A requirements approach that does not account for both technical and human concerns can have only limited success. For developers seeking new methods, the lesson is *caveat emptor*. If someone tells you his method makes requirements easy, keep a hand on your wallet.

Nevertheless, difficulty is not impossibility and the inability to achieve perfection is not an excuse for surrender. While all of the approaches discussed have significant weaknesses, they all contribute to the attempt to make requirements analysis and specification a controlled, systematic, and effective process. Though there is no easy path, experience confirms that the use of **any** careful and systematic approach is preferable to an *ad hoc* and chaotic one. Further good news is that, if the requirements are done well, chances are much improved that the rest of the development will also go well. Unfortunately, *ad hoc* approaches remain the norm in much of the software industry.

A final observation is that the benefits of good requirements come at a cost. Such a difficult and exacting task cannot be done properly by personnel with inadequate experience, training, or resources. Providing the time and the means to do the job right is the task of responsible management. The time to commit the best and brightest is before, not after, disaster occurs. The monumental failures of a host of ambitious developments bear witness to the folly of doing otherwise.

10. Further Reading

Those seeking more depth on requirements methodologies than this tutorial can provide have access to a number of good texts on software requirements. Berenbach, *et al.*, [Berenbach 09] focuses on practical approaches with depth in elicitation and quality attribute requirements. Weigers [Weigers 03] provides broad coverage with emphasis on the voice of the customer and requirements management. Young [Young 06] addresses effective practices and the role of a requirements plan. Middleton and Sutton [Middleton 05] provide a business-oriented approach driven by customer value.

Acknowledgements

The quality of this paper has been much improved thanks to thoughtful reviews by Merlin Dorfman and Richard Thayer. Paul Clements, Connie Heitmeyer, Jim Kirby, Bruce Labaw, Richard Morrison, and David Weiss provided helpful reviews of the first version.

REFERENCES

[Alford 79] M. Alford and J. Lawson, "Software Requirements Engineering Methodology (Development)," *RADC–TR–79–168*, U.S. Air Force Rome Air Development Center, June 1979.

[Alspaugh 92] T. Alspaugh, S. Faulk, K. Britton, R. Parker, D. Parnas, and J. Shore, "Software Requirements for the A–7E Aircraft," NRL/FR/5530–92–9194. Washington, D.C.: Naval Research Laboratory, 1992.

[Ambler 02] S. Ambler and R. Jeffries, *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*, Wiley, Boston, March 2002.

[Balasubramanian 07] K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, D.Schmidt, "A Platform-Independent Component Modeling Language for Distributed Real-Time and Embedded Systems," *Journal of Computer and System Sciences*, 73 (2), March 2007, 171-185.

[Bahill 05] A. Bahill and S. Henderson, "Requirements Development, Verification, and Validation Exhibited in Famous Failures, *Systems Engineering*, 8 (2), 2005, 1–14.

[Bass 03] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice* (Second Edition), Addison-Wesley, New York, 2003.

[Battin 01] R. Battin, Crocker, R., and Kreidler, J., "Leveraging Resources in Global Software Development," *IEEE Software*, 18 (2), 2001, 70-77.

[Beck 04] K. Beck and Andres, C., *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, Boston, Nov. 2004.

[Berenbach 09] B. Berenbach, D. Paulish, J. Kazmeier and A. Rudorfer, *Software & Systems Requirements Engineering in Practice*, McGraw Hill, New York, 2009.

[Boehm 81] B. Boehm, *Software Engineering Economics*, Prentice Hall, New Jersey, 1981.

[Boehm 88] B. Boehm and P. Papaccio, "Understanding and Controlling Software Costs," IEEE *Transactions on Software Engineering*, Oct. 1988.

[Boehm 94] B. Boehm, P. Bose, E. Horowitz, and M. Lee, "Software Requirements as Negotiated Win Conditions," in *Proceedings of the First International Conference on Requirements Engineering*, Colorado Springs, Colorado, Apr. 18-22, 1994, 74-83.

[Boehm 02] B. Boehm, and T. DeMarco, "The Agile Methods Fray," *IEEE Computer*, June 2002, 90-92.

[Bradner 02] E. Bradner, and G. Mark, "Why Distance Matters: Effects on Cooperation, Persuasion and Deception," *Proceedings of the 2002 ACM Conference on Computer Supported Cooperative Work,* New Orleans, 2002, 226 - 235.

[Brooks 95] F. Brooks, *The Mythical Man–Month: Essays on Software Engineering*, 2nd Edition, Addison–Wesley, 1995.

[Brooks 87] F. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer*, April 1987, 10-19.

[Clements 01] P. Clements and Northrop, L., *Software Product Lines: Practices and Patterns*, 3rd ed., Addison-Wesley Professional, Boston, Aug. 2001.

[Cockburn 00] A. Cockburn, *Writing Effective Use Cases*, Addison-Wesley, Reading, MA, 2000.

[Damian 07] D. Damian, "Stakeholders in Global Requirements Engineering: Lessons Learned from Practice," *IEEE Software*, 2007, 21-27.

[Davis 88] A. Davis, "A Taxonomy for the Early Stages of the Software Development Life Cycle," *Journal of Systems and Software*, Sep. 1988, 297-311.

[Davis 93] A. Davis, *Software Requirements (Revised): Objects, Functions, and States*, Prentice Hall, New Jersey, 1993.

[DeMarco 78] T. DeMarco, *Structured Analysis and System Specification*, Prentice Hall, New Jersey, 1978.

[Dorfman 90] M. Dorfman and R. Thayer (eds.), *Standards, Guidelines, and Examples on System and Software Requirements Engineering*, IEEE Computer Society Press, Los Alamitos, California, 1990.

[Fairley 97] R. Fairley and R. Thayer, "The Concept of Operations Document: The Bridge from Operational Requirements to Technical Specifications," in *Software Engineering*, R.H. Thayer and M. Dorfman (eds.), IEEE Computer Society Press, 1997.

[Faulk 92] S. Faulk, J. Brackett, P. Ward, and J. Kirby, Jr., "The Core Method for Real–Time Requirements," *IEEE Software*, Vol. 9, No. 5, Sep. 1992.

[Faulk 93] S. Faulk, L. Finneran, J. Kirby Jr., and A. Moini, *Consortium Requirements Engineering Guidebook*, Version 1.0, SPC–92060–CMC, Software Productivity Consortium, Herndon, Virginia, 1993.

[Faulk 01] S. Faulk, "Product-Line Requirements Specification (PRS): An Approach and Case Study," *Proceedings, Fifth IEEE International Symposium on Requirements Engineering*, Toronto, Canada, Aug. 27-31, 2001, 48-55.

[Faulk 10] S. Faulk and M. Young, "Sharing What We Know About Software Engineering," *Proceedings: Foundations of Software Engineering*, FOSER 10, Santa Fe, NM, and Nov., 2010.

[Fickas 95] S. Fickas and M. Feather, "Requirements Monitoring in Dynamic Environments," *Proceedings, Second IEEE International Symposium on Requirements Engineering*, York, England, March 1995, 140-150.

[GAO 79] U.S. General Accounting Office, *Contracting for Computer Software Development-Serious Problems Require Management Attention to Avoid Wasting Additional Millions*, Report FGMSD–80–4, Nov. 1979.

[GAO 92] U.S. General Accounting Office, *Mission Critical Systems: Defense Attempting to Address Major Software Challenges*, GAO/IMTEC–93–13, Dec. 1992.

[GAO 08] U.S. General Accounting Office, *Significant Problems of Critical Automation Program Contribute to Risks Facing 2010 Census*, GAO-08-550T, March 2008.

[GAO 10] U.S. General Accounting Office, *Defense Acquisitions: Assessments of Selected Weapon Programs*, GAO-10-388SP, March 2010.

[Gargantini 99] A. Gargantini and C. Heitmeyer, "Using Model Checking to Generate Tests from Requirements Specifications," *Proceedings, Joint 7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (ESEC/FSE99), Toulouse, FR, Sept. 6-10, 1999.

[Gause 89] D. Gause and G. Weinberg, *Exploring Requirements: Quality Before Design*, Dorset House, 1989.

[Game & Sarsen 79] C. Game and T. Sarsen, *Structured Systems Analysis*, Prentice Hall, New Jersey, 1979.

[Heitmeyer 95a] C. Heitmeyer, B. Labaw, and D. Ki skis, "Consistency Checking of SCR–Style Requirements Specifications," in *Proceedings, IEEE International Symposium on Requirements Engineering*, March 1995.

[Heitmeyer 95b] C. Heitmeyer, R. Jeffords, and B. Labaw, "Tools for Analyzing SCR–Style Requirements Specifications: A Formal Foundation," NRL Technical Report NRL-7499, U.S. Naval Research Laboratory, Washington, DC, 1995.

[Heitmeyer 98] C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj, "Using Abstraction and Model Checking to Detect Safety Violations in Requirements Specifications," *IEEE Transactions on Software Engineering*, 24, (11), November 1998.

[Heninger 80] K. Heninger, Specifying Software Requirements for Complex Systems: New Techniques and Their Application, *IEEE Transactions on Software Engineering*, 6 (1), Jan. 1980.

[Herbsleb 07] J. Herbsleb, "Global Software Engineering: The Future of Socio-Technical Coordination," *International Conference on Software Engineering 2007 Future of Software Engineering*, IEEE Computer Society, 2007, 188-198.

[Hester 81] S. Hester, D. Parnas, and D. Utter, "Using Documentation as a Software Design Medium," *Bell System Technical Journal*, 60 (8), Oct. 1981, 1941-1977.

[Jackson 94] M. Jackson., "Problems, Methods, and Specialization," *IEEE Software*, Nov. 1994, 57-62.

[Jacobsen 92] I. Jacobson, Christerson, M., Jonsson, P., and Övergaard, G., *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, Reading, MA,1992.

[Jeffords 09] R. Jeffords, C. Heitmeyer, M. Archer, and E. Leonard, A Formal Method for Developing Provably Correct Fault-Tolerant Systems Using Partial Refinement and Composition, *Proceedings, Formal Methods, Second World Congress (FM 2009),* Eindhoven, The Netherlands, November 2-6, 2009, 173-189.

[Karsai 02] G. Karsai, S. Neema, B. Abbott, and D. Sharp, "A Modeling Language and its Supporting Tools for Avionics Systems," *Proceedings of the 21st Digital Avionics Systems Conference*, Oct. 2002, 6A3 1-13.

[Kelly 08] S. Kelly and Juha-Pekka Tolvanen, *Domain-Specific Modeling: Enabling Full Code Generation*, Wiley-IEEE Computer Society Press, March 2008.

[Komi 05] S. Komi-Sirvio and M. Tihinen, "Lessons Learned by Participants of Distributed Software Development," *Knowledge and Process Management*, 2005, 108-122.

[Kruchten 99] P. Kruchten, *The Rational Unified Process: An Introduction*, Addison-Wesley, Reading, MA,1999.

[Lam 97] W. Lam, "Achieving Requirements Reuse: A Domain Specific Approach from Avionics," *Journal of Systems and Software*, 38 (3), Sept. 1997, 197-209.

[Lamsweerde 98] A. van Lamsweerde, R. Darimont, and E. Letier, "Managing Conflicts in Goal-Driven Requirements Engineering" in *IEEE Transactions on Software Engineering*, Nov. 1998.

[Lamsweerde 09] A. van Lamsweerde, *Requirements Engineering: From System Goals to UML Models to Software Specifications*, Wiley, March 2009.

[Lauesen 02] S. Lauesen, "*Software Requirements: Styles and Techniques*," Addison-Wesley Professional, London, 2002.

[Lee 99] J. Lee and N. Xue, "Analyzing User Requirements by Use Cases: A Goal-Driven Approach," *IEEE Software*, 16 (4): July/Aug. 1999, 92-101.

[Lutz 93] R. Lutz, "Analyzing Software Requirements Errors in Safety–Critical Embedded Systems," *Proceedings, IEEE International Symposium on Requirements Engineering*, Jan. 4-6, 1993, 126-133.

[Middleton 05] P. Middleton and J. Sutton, *Lean Software Strategies: Proven Techniques for Managers and Developers*, Productivity Press, NY, 2005.

[Mockus 01] A. Mockus and J. Herbsleb, "Challenges of Global Software Development," *Proceedings of the Seventh International Software Metrics Symposium*, 2001.

[Nuseibeh 00] B. Nuseibeh and S. Easterbrook, "Requirements Engineering: A Roadmap," in *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, New York, NY, 2000, ACM, 35–46.

[NASA 05] S. Cavanaugh, A. Wilhite, "Systems Engineering Cost/Risk Analysis Capability Roadmap Progress Review," Apr. 6, 2005.

[Olson 2000] G. Olson, and Olson, J., "Distance Matters," *Human-Computer Interaction, 15* (2), 2000, 139-178.

[OMG 05] Object Management Group, "Introduction to OMG's Unified Modeling Language (UML)," [http://www.omg.org/gettingstarted/what_is_uml.htm], 2005.

[Parnas 72] D. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM*, 15 (12), December 1972, 1053-1058.

[Parnas 76] D. Parnas, "On the Design and Development of Program Families," *IEEE Transactions on Software Engineering*, 2 (l) March 1976, 1-9.

[Parnas 85a] D. Parnas, and D. Weiss, Active Design Reviews: Principles and Practices, in *Proceedings of the Eighth International Conference on Software Engineering*, London, England, Aug. 1985.

[Parnas 86] D. Parnas, and P. Clements, "A Rational Design Process: How and Why to Fake It," *IEEE Transactions on Software Engineering*, 12 (2), Feb. 1986. 251-257.

[Parnas 89] D. Parnas, *Education for Computing Professionals*, Technical Report 89-247, Department of Computing and Information Science, Queens University, Kingston, Ontario, 1989.

[Parnas 91] D., Parnas and J. Madey, *Functional Documentation for Computer Systems Engineering (Version 2)*, CRL Report No. 237, McMaster University, Hamilton, Ontario, Canada, Sept. 1991.

[Pohl 05] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*, 1st ed., Springer, Sept. 2005.

[Prieto-Diaz 97] R. Prieto-Diaz, M. Lubars, M. Carrion, "DSSR: Support for Domain Specific Software Requirements," U.S. Army Communications-Electronics Command, April 1994.

[Rising 02] L. Rising and Janoff, N. S., "The Scrum Software Development Process for Small Teams," *Software, IEEE*, 17 (4), pp. 26-32, Aug. 2002.

[Robinson 89] W. Robinson, "Integrating Multiple Specifications Using Domain Goals," *Proceedings, 5th International Workshop on Software Specification and Design*, IEEE, 1989, 219-225.

[Robinson 05] W. Robinson, "Implementing Rule-Based Monitors within a Framework for Continuous Requirements Monitoring," in *Hawaii International Conference on System Sciences (HICSS'05)*, Big Island, Hawaii, USA, 2005.

[Ross 77] D. Ross and K. Schuman Jr., "Structured Analysis for Requirements Definitions," IEEE *Transactions on Software Engineering*, 3 (1), Jan. 1977, 6-15.

[Rothamel 06] T. Rothamel, C. Heitmeyer, Y. Liu, and E. Leonard, "Generating Optimized Code from SCR Specifications," in *Proceedings, ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2006),* Ottawa, Canada, June 14-16, 2006.

[Schneider 98] G. Schneider, and J. P. Winters, *Applying Use Cases: A Practical Guide*. Reading, MA: Addison-Wesley, Reading, MA, 1998.

[SEI 06] *CMMI for Development, Version 1.2,* CMMI-DEV, Carnegie Mellon University Software Engineering Institute, Aug., 2006.

[Stephens 01] M. Stephens and D. Rosenberg, *Extreme Programming Refactored: The Case against XP*. APRESS, Sep. 2003.

[Sutcliffe 06] A. Sutcliffe, S. Fickas, M. Sohlberg, *Journal of Requirements Engineering*, 11 (3), Jun. 2006.

[Svahnberg 05] M. Svahnberg, J. van Gurp, and J. Bosch, "A Taxonomy of Variability Realization Techniques," *Software: Practice and Experience*, 35 (8), pp. 705-754, 2005.

[Svoboda 90] C. Svoboda, "Structured Analysis," in *Tutorial: System and Software Requirements Engineering*, R. Thayer and M. Dorfman (eds.), IEEE Computer Society Press, Los Alamitos, California, 1990, 218-237.

[Thayer 90] R. Thayer and M. Dorfman (eds.), *Tutorial: System and Software Requirements Engineering,* IEEE Computer Society Press, Los Alamitos, California, 1990.

[Weigers 03] K. Weigers, *Software Requirements*, Microsoft Press, Redmond, WA, 2003.

[Weiss 99] D. Weiss and C. T. R. Lai, *Software Product-Line Engineering: A Family-Based Software Development Process*, Addison-Wesley, Reading MA, 1999.

[Winter 01] V. Winter, R. Berg, and J. Ringland, "Bay Area Rapid Transit District Advance Automated Train Control System Case Study Description," in *High Integrity Software*, Kluwer Academic Publishers, Norwell, MA, 2001.

[Young 06] R. Young, *Project Requirements: A Guide to Best Practices*, Management Concepts, 2006.

[Yourdon 89] E. Yourdon, *Modern Structured Analysis*, Yourdon Press/Prentice Hall, Upper Saddle River, NJ, 1989.

Chapter 1.2

Essentials of Software Requirements Engineering

Richard Hall Thayer and Merlin Dorfman

This is the first chapter of a textbook to aid individual software engineers in a greater understanding of the IEEE SWEBOK [2013] and a guide book to aid software engineers in passing the IEEE CSDP/CSDA certification exams.

Missing, incomplete, or inaccurate software requirements are a major issue in software engineering. Better quality in both the software development process and the software product can be obtained if our methods and tools for gathering, modeling, and analyzing user requirements are more effective, robust, and codified in practice. Therefore, software requirements engineering (SRE) has emerged as an "engineering" approach to what used to be called "requirements analysis and specification."

This increased awareness of software requirements engineering is shown by an increase in the number of conferences, workshops, books and journals devoted exclusively to requirements engineering.

Chapter 1 covers the CSDP exam specifications for the software requirements engineering module [Software Exam Specification, Version 2, 18 March 2009]:

1. Software requirements fundamentals (definition of a software requirement; product and process requirements; functional and non-functional requirements; emergent properties; quantifiable requirements; system requirements and software requirements)
2. Requirements process (process models; process actors; process support and management; process quality and improvement)
3. Requirements elicitation (requirements sources; elicitation techniques)
4. Requirements analysis (requirements classification; conceptual modeling; architectural design and requirements; requirements negotiation; formal analysis)
5. Requirements specification (the system definition document; the system requirements specification; the software requirements specification)
6. Practical considerations (iterative process; change management; requirements attributes; requirements tracing; measuring requirements; software requirements tools)

1.1 Software Requirements Fundamentals

1.1.1 Definition of software requirements. Software requirements are defined as:

- A software capability required by a user to solve a problem or achieve an objective.
- A software capability that must be met or possessed by a system or system component to satisfy a contract, specification, standard, or other formally imposed document [IEEE Std 610.12-1990].

- A *problem definition*, which is, determining the needs and constraints of the software system by analyzing the system requirements that have been allocated to software [Thayer 2004].
- An externally observable characteristic of a desired system [Davis 2005].

1.1.2 Product and process requirements. Possible groupings of requirements involve *product* versus *process* requirements. *Product requirements* apply to the product or services to be developed, and include what the system does, data or command inputs or outputs to the system, speed and memory required by the system, *quality metrics* (e.g., reliability, maintainability, and security), and limits on the design's freedom such as interfaces (keyboard or mouse), language, and accuracy of computations.

Process requirements apply to the activities associated with enabling the creation of a product or service. Some examples of tasks to be performed are processes that:

- Analyze a manual effort
- Develop a product
- Operate a system
- Comply with a product requirement
- Comply with constraints such as compliance with laws, standards, regulations, and rules

Process requirements are normally defined in a contractual statement of work (SOW) or program *plan, NOT in the software requirements specification, which, in standard US practice, is limited* to product requirements.

1.1.3 Functional and non-functional requirements. There are five general types of requirements [Thayer 2004]: *functional, performance, external interface, design constraints,* and *quality attributes*. A sixth type, "Other," may be added if a system has requirements that do not fit neatly into one of the first five.

1.1.3.1 Functional requirements. A functional requirement is a system/software requirement that specifies a function that a system, software system or component must be capable of performing. Functional requirements define system behavior—the fundamental process of transformation that the system's software and hardware components must perform on inputs to produce outputs. Functional requirements should define the fundamental actions that must take place in the software when accepting and processing the inputs and when processing and generating the outputs.

1.1.3.2 Non-functional requirements. Non-functional requirements are just that—**those** requirements that do not directly affect the functionality of the system.

1.1.3.2.1 Performance requirements. *Performance requirements* specify a performance characteristic that a system or system component must possess – typically speed, volume, or accuracy. For example, static numerical requirements may include:

- The number of customer contacts to be supported
- The number of simultaneous users to be supported
- The number of files and records to be handled

- The size of tables and files that must be handled
- Dynamic numerical requirements may include, for example, the number of transactions and tasks and the amount of data to be processed within certain times for both normal and maximum workload conditions

All these requirements should be in measurable terms—for example, "95% of the transactions shall be processed in less than one second" rather than "the operator shall not have to wait for the transaction to complete." All requirements must be validated (tested) as part of system development, and unmeasureable requirements may be difficult or impossible to test.

Numerical requirements should also specify the logical requirements for any information to be placed into a database. See section on "Other requirements" (1.1.3.2.5) for additional information on logical requirements.

1.1.3.2.2 External interface requirements. An external interface requirement is a system/software requirement that specifies a hardware, software, or database element with which a system, software component or human must interface or that sets forth constraints on formats, timing, or other factors that such an interface causes. The interface requirement should consider:

- Item name
- Numerical requirement
- Description of purpose
- Input source or output destination
- Valid range, accuracy, or tolerance
- Units of measure
- Timing

1.1.3.2.3 Design constraints. A *design constraint* is any requirement that affects or constrains the design of a software system or software system component. Table 1.1 lists several design constraints. Performance requirements and quality attributes may also be considered to be special cases of design constraints.

Table 1.1: Sample design constraints

Size	Physical size of the product
Programming language	Programmed in the Ada language
Power consumption	Maximum electric power that the product may use
Human computer interface	Requires menus for system interface
Weight	Physical weight of the product
Computer resource utilization	Uses no more than a specified fraction of CPU cycles, communications bandwidth, etc.
Incorporated software	Must use a specified data base management system

1.1.3.2.4 Quality attributes. A quality attribute specifies the degree of an attribute (a number) that affects the quality the software must possess. Some quality attributes are included in Table 1.2.

Other quality factors are listed in IEEE Standard 1061-1998, IEEE Standard for a Software Quality Metrics Methodology, IEEE Standard 982.1-1988, IEEE Standard for Dictionary of Measures to Produce Reliable Software, and IEEE Standard 1044-1993, IEEE Standard Classification for Software Anomalies.

It is often very difficult to provide proper requirements metrics for software quality that can prove performance to the requirements specifications.

The developer and user then need to agree on what will constitute a valid software requirement for the particular quality attribute.

1.1.3.2.5 Other requirements. These are requirements that do not fit within the basic types of requirements and thus fall under the "miscellaneous requirements" category:

- Data definition and database requirements
- Installation and acceptance requirements for the delivered software product at the operation and maintenance site(s)
- User documentation requirements
- User operation and execution requirements

Table 1.2: Examples of quality requirements

Maintainability — The average effort required to locate and fix a software failure.
Reliability — The probability that the software will perform its logical operation in the specified environment without failure.
Safety —The probability that a system does not lead to a state in which human life or environment are endangered.
Security — The protection of computer hardware and software from accidental or malicious access, use, modification, destruction, or disclosure; the probability that the system can be made secure for a predetermined amount of time.
Survivability — The probability that the system will continue to perform or support critical functions when a portion of the system is inoperable.
User friendliness — The degree of ease of use or learning of a system.

1.1.4 Emergent properties. *Emergent properties* (i.e., requirements) of software are requirements that cannot be addressed by a single component, but that depend for their satisfaction on how all the software components interoperate. Emergent properties are crucially dependent on the system architecture.

1.1.5 Quantifiable requirements. Software requirements should be stated as clearly and as unambiguously as possible, and, where appropriate, quantitatively. It is important to avoid vague and unverifiable requirements that depend for their interpretation on subjective judgment ("the software shall be reliable;" "the software shall be user-friendly").

These non-quantifiable requirements can be stated in user-defined documents such as a Concept of Operations (ConOps) document.

1.1.6 System requirements and software requirements. System requirements are the requirements for the system as a whole. In a system containing software components, software requirements are derived from system requirements.

The literature on system requirements sometimes calls system requirements "user requirements." We can define "user requirements" in a restricted way as the requirements provided by the system's customers or end-users. System requirements, by contrast, encompass user requirements, requirements of other stakeholders (such as regulatory authorities), and requirements without an identifiable human source ("derived requirements").

1.2 Requirements Process

This section introduces the software requirements process, orienting the remaining five subareas and showing how the requirements process dovetails with the overall software engineering process [SWEBOK 2004].

1.2.1 Process models. The objective of this topic is to provide an understanding that the requirements process:

- Is not a discrete front-end activity of the software life cycle, but rather a process initiated at the beginning of a project and continuing to be refined throughout the life cycle.
- Identifies software requirements as configuration items, and manages them using the same software configuration management practices as other products of the software life-cycle processes.
- Needs to be adapted to the organization and project context.
- In particular is concerned with how the activities of elicitation, analysis, specification, and validation are configured for different types of projects and constraints.

1.2.2 Process actors. This topic introduces the roles of the people who participate in the requirements process. This process is fundamentally interdisciplinary, and the requirements specialist (i.e., the "System Engineer") needs to mediate between the domain of the stakeholder and that of software engineering. There are often many people involved besides the requirements specialist, each of whom has a stake in the software. The stakeholders will vary across projects, but will always include users/operators and customers (who need not be the same).

Typical examples of software stakeholders include (but are not restricted to):

- ***Users*** — This group comprises those who will operate the software. It is often a heterogeneous group comprising people with different roles and requirements.
- ***Customers*** — This group comprises those who have commissioned the software or who represent the software's target market.
- ***Market analysts*** — A mass-market product will often not have a real customer, so marketing people are needed to establish the market needs and to act as proxy customer.
- ***Regulators*** — Many application domains such as medical, banking, and public transport are regulated. Software in these domains must comply with the requirements of the regulatory authorities.

- ***Software engineers*** — These individuals have a legitimate interest in profiting from developing the software by, for example, reusing components in other products. If, in this scenario, a customer of a particular product has specific requirements that compromise the potential for component reuse, the software engineers must carefully weigh their own stake against that of the customer.

It will not be possible to perfectly satisfy the requirements of every stakeholder, and it is the system engineer's job to negotiate trade-offs that are both acceptable to the principal stakeholders and within budgetary, technical, regulatory, and other constraints. A prerequisite for successful negotiations is that all the stakeholders be identified. the nature of their "stake" analyzed, and their requirements elicited.

1.2.3 Process support and management. This topic introduces the project management resources required and consumed by the requirements process. Its principal purpose is to make the link between the identified process activities and the issues of cost, human resources, training, and tools.

1.2.4 Process quality and improvement. This topic is concerned with the assessment of the quality and improvement of the requirements process. Its purpose is to emphasize the key role the requirements process plays in terms of the cost and timeliness of a software product, and of the customer's satisfaction with it. It will help to orient the requirements process with quality standards and process improvement models for software and systems. Of particular interest are issues of software quality attributes and measurement, and software process definition. This topic covers:

- Requirements process measures and benchmarking
- Improvement planning and implementation

1.3 Requirements Elicitation

Software requirements elicitation is the process through which the customer (buyer and/or user) and developer (contractor) of a software system discover, reveal, articulate, and understand the customers' requirements.

1.3.1 Requirements sources. Software requirements information can be obtained from many sources:

- *System requirements specifications* prepared by a system engineering function describe the totality of the requirements for the entire system. Buried in these systems requirements are those requirements that can best be satisfied, completely or partially, through software.
- *Procurement specifications and statements of work (SOWs)* are documents produced by the acquisition agency in preparing for a contract to develop and deliver a software system. By necessity, these documents contain top-level software requirements.
- *Marketing (product) requirements documents* are sales documents in which marketing has described a possible future product and obtained customer concurrence, sometimes without the knowledge and agreement of engineering or other software developers.
- *Customer-prepared needs documents* are requirements-type documents prepared by a system user to establish the need for a new system or changes to an existing system.

These documents typically describe the system's operational needs rather than its technical requirements.

- *Descriptions* of how the system is intended to operate in service, e.g., scenarios, use cases, and user "stories." These are typically developed by, or in conjunction with, intended users.
- A *concept of operations (ConOps) document* [IEEE Std 1362-1998] is a rather formal method of documenting a system's operational needs. This document is prepared by the system's potential user(s) and spells out needs and expectations.
- *Observations and measurements of the current system* by the user, developer, or acquirer (a third party contracted to prepare the needs documents).
- Interviews with the customers and users to elicit system requirements.
- *Current system documentation* that contains the current system's processes and products.
- *Feasibility studies* performed to justify the development of a new software system.
- *Models and prototypes* built to demonstrate parts of the finished system.

1.3.2 Elicitation techniques. [Cleland-Huang 2004], [Christel & Kang 1992]. A few of the techniques that can be used to identify the software requirements are:

- ***Asking*** — *Identify the appropriate person, such as the acquirer or user of the software, and ask what the requirements are.*
- ***Discussing and formulating*** — *Discuss needs with the customer and/or system acquirer and jointly formulate a common understanding of the requirements.*
- ***Observing*** — *Observe the behavior of users of an existing system (whether manual or automated).*
- ***Facilitated meetings*** — *A meeting between interested stakeholders to arrive at a consensus, e.g., JAD (Joint Application Development) [Wood & Silver 1995].*
- ***Negotiating with respect to a standard set*** — *Beginning with an existing set of requirements or features, negotiate with users which of those features will be included, excluded, or modified.*
- ***Prototyping*** — *Provides a context in which the users better understand what they need.*

1.4 Requirements Analysis

This topic is concerned with the process of analyzing requirements to:

- Detect and resolve conflicts between requirements
- Discover the bounds of the software and how it must interact with its environment
- Elaborate system requirements to derive software requirements.

The traditional view of requirements analysis has been that it should be reduced to conceptual modeling using one of a number of analysis methods such as structured analysis, object-oriented analysis, and use cases. While conceptual modeling is important, we include the classification of requirements to help inform trade-offs between requirements (requirements classifica-

tion) and the process of establishing these trade-offs (requirements negotiation).

Care must be taken to describe requirements precisely enough to enable the requirements to be validated, their implementation to be verified, and their costs to be estimated.

1.4.1 Requirements classification. Requirements can be classified on a number of dimensions:

- Whether the requirement is *functional or non-functional*
- Whether the requirement is an *original requirement* (from the customer/user) or is a *derived requirement* (from the developer)
- Whether the requirement is a *product* (from the requirements specifications) or a *process* (documented in the statement of work [SOW]) requirement
- The degree of priority, e.g., *mandatory, desirable, optional*
- The range of the requirements
- Whether the requirements might be verifiably stable.

Other classifications may be appropriate, depending upon the organization's normal practice and the application itself.

1.4.2 Conceptual modeling. The development of models of a real-world problem is a key to software requirements analysis. Their purpose is to aid in understanding the problem, rather than to initiate design of the solution. Hence, conceptual models comprise models of entities from the problem domain configured to reflect their real-world relationships and dependencies.

Several kinds of models can be developed. These include data and control flows, state models, event traces, user interactions, object models, data models, and many others. The factors that influence the choice of model include:

- *The nature of the problem.* Some types of software demand that certain aspects be analyzed particularly rigorously, because these aspects are known to be difficult or error-prone. For example, control flow and state models are likely to be more important for real-time software than for management information software, while it would usually be the opposite for data models.
- *The expertise of the software engineer.* It is often more productive to adopt a modeling notation or method with which the software engineer has experience.
- *The process requirements of the customer.* Customers may impose their favored notation or method, or prohibit any with which they are unfamiliar. This factor can conflict with the previous factor.
- *The availability of methods and tools.* Notations or methods that are poorly supported by training and tools may not achieve widespread acceptance even if they are suited to particular types of problems.

Note that, in almost all cases, it is useful to start by building a model of the software context. The software context provides a connection between the intended software and its external environment. This is crucial to understanding the software's context in its operational environment and to identifying its interfaces with the environment.

The issue of modeling is tightly coupled with that of methods. For practical purposes, a method is a notation (or set of notations) supported by a process that guides the application of the notations. There is little empirical evidence to support claims for the general superiority of one notation over another (though some are known to be more useful in a particular problem space than in others). However, the widespread acceptance of a particular method or notation can lead to beneficial industry-wide pooling of skills and knowledge.

Some examples of current software requirements models (or methods) are:

- Structured analysis (a.k.a. the Yourdon method)
- Object-Oriented analysis
- Use cases

- *Formal modeling* using notations based on discrete mathematics, and which are traceable to logical reasoning, have made an impact in some specialized domains. These tend to be difficult and time-consuming but may be imposed by customers or standards, or may offer compelling advantages to the analysis of certain critical functions or components.

1.4.3 Architectural design and requirements allocation. At some point, the architecture of the solution must be derived. Architectural design is the point at which the requirements process overlaps with software or systems design and illustrates how impossible it is to cleanly decouple the two tasks. In many cases, the software engineer acts as software architect because the process of analyzing and elaborating the requirements demands the identification of components that will be responsible for satisfying the requirements. This is requirements allocation—the assignment to components of the responsibility for satisfying requirements.

Architectural design is closely identified with conceptual modeling. Because the mapping from real-world domain entities to software components is not always obvious, architectural design is identified as a separate topic. The requirements of notations and methods are broadly the same for both conceptual modeling and architectural design.

1.4.4 Requirements negotiation. *Requirements negotiation*, a.k.a. "conflict resolution," concerns resolving problems with requirements where conflicts occur between two stakeholders requiring mutually incompatible features, between requirements and resources, or between functional and non-functional requirements. In most cases, it is unwise for the software engineer to make a unilateral decision, and so it becomes necessary to consult with the stakeholder(s) to reach a consensus on an appropriate trade-off. It is often important for contractual reasons that such decisions be traceable back to the customer. We have classified this as a software requirements analysis topic because problems emerge as the result of analysis. However, a strong case can also be made for considering this to be a requirements validation topic.

1.4.5 Formal analysis. In computer science and software engineering, *formal methods* are a particular kind of mathematically-based technique for the specification, development and verification of software and hardware systems. The use of formal methods for software and hardware design is motivated by the expectation that, as in other engineering disciplines, performing appropriate mathematical analysis can contribute to the reliability and robustness of a design. However, the high cost of using formal methods means that they are usually only used in the development of high-integrity systems, where safety or security is of utmost importance.

Formal methods are best described as the application of a fairly broad variety of theoretical computer science fundamentals, in particular logic calculi, formal languages, automata theory, and program semantics, but also type systems and algebraic data types to problems in software and hardware specification and verification.

1.5 Requirements Specification

For most engineering professions, the term "specification" refers to the assignment of numerical values or limits to a product's design goals. Typical physical systems have a relatively small number of such values. A typical software system has a large number of requirements, and the emphasis is shared between performing the numerical quantification and managing the complexity of interaction among the large number of requirements. So, in software engineering jargon, "software requirements specification" typically refers to the production of a document, or its electronic equivalent, that can be systematically reviewed, evaluated, and approved.

For complex systems, particularly those involving substantial non-software components, as many as three different types of documents are produced. However, in small, software-driven documentations, only a software requirements specification is required. In this paragraph we will discuss: [SWEBOK 2004].

- System definition document
- System requirements specification
- Software requirements specification

1.5.1 System definition document. The *software acquisition manager* or *product manager* is responsible for assisting the user in developing the user document (sometimes called the concept of operation [ConOps] document or "needs document") supporting the user during acceptance testing, and, finally, delivering the system to the user.

The document lists the system requirements along with background information about the overall objectives for the system, its target environment and a statement of the constraints, assumptions, and non-functional requirements. It may include conceptual models designed to illustrate the system context, usage scenarios and the principal domain entities, as well as data, information, and workflows.

In turn, the *software project manager* or *system engineer* is responsible for developing the software requirements specification from the users' needs documents, delivering the system to the acquirer within budget, and meeting the acquirer's expectations and requirements.

1.5.2 System requirements specification. A *system requirements specification* is a document that sets forth the requirements for a system or system segment. Typically included are functional requirements, performance requirements, interface requirements, quality attributes, design constraints, and development standards. The software requirements are derived from the system requirements specification.

A *software requirement* is a software capability that must be met or possessed by a system component to satisfy a contract, specification, standard, or other formally imposed document.

1.5.3 Software requirements specification. This document is based on a model in which the result of the software requirements specification process is an unambiguous and complete specification document. The purpose of this document is so that [Thayer & Dorfman 1998]:

- Software customers can accurately describe what they wish to obtain
- Software suppliers can understand exactly what the customer wants
- Individuals can accomplish the following goals:
    - Develop a standard software requirements specification (SRS) outline for their own organization
    - Define the format and content of their specific software requirements specification
    - Develop additional local supporting items such as an SRS quality checklist, or an SRS writer's handbook

To the customers, suppliers, and other individuals, a good SRS should provide several specific benefits, such as the following:

- *Establish the basis for agreement between the customers and the suppliers on what the software product is to do.* The complete description of the functions to be performed by the software specified in the SRS will assist potential users in determining if the software as specified meets their needs or how the software must be modified to meet their needs.
- *Reduce the development effort.* The preparation of the SRS forces the various concerned groups in the customer's organization to rigorously consider all of the requirements before design begins and reduces later redesign, recoding, and retesting. Careful review of the requirements in the SRS can reveal omissions, misunderstandings, and inconsistencies early in the development cycle when these problems are easier to correct. Requirements that are infeasible or exceptionally difficult, risky, and/or time-consuming can also be identified, and negotiations can take place concerning their removal or modification.
- *Provide a basis for estimating costs and schedules.* The description of the product to be developed as given in the SRS is a realistic basis for estimating project costs and can be used to obtain approval for bids or price estimates.
- *Provide a baseline for validation and verification (V&V).* Organizations can develop their validation and verification plans much more productively from a good SRS. As a part of the development contract, the SRS provides a baseline against which compliance can be measured.
- *Facilitate transfer.* The SRS makes it easier to transfer the software product to new users or new machines. Since the use of standards makes is easier for numerous organizations to use a standard developed requirements specification without additional training, customers thus find it easier to transfer the software to other parts of their organization, and suppliers find it easier to transfer it to new customers.
- *Serve as a basis for enhancement.* Because the SRS discusses the product but not the project that developed it, the SRS serves as a basis for later enhancement of the finished product. Although the SRS may need to be altered, it does provide a foundation for continued production evaluation.

1.5.4 What the software requirements specifications should not contain. The SRS should specify valid design constraints, not needless designs. This particular issue is very hard to

enforce or even identify. Any designs that is absolutely required by the customer or acquirer must be included under the category "design constraints."

Other things that should *not* be specified in a project requirement are programmatic: (In the U.S. these items are found in the project plan or other plans, but in Europe the SRS often contains these items.)

- Project cost and schedule
- Software quality assurance procedures
- Software development methods
- Acceptance test procedures
- Project reporting procedures.

Finally, an SRS does not specify a service. A service contract is a legitimate contract, but it is NOT a software requirements contract.

1.6 Practical Considerations

1.6.1 Iterative process. This is a process in which the developers sometimes initiate the design phase before the requirements are complete or verified. In the event that errors are found in the requirements specification, some or all of the requirements analysis is reinitiated and the requirements problems or errors are fixed. At the new completion of the requirements phase, the design phase is again reinitiated. In the event that more errors are found, the requirements phase is again reinitiated. And the cycle begins all over again.

Dr. Winston Royce (the well-known developer of the "waterfall chart") believed that you can begin the design of a software system as soon as 80% of the requirements are completed [Royce 1987].

1.6.2 Change management. Project managers are responsible to see that [Thayer 2004]:

- All requests for changes are presented as formal requests in writing
- All change requests are reviewed and changes are limited to those approved
- Type and frequency of change requests are analyzed and evaluated
- The degree to which a change is needed, and its anticipated use, are considered
- Changes are evaluated to ensure they are not incompatible with the original system design and intent
- No change is implemented without careful consideration of its ramifications
- The need to determine whether a proposed change will enhance or degrade the system is emphasized

1.6.3 Requirements attributes. All requirements need to have the following attributes:

- ***Complete*** — No requirements are overlooked
- ***Consistent*** — No individual requirements or set of requirements conflicts with any other
- ***Correct***— No error exists that will affect design

- ***Clear*** — There is only one semantic interpretation (i.e., unambiguous)
- ***Modifiable*** — Any necessary changes can be made completely and consistently (this encourages having a requirement specified in only one place)
- ***Verifiable*** — Some finite process exists to verify that the product meets the requirements
- ***Traceable and Traced*** — An audit trail exists from requirements to tested code and back
- ***Implementation-free*** — Design and management requirements are excluded

1.6.4 Requirements tracing. *Requirements tracing* is the identification and documentation of the derivation path (upward) and allocation/flowdown path (downward) of requirements in the requirements hierarchy. Requirements tracing is a valuable software maintenance tool as well as a requirements analysis and design tool. A requirements traceability matrix will allow the maintainer to identify the breadth and depth of the impact of a required software system change.

If a traceability matrix was not produced during the software development phase, a limited matrix will have to be developed around any system changes required by the maintenance effort. This approach will reduce the impact of the so-called "domino effect" in which a software change spins off numerous unneeded and unwanted secondary changes.

1.6.5 Measuring requirements. See Chapter 16 regarding software requirements measurements

1.6.6 Software requirements tools. See the paper by Stuart Faulk [2012] on *Understanding Software Requirements.*

References

Additional information on the *software requirements* KA can be found in the following documents:

- **[Christel & Kang 1992]** Michael G. Christel and Kyo C. Kang, *Issues in Requirements Elicitation*, CMU/SEI-92-TR-012, ESC-TR-92-012, Software Engineering Institute, September 1992.
- **[Cleland-Huang 2004]** Jane Cleland-Huang, "Software Requirements," in *Software Engineering, Vol. 1: The Development Process*, R.H. Thayer and M. J. Christensen, (eds.), IEEE Computer Society Press, Los Alamitos, CA, 2004.
- **[Davis 2005]** Alan M. Davis, *Just Enough Requirements Management: Where Software Development Meets Marketing,* Dorset House Publishing, New York, 2005.
- **[Faulk 2012]** Stuart R. Faulk, "Understanding Software Requirements" in R.H. Thayer and M. Dorfman, *Software Engineering Essentials,* Vol. I: The Development Process. 2012, Chapter 1.1.
- **[IEEE Std 1362-1998]** IEEE Standard 1362-1998, *IEEE Guide for Information Technology—System Definition—Concept of Operations (ConOps) Document.*
- **[IEEE Std 610.12-1990]** Adapted from *IEEE Standard Glossary of Software Engineering Terminology*, Standard 610.12-1990.
- **[IEEE Std 830-1998]** ANSI/IEEE Standard 830-1998, *IEEE Recommended Practice for Software Requirements Specifications.*

- **[Royce 1987]** Dr. W.W. Royce, Pers. comm. 1987.
- **[SWEBOK 2004]** E. Bourque and R. Dupuis, (eds.), *Software Engineering Body of Knowledge* (SWEBOK), IEEE Computer Society Press, Los Alamitos, CA, 2004.
- **[Thayer 2004]** R.H. Thayer, "Software System Engineering," in *Software Engineering Vol. 1: The Development Process*, R.H. Thayer and M. J. Christensen, (eds.), IEEE Computer Society Press, Los Alamitos, CA, 2004.
- **[Weigers 2003]** Karl E. Weigers, *Software Requirements* (Paperback), 2nd Edition, Microsoft Press, 2003, 544 pages, ISBN-13: 978-0735618794 (Recommended as CSDP/CSDA exam reference book by the IEEE Computer Society).
- **[Wood & Silver 1995]** Jane Wood and Denise Silver, *Joint Application Development*, 2nd ed., Wiley, New York, 1995.

Chapter 2.1a

Software Design: Introduction & Overview[7]

David Budgen
School of Engineering & Computing Sciences
University of Durham
Durham City, England

1. The Nature of Software Design

While software is almost all-pervasive in the modern world, the act of designing software poses some very significant challenges. The aim of this overview paper is therefore to describe the key properties of software; to explain how these influence the design process; and to review some major examples of the strategies and forms that have evolved to address them.

In the context of software design, we are seeking to create a 'solution' that involves creating some form of artefact from appropriate software forms. In the rest of this first section we therefore examine how software design is influenced by the nature of design activities, by the particular characteristics of software, by the context within which software design is performed, and by our ideas about what might constitute a 'good' design.

First of all though, we should ask the question:

What exactly is the purpose of design?

The answer to this question essentially defines the scope of this paper by identifying the issues that any design solution should address. There are of course many possible answers, some of which will reflect the context within which a particular design task is undertaken. However, a reasonably generic answer is:

To produce a plan (or model) that represents a workable (implementable) solution to a given need.

The following sections then examine some of the conceptual tools that we employ to assist with software design. Section 2 discusses some of the different ways that software can be organized, broadly classified as *architecture*. Section 3 examines how we can *visualise* ideas about a design through the use of different notations. Together, these then underpin Section 4, where we review some well-established ways of organising the *process* of designing.

1.1 The nature of designing

Software design activities need to conform to the constraints imposed by the nature of designing in general. Design problems are widely recognised to be 'wicked' problems [Rittel & Webber, 1984], sometimes also termed 'ill-structured problems.' Such problems are characterised by having such properties as:

---

7. Based on "Software Design: An Introduction," by David Budgen, which appeared in R.H. Thayer and M. Dorfman (editors), *Software Engineering*, Volume I, 3rd edition, IEEE Computer Society Press, Los Alamitos, CA,

- no *true/false* solutions, with many possible solutions that can only be ranked as better or worse from a particular perspective;
- no *definitive formulation*, so that the specification and understanding of a problem are bound up with our ideas about 'solving' it;
- no *stopping rule* that can be used to determine when an optimum design has been achieved;
- no immediate and ultimate *test* that can be used to determine that a design solution fits the needs of the problem (in our case, the requirements).

A key consequence is that the activity of designing cannot ever be a 'procedural' or 'defined' process. Indeed, the design process is essentially *empirical* or *opportunistic* in nature [Hayes-Roth & Hayes-Roth, 1979; Williams & Cockburn, 2003], and involves exploration of a potentially very large 'solution space'.

These characteristics can be illustrated by a very simple example of a design task that will be familiar to many people, which is that of moving home. When we move to a new house or apartment, we are faced with a classical design problem: namely that of determining where our furniture is to be placed. Indeed, we may also be expected to provide a design 'plan' for the removal company to indicate our intentions.

There are of course many ways in which furniture can be arranged within a house or apartment. For each item, we need to decide in which room we want it to be placed, perhaps determined mainly by functionality, and then where it might go within that room. We might choose to concentrate on getting a good balance of style and arrangement in more public rooms at the expense of others. The form of the building will also provide constraints, in that furniture should not block doors or windows and should leave power outlets and other connections accessible.

The process of designing software is really not so very different from this, and exhibits the same forms and issues, further complicated by some properties of software itself that we now need to consider.

1.2 Characteristics of software

Given that an important task for the software designer is to formulate some form of abstract design model that represents his or her ideas about a design solution, we might ask what causes this to be so great a challenge. Fred Brooks [1987] suggested that some software characteristics that contribute to this include:

- *The complexity of software*, with no two parts ever being quite alike, and with a process or system having many possible states during execution.
- *The problem of conformity* that arises from the very pliable nature of software, so that the designer is expected to tailor software so as to meet the needs of hardware, of existing systems, or to meet other 'standards'.
- *The (apparent) ease of changeability* of software means that users are apt to expect changes to be made without an appreciation of the true costs (in terms of money, resources and structure) that these imply.

- *The invisibility* of software means that our descriptions of our design ideas lack any clear visual link to the form of the end product, and so are unable to help with comprehension in the same way that occurs with more 'physically' connected forms of description.

Together these explain why it is so difficult to clearly and unambiguously 'capture' any ideas that we might have about the design for a software system. Designing is always a challenging activity, and for software it is rendered even harder by these characteristics.

1.3 What constitutes design knowledge?

The process of learning about design may well involve both a period of academic study and also a spell of 'apprenticeship', which involves learning directly from a more experienced practitioner by working with them in some way. Regardless of how this may be organized, the aim is to provide a fledgling designer with both experience and an understanding of how they can most effectively deploy the design elements available in their particular medium.

Studies of the role that experience plays in designing software are consistent with this view of design knowledge. Adelson & Soloway [1985] noted a range of techniques being used, with the choice being dependent both upon the expertise of the designer and also their familiarity with the given problem domain. They particularly noted the use of 'labels for plans' by experts, whereby a designer identified a part of their task for which they could reuse prior design experience, 'labelling' this intention at an abstract level.

A later study of expert designers by Guindon [1990], observed that a variety of *knowledge schemas* were employed, from simple rules to part solutions. Indeed, for object-oriented development, Détienne [2002] has noted the use of three forms of knowledge schema: *application domain* schemas, *function* schemas, and *procedure* schemas. From a cognitive perspective, a *schema* can be considered as some form of internal 'knowledge structure' that an expert employs to represent 'generic concepts stored in memory', and it is their possession of a richer set of internal schema that largely distinguishes experts from the less experienced.

1.4 The software development lifecycle

Designing software is not an isolated and independent activity. The eventual system as implemented will be expected to meet a whole set of user needs (reminding us of the criterion of "fitness for purpose"). In a classical software lifecycle such as the well-known 'waterfall' model, it is expected that these needs will be determined in advance through some form of *requirements elicitation* process, possibly aided by an *analysis* of what the system is to do. But in reality, these tasks are likely to interact, both with each other and also with the activities of design, since each step can both constrain later steps and also reveal inconsistencies in the earlier ones.

In addition, it is expected that the designer will provide a set of specifications for those whose job it is to construct the system. These will need to be clear, complete and unambiguous, yet despite this (if such an ideal can be achieved), it is likely that further needs for change will emerge during implementation too. Furthermore, over and above such immediate issues, the designer also needs to think about the long-term evolution of a system and seek to devise a structure that can accommodate any likely changes.

The sheer difficulty of balancing this medley of conflicting goals has led to the emergence of a quite different way of thinking about the software development context that we describe as *agile methods*. These seek to recognise the uncertainties in the overall development process, and assume that the form of system will evolve as understanding of its role emerges. For such forms,

the role of design becomes much more closely entwined with those of requirements elicitation and implementation. As far as design is concerned, an important aspect of such methods is to ensure that constant change and evolution does not unduly undermine design qualities and structures.

1.5 Quality factors for design

Quality can be an elusive concept at best, and given the properties of software discussed above, it is not surprising that this is particularly so for software design. Indeed, our ideas about quality are almost always bound up with our particular relationship with the system itself.

Having suggested that the concept of 'fitness for purpose' was a paramount goal for any system, we have to recognise that this cannot be directly measured, nor, indeed, is it absolute. Simply performing the specified task(s) correctly and within the given resource constraints may not be enough to achieve fitness for purpose. An example of this would be a system that is expected to be in service for at least ten years, with modification at frequent intervals. For this case, our notions of fitness for purpose are very likely to incorporate ideas about how well the overall structure can be adapted to accommodate the likely changes without compromising the other qualities. The converse is equally true. Where a system is urgently required to meet a short-term need, getting a system that works (correctly) will be more important than ensuring that it can be modified and extended.

Space limits what we can say here about quality factors, but a useful group to note are those that are generally referred to as the '*ilities*'. The exact composition of this group may be dependent upon context, but the key ones are generally those of *reliability*, *efficiency*, *maintainability*, and *usability*. They describe rather abstract 'top-level' properties of the eventual system and cannot be easily accessed from design information alone.

Indeed, devising suitable ways to measure design information in a reliable and systematic manner is something of a challenge. While at the level of implementation we can employ basic code measurement (metrics) by such means as counting lexical tokens [Fenton & Pfleeger, 1997], the variability and the weak syntax and semantics associated with design notations makes such an approach less appropriate for design. Hence, more qualitative forms such as design walk-throughs and design reviews may well be more suitable [Parnas & Weiss, 1987].

2. Software Architecture

The idea of *architecture* in connection with software began to emerge in the early 1990s. To some extent this probably reflected a growth in the different ways of organising software systems. Where once almost all software was organized on the basis of a main program unit invoking a set of sub-programs (what we now usually term 'call-and-return'), later systems began to be organized around other forms such as objects, processes that were spread across a range of different computers, and large databases.

Various terms were used to capture these ideas, and as usual when something new emerges, these were not always used consistently, thus reducing their value as a 'vocabulary'. In the next subsection we examine some of the early ideas about resolving this issue, and how these have evolved. We then briefly look at some examples of what we now term *software architectures*.

2.1 Basic concepts

An early, and very clear discussion of this appeared in a paper by David Garlan and Dewayne Perry [1995], written as an introduction to a collection of papers on this topic. In this, they

examined some of the roles that the concept could perform, including: helping with understanding of a high-level design through the provision of an abstract vocabulary; helping to identify where elements could be reused; and providing an understanding of "a system is expected to evolve". Indeed, in many ways, the architecture of a system is simply the abstract form of its top-level design.

The book by Mary Shaw and David Garlan [1996] provided a valuable baseline for the emerging ideas. In this, they employed a basic framework of describing an architecture in terms of the kinds of *components* and *connectors* employed in a given system architecture. Their book examined and classified a number of *architectural styles* based upon these ideas, and hence had the benefit of clarifying a vocabulary that was increasingly being used, but not always consistently.

Ideas about architecture and about its influence upon such developments as the concept of 'software product lines,' involving the reuse of architectural ideas for multiple systems, have continued to evolve. For a fuller understanding of this area, a book such as [Taylor et al., 2010] provides more detail as well as examples. For our purposes though, the basic concepts should be a sufficient introduction.

2.2 Some architectural styles

The concept of an *architectural style* has proved a useful one in a number of other domains. When speaking of buildings, referring to a house as being in a 'black and white' style tells us about its likely characteristics—with external wooden beams and small windows. The same concept applies to ships where the term 'naval architecture' has long been employed and where it is recognised that the overall characteristics of a ship will reflect its purpose. Aircraft carriers have large flat decks and a small superstructure to one side; oil tankers have large tanks in the main hull and small superstructures at the stern, etc. In the case of buildings, style may be dictated partly by the materials available in the period when it was constructed, while for ships, it is driven more by function.

Software architecture is probably driven by all of these: the type of elements used the way that they interact, and the purpose of the system being the main influences. Table 1 below summarises some common examples of software architectural styles, drawing upon the categorisations proposed in [Shaw & Garlan, 1996].

2.3 Architectural patterns

While the notion of architectural style tells us something about the type of elements within a system and how they interact, an *architectural pattern* focuses more upon the overall organisation of the elements. A useful introduction with illustrations is provided in [Buschmann et al., 1996]. Here, we illustrate the concept by discussing two particularly familiar forms.

Model-View-Controller (MVC)

This is a widely-used pattern (many student projects have this form) in which the overall design of an interactive application is organized as three elements that have clearly-defined roles and functionality:

- The model contains the core functionality and any relevant data
- Views provide information to the user

- ❑ Controllers handle user input. Each view has an associated controller that also handles related forms of input

The user interface then consists of views and controllers together and is independent of the model itself. However, the model will need to propagate information about changes to the controllers. Such an approach makes it easy to change the interface for a new platform, or to employ new forms for presenting the information. Figure 1 illustrates the MVC structure.

Table 1: Some examples of architectural style

Category	Characteristics	Examples of styles
• Data-flow	Movement of data with recipients having no control of content	Batch sequential Pipe-and-filter
• Call-and-return	Single thread of control determining order of computation	Main program/subprograms Classical objects
• Data-centered repository	Focus upon a complex central data store	Transactional databases Client-server Blackboard

Layers

The *layers* pattern is another form that is employed in many roles. It is commonly (but not exclusively) used for organizing a hierarchy of protocols (such as that of the OSI seven-layer model used in computer networking). The idea is that each layer will deal with a specific aspect of communication and will employ the services of the layer below (and no other layers) while providing service to the layer above. The value of this pattern is that specific functionality related to a particular layer of abstraction is encapsulated in a layer and may easily be redeployed into a new context simply by substituting new layers below it. This form is illustrated in Figure 2, where we use part of the network OSI model to illustrate the characteristics.

3. Describing Designs

When discussing the nature of software in Section 1, it was observed that its characteristics, and particularly that of *invisibility*, provide a challenge to any attempts to visualise our ideas about a design. However, regardless of the different roles for design notations explored in this section, we need them simply to avoid *cognitive overload* in developing design ideas. There is a limit to how much we can reliably 'store' in our own working memory and so for anything but the smallest problems we simply need to find ways of visualising design ideas, even if these are not embedded in any form of 'physical reality'.

Over the years, software engineers have therefore developed a range of 'box and line' notations intended to help with this task. Whether this has really had the attention (as a design task in itself) that it should perhaps have had is a moot point. Analysed against a cognitive framework, most software engineering notations do seem to come out rather badly [Moody 2009]. Anecdotally too, experienced designers seem to produce fairly informal diagrams to help develop their

ideas, only reverting to more formal notations when these ideas need to be recorded and shared with others.

However, regardless of these issues, we still have a need for well-defined notations for such purposes as:

- documenting and exploring our ideas about a design solution;
- explaining our ideas to others (the customer, implementers, and other members of a design team . . .);
- checking for consistency and completeness of a design model

So in this section we examine a general categorisation of design notations and then examine some examples of these.

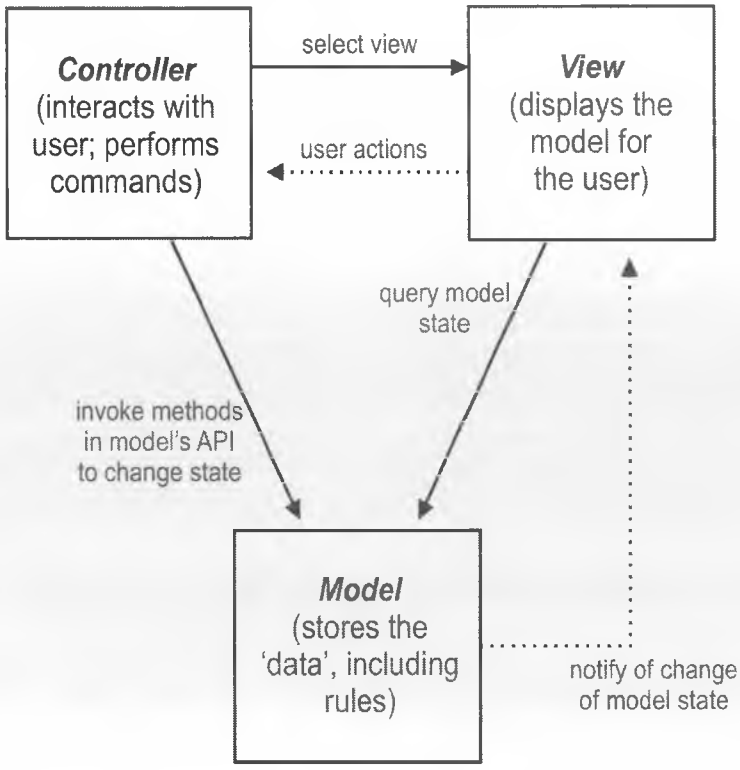


Figure 1: The Model-View-Controller pattern

3.1 Design viewpoints

The wide range of notational forms that we use can be categorised in a number of different ways. Some, such as the '4+1' model [Kruchten, 1994] are closely linked to a particular architectural style (in this case, object-orientation). For this section, though, we will employ a more generic categorisation into four different groupings, described more fully in [Budgen, 2003].

This grouping is based upon the idea of a *design viewpoint*, where a viewpoint is considered as being a 'projection' from the 'internal' design model that displays certain of the characteristics with an appropriate level of abstraction.

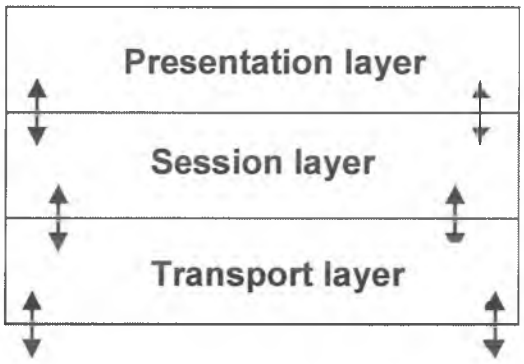


Figure 2: Example of the Layers pattern

Figure 3 provides a schematic illustration of this idea along with some examples of forms used to record these viewpoints. The four viewpoints employed here are as follows:

1. The *behavioural* viewpoint, describing the causal links between external events and system activities during system execution.
2. The *functional* viewpoint, describing the operations performed by a system.
3. The *constructional* viewpoint, describing the static interdependencies of the constructional elements that make up a system, such as objects, subprograms, and processes. (In earlier papers this was often termed the 'structural' viewpoint, but as all viewpoints have some degree of structure, this has been revised.)
4. The *data-modeling* viewpoint, describing the relationships that exist between the data objects in a system.

The term 'system' in the above definitions is used fairly loosely, since at different times we might want to describe a complete design solution, or parts of it.

We might also note that, while none of these can be considered easy to represent, the *functional* viewpoint tends to offer the greatest challenge to 'box and line' forms, partly because of its very task-centered nature.

3.2 Forms of representation

Although this section has so far largely referred to the use of 'box and line' notations, this is only one of several forms that can be used to realise the design viewpoints. The other two forms that are widely used are text and mathematical notations. Each has features that render it useful and we should avoid the assumption that all notations are necessarily diagrammatical.

Text

Text is widely used, largely in conjunction with the other forms, but also on its own. The practice of *note-making* has been widely observed in studies of software design [Adelson & Soloway, 1985; Guindon, 1990], and such notes are often organized as *lists*, which can have some degree of informal structure through indentation, numbering, bullets, etc. Ideas and descriptions can also be usefully recorded as tables.

However, in exchange for its relative ease of use, text offers only limited scope for representing any structure that may be present in information, beyond the use of lists and tables. In addition, natural language is prone to ambiguity that can only be resolved by using long and complex sequences of words (a good example is a legal document).

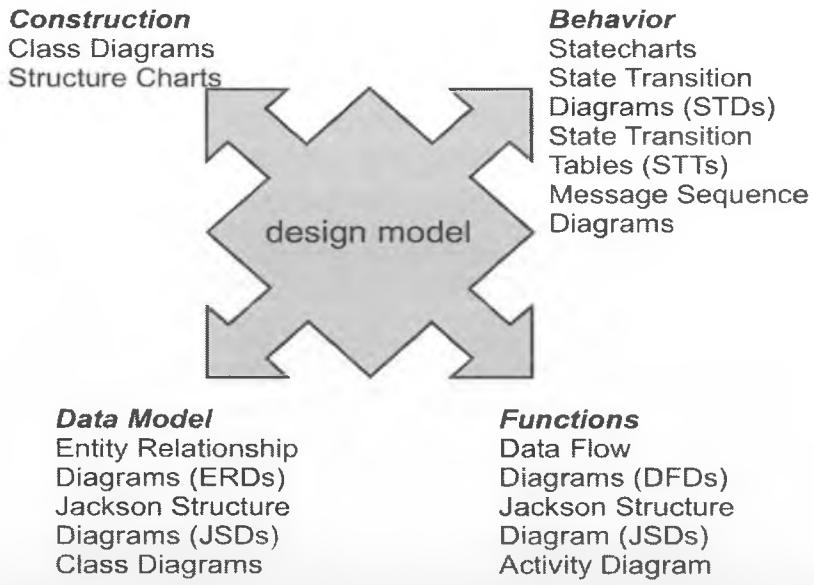


Figure 3: Examples of the design viewpoints

Diagrams

Since our examples will largely focus upon diagrams, we briefly describe two characteristics that appear to be significant for the successful use of diagrams.

The first relates to the number of symbols in use to describe the concepts (the 'elements') of a diagram. As a loose rule of thumb, the more abstract the diagram the fewer the symbols. Diagrams with a large number of symbols tend to be more complex to use. (In this context, what we might term symbols are not necessarily shapes or characters; they might be arrowheads, solid or dashed lines, etc.). A supplementary aspect of this is that we should also be able to draw the symbols easily—many designs are worked out and explored using whiteboards or pencil & paper and the designer wants to be able to concentrate on exploring an idea without needing to spend time drawing complex shapes. So, symbols should be simple in form and easily distinguished from each other.

The second is concerned with having a *hierarchy* within a notation. Large diagrams may be very difficult to understand, and many forms therefore allow the use of a hierarchy, whereby symbols at a higher level of abstraction are expanded into a 'tree' of diagrams, with each level providing greater detail. Figure 4 demonstrates this idea in a schematic manner.

Mathematics

Mathematical notations are of course ideally suited to providing concise abstractions, so it is hardly surprising that they have been employed for this purpose through the use of *Formal*

Description Techniques, or FDTs for short. Traditionally, they are usually employed for the purpose of specification, whether this be of system properties (for analysis and requirements specification), or of the behaviour and functionality of individual design elements. As with the case of text, it can be argued that these notations are most valuable when being used to support other descriptive forms, rather than being used solely on their own. On the downside, their use requires learning a set of (usually non-intuitive) symbols, and they are less well suited to describing larger-scale systems.

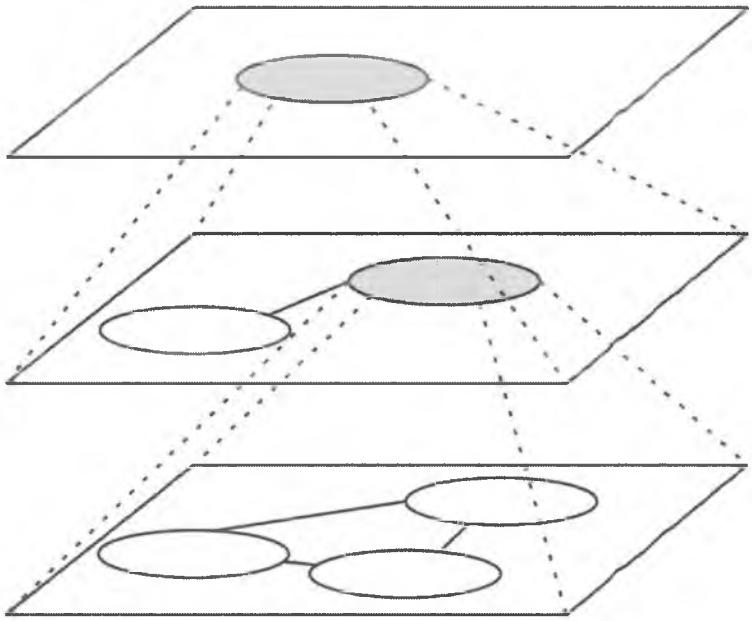


Figure 4: Hierarchy in representations

3.3 Some examples

Before discussing some simple examples of the concepts outlined above, we should note that the old saying about fire, that it "makes a good servant but a bad master" applies equally to the use of diagrams when designing software.

Any given form of diagram will have an established syntax (how we draw it) and semantics (what is meant by the symbols, their positioning, etc.). However, the aim of a diagram is to *assist* with the process of design—too slavish a regard for syntax in particular during the evolution of design ideas can be a handicap. Indeed, observation suggests that experienced designers often produce fairly informal diagrams while developing their ideas and formalise these later. This is a point that we will return to when considering the use of tools to support the design process.

Below, we briefly describe how the different viewpoints are used within different system forms and provide some simple examples of how these might be organized.

We should first also observe that where the object-oriented architectural forms are concerned, the *Unified Modeling Language* (UML) is widely supported by a range of tools and well-

defined forms [Rumbaugh et al., 1999]. However, the term 'unified' as used here refers to the drawing together of the ideas of the three methodology 'gurus' who wrote the standard, and there has subsequently been some significant empirically-based questioning about its complexity and general usefulness [Moody, 2009; Budgen et al., 2011]. Hence, given its wide recognition, we have mainly illustrated the viewpoints with forms that may be less familiar to those accustomed to the UML, in order to demonstrate the breadth of the concepts.

The constructional viewpoint

Various forms of 'object' and 'class' diagram have been developed, although most tend to be broadly similar in form to the UML *class diagram*. Indeed, in many ways, they closely resemble the entity-relationship diagrams (ERDs) used for data modeling, although obviously the range of relationships included in such forms is more related to object and class interactions, such as aggregation, uses and inheritance.

In contrast, Figure 5 shows an example of a *Structure Chart*, a form that is generally associated with a call-and-return form of architectural style.

In this notation, the boxes represent sub-program units, and the lines joining them represent invocation (akin to the use of methods in object-oriented terminology). Order and frequency of invocation are not shown, only its existence. The small arrows provide details of the parameters being passed—there are other drawing conventions used with this form, as well as some variations (some authors prefer to provide a table detailing the parameters).

This form of 'call graph' sometimes has another role, in that it is often created by maintainers of software (either manually, or via support tools) in order to gain a clearer visualisation of the hierarchy of units within an existing system.

The functional viewpoint

As indicated above, this aspect of a system is probably the most difficult one to capture in diagrammatical form. *Data Flow Diagrams* (DFDs) capture functionality in terms of how the operations of the system affect the forms of information it holds. (Tom De Marco suggests that this form is much older than computing, and was certainly used in the early 1900s to model how teams of clerks processed things like insurance claims.)

A related but different way to describe function in this way is in terms of workflows (focusing on the tasks rather than the data). Figure 6 shows one of the UML notations used for this (the *Activity Diagram*) being used to describe part of the process of conducting a systematic review (see [Budgen et al., 2011] for an example of this form of study).

Here the focus lies upon the activities being performed by the researchers and where the related flows are divided and then recombined.

The behavioural viewpoint

The idea of *state* is a valuable one in computing, and particularly when considering object models, since we can view the operations on an object (performed by its methods) as either modifying its state or providing the end-user with information about its state. Indeed, state models can be used to describe the real-world activities that the system is intended to interact with, the activities of the system itself, or the activities of the parts making up the system (such as the objects).

Figure 8 shows an example of a *Statechart* [Harel 1987] being used to model the part of an air traffic control system concerned with 'stacking' of aircraft that may not yet be able to land. We might also note that the form of the State Diagram that is used in the UML differs relatively little from this.

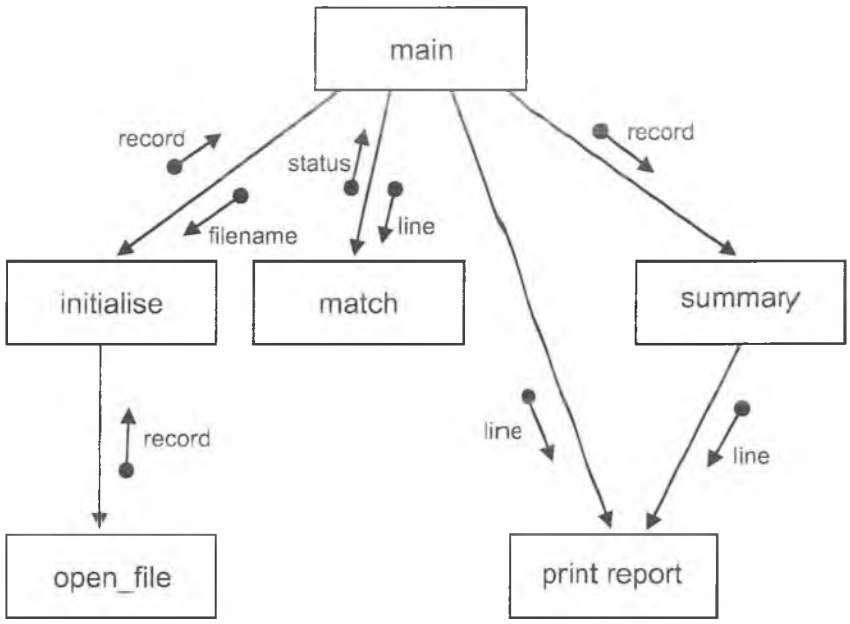


Figure 5: Example of a Structure Chart

Here states can be combined to form 'super states' and also decomposed into sub-states. Our figure labels only a few of the transitions in order to keep it reasonably clear—the reader might wish to complete the missing ones.

The data modeling viewpoint

An example of the classic *Entity-Relationship Diagram* is shown in Figure 7, and continues the theme set by the example used in Figure 8.

There are different drawing conventions associated with ERDs and this one uses a fairly long-established set of conventions by which the entities are shown as boxes, attributes of entities are in boxes with rounded corners, and the 'arity' of the relationship is shown as numbers on the lines linking to the entities. (Here, some *n* aircraft can be held in the stack.)

While this form of model is often largely associated with the data-centered repository architectural style and the use of databases in particular, it does have wider application since we may also be interested in modeling how the data entities within a system are related. Such models might be hierarchical (i.e., how is the information associated with some system-level entity mapped onto lower level structures).

4. Organising the Design Process

In section 1.3 we discussed the concept of *knowledge schema*, with expert designers owning a richer [and more organized) set of such schema than less experienced designers. Since a knowledge schema is an internal representation of that knowledge, one of the challenges since the early days of software development has been to find ways of codifying that knowledge in such a way that the less experienced designer can learn design skills as quickly and effectively as possible. While the ideal might be that of a 'design studio' where the 'novice' can sit beside and learn from the 'master', this is rarely a practical option. Indeed, expert designers are likely to be a rare commodity in most organisations [Curtis et al., 1988].

The earliest forms used for knowledge transfer were often termed 'software design methods', using what we often term a *plan-driven* approach. As experience of design grew, and the range of software architectural styles expanded with technology, so did ideas about how design knowledge could usefully be organized. Design *patterns* offer an approach that is often considered as more appropriate for object-oriented forms and *component-based* design approaches have tried to employ a 'black-box' model that in some ways approximates to the way that electronic hardware design is organized. The classical approach of the design method was often seen as being too closely linked to the waterfall model of development, and *agile methods* have subsequently emerged as one means of making the software design process more responsive to its environment (including the business needs of the customer for the software).

In the following sections we provide a brief introduction to each of these concepts and identify some key examples and references. Because space precludes an in-depth exposition of their features, the discussion necessarily has to be at a fairly high level of abstraction. The final section discusses the role of design support tools and their limitations.

4.1 Plan-driven design

Plan-driven design approaches essentially structure design knowledge as follows:

- A set of procedures that should be followed in order to create a 'design model' that eventually evolves into the actual design plan.
- A set of descriptions, usually in the form of diagrams, which are used to represent the design model in various stages of evolution.
- A set of heuristics, that are based upon experience of using the method and might relate to such things as how to go about creating the initial model for a particular form of problem, or how to adapt the process for specific needs.
- So these three elements essentially represent the knowledge schema as conveyed through a method. The procedures are organized around a strategy. Usually this is either based upon a 'top-down' [decompositional) form, or a 'bottom-up' (compositional) form.
- One of the earliest (and quite successful) design methods was known under various names, but can be summarised as *Structured Analysis and Design*. Here the initial model was based upon analysing the data flow involved in the whole system (using some form of *Data Flow Diagram*) and the procedures were concerned with transforming this into a call-and-return form of model based upon a main program and sub-programs (usually represented as a *Structure Chart*). Here the procedures were concerned with constructing the original model and then performing what were usually referred to as *transaction*

analysis (identifying the different functions in the model) and *transform analysis* (changing the form of the model to map onto executable structures). The heuristics helped with identifying such concepts as the 'central transform' for a particular transaction. A good example of a textbook describing this process is [Page-Jones, 1988].

The original strategy was essentially one of functional decomposition, but perhaps reflecting both the growing size of systems and also greater experience, other, more compositional forms such as 'event partitioning' were later developed.

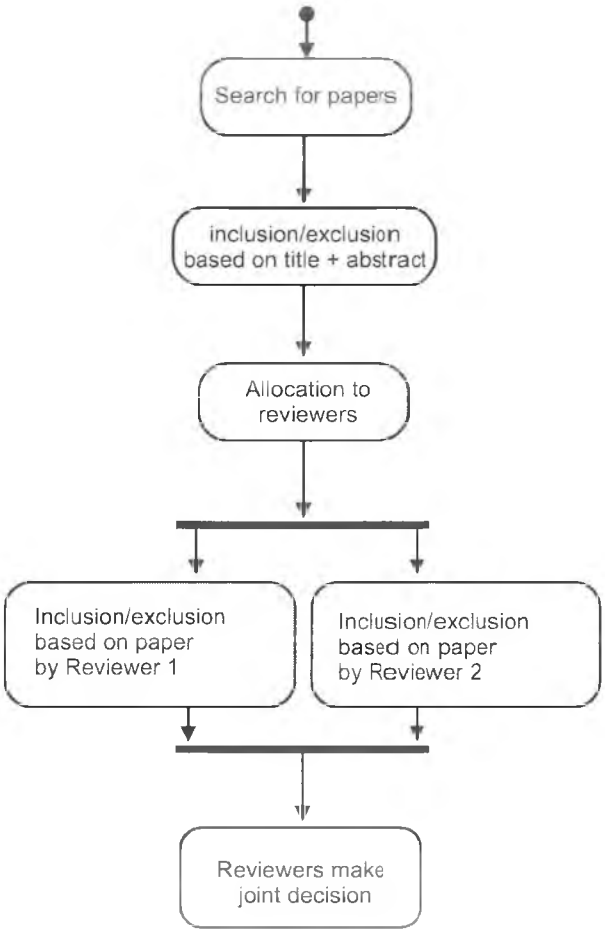


Figure 6: Example UML Activity Diagram

While the model could be and was extended, especially with the addition of real-time modeling features such as *State Transition Diagrams*, it was essentially limited by the use of rather one-dimensional models and also by being tied to an architectural style (call-and-return) that was gradually replaced by the now dominant object-oriented forms. Hence evolution essentially ended in the late 1980s, although its basic influence should not be under-rated as has been noted by Avison & Fitzgerald [1995].

The emergence of the object-oriented (OO) paradigm created problems for plan-driven forms. While these had proved quite effective when designing around such architectural styles as call-and-return and distributed processes, objects represent a much more complicated end model.

For this paper we will side-step the (sometimes thorny) issue of exactly what constitutes an 'object'. The terminology relating to objects is now fairly well established, and for this discussion we will assume that objects are created from classes and that objects provide for encapsulation of their internal state and have public methods that can be used to inspect or modify that state. Objects can also inherit part of their structure from parent objects.

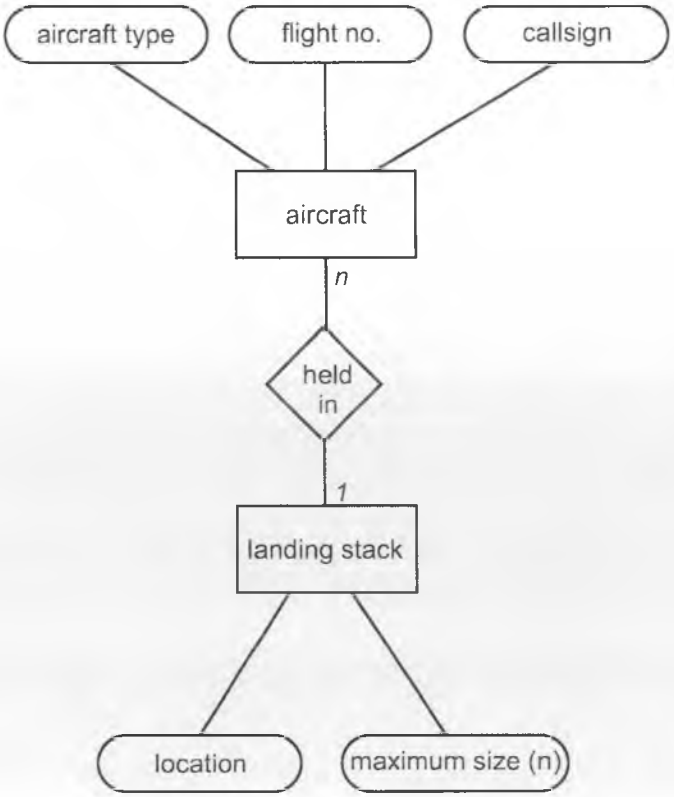


Figure 7: Example entity-relationship diagram

Two characteristics from the above, rather brief, outlines have provided a substantial problem for plan-driven approaches. One is encapsulation, while the other is inheritance. Neither of these fits well into the forms of description or procedure that were used for earlier design methods, nor have both continued to provide particular challenges for methodologists.

Through the 1980s and 1990s, a wide variety of OO methods were developed. Those of the 'first generation' were largely evolutionary in nature, in the sense that they derived many of their ideas from earlier forms and often attempted to use non-object-oriented forms of system analysis. Later methods were more revolutionary, in the sense of using quite different (and more complex) procedures than those of earlier methods, with a stronger emphasis upon composition.

A key problem, regardless of strategy, has been to identify the 'right' objects to use for a given design problem. While this tends to favour a compositional strategy, determining the choice of objects is still a complex one. Indeed, Etienne [2002] has noted that "early books on OO emphasized how easy it was to identify objects, while later ones, often by the same authors, emphasize the difficulty of identifying them."

A very comprehensive review of this theme, including descriptions of some 18 OO methods, is provided in [Wieringa 1998]. Wieringa particularly noted that the use of forms such as DFDs was incompatible with object-oriented structuring because the enforced separation of data storage and data processing implicit in the DFD did not map onto the encapsulation of data and related operations embodied in the OO model. He did also note that there was "overwhelming agreement that the decomposition must be represented by a class diagram, component behaviour by a Statechart, and component communications by sequence or collaboration diagrams," although the detailed forms of these varied quite extensively.

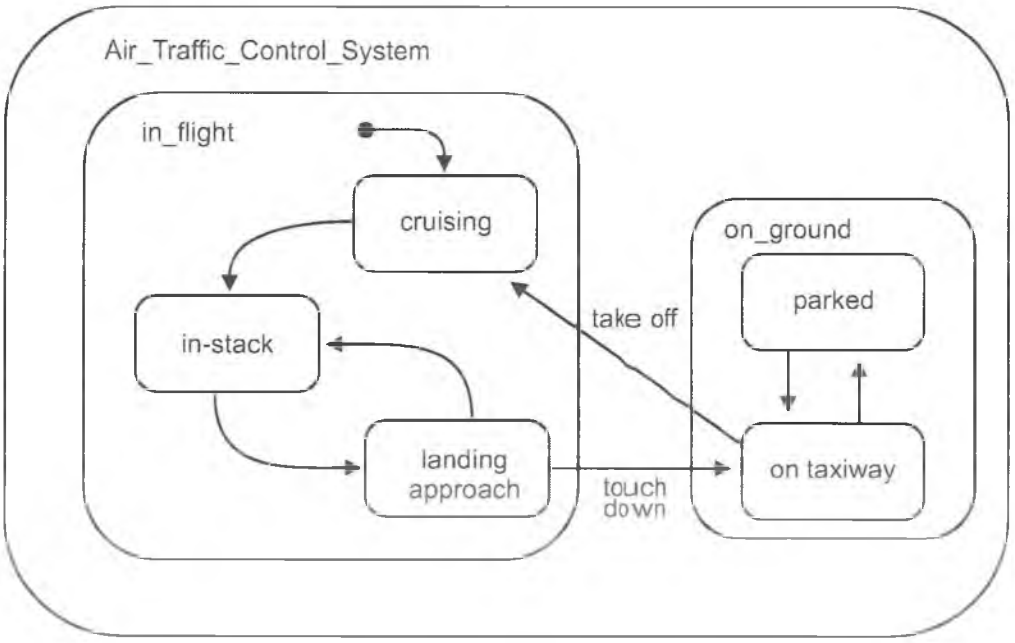


Figure 8: Example of a Statechart

By the late 1990s, the ideas of the major players in the OO method domain (Booch, Jacobson, and Rumbaugh) converged to create the *Unified Process* which can perhaps be considered as a 'third generation' method. Their form is much more complex—see Jacobson et al. [1999]—and indeed, is almost an intermediate form between that of earlier plan-driven forms and the agile methods that we discuss below. Certainly, this seems to represent the 'outer limit' as far as the development of plan-driven methods is concerned.

4.2 Design patterns

Design patterns offer a quite different way of codifying design experience for reuse by others. Unlike architectural patterns, which describe the overarching form of the whole system, a design pattern is usually concerned with organising a part of a design. Curiously, although the idea of the software design pattern resonates with the idea of 'labels for plans' identified in such empirical studies of designers as that of Adelson & Soloway [1985], the design pattern community have instead drawn their inspiration from the ideas of an architect, Christopher Alexander. Alexander, et al. [1977] characterise a pattern as:

- describing a recurring problem;
- also describing the core of a solution to that problem;
- with that solution being capable of being reused many times without actually using it in exactly the same way twice.

For software design, the pioneering work that established the concept widely as well as providing a standard for cataloguing patterns is the book by Gamma et al. [1995], with the authors (and the book) often referred to as the 'gang of four' or *GoF*. Their book catalogues some 23 design patterns. While a recent survey of software developers with extensive experience of pattern use [Zhang & Budgen, 2010] suggests that not all have proven to be equally useful (and six of them to be of very questionable use), there is little question that patterns such as *Observer* and *Abstract Factory* provide useful guidelines about how to structure systems for ease of extension and change.

The key point about a pattern is that it is not a template into which the would-be user simply plugs in their own choice of objects. A pattern is a way of organising part of a design and as such, needs to be realised in a manner that fits local requirements, whatever their form might be.

As defined by the GoF, patterns fall into two categories in terms of their *scope*: classes or objects, with most patterns addressing the use of objects. They are also categorised by their *purpose*, whereby:

- *creational* patterns are concerned with how and when objects need to be created for some purpose;
- *structural* patterns are concerned with the ways that objects and classes are composed together;
- *behavioural* patterns address the interaction between objects/classes and the way that responsibility is shared between them.

While patterns are unquestionably a valuable addition to the designer's repertoire, the enthusiasm of the pattern community for finding and documenting new patterns needs to be regarded with some caution. In particular:

- Over-enthusiastic use by inexperienced designers may lead to poorly structured designs. Sommerville [2010] argues firmly that patterns are best employed by more experienced designers who are better able to recognise when a design is truly of a generic form.
- The impact of using patterns is apt to be found during maintenance activities. Evidence here is patchy, but the paper by Wendorff [2001] provides some illustrations of the hazards of misuse, taken directly from experience.

To illustrate the concept, we will briefly examine the example of the widely used *Observer (293)* pattern. (By convention, when referring to patterns from the GoF, we often append the page number to its name.) Observer provides an example of an object behavioural pattern, in that it concerns objects rather than classes and addresses a problem that is related to the dynamic behaviour of those objects.

The situation that it addresses is one where a change of state that occurs in one object requires that (a variable number of) other objects then need to be notified so that, where appropriate, they can change their state to reflect this. A good illustration of such a situation occurs with a spreadsheet (the 'subject') and a data graphing tool (the 'observer') that is providing a chart of the data in the spreadsheet. If we change the data values in some of the spreadsheet cells, then we expect that the graph will change in response, and that it will do so without any need for us to do anything.

Figure 9 shows a simple object model that represents this one-to-many situation, usually referred to as a 'publish-subscribe' model. In essence, (concrete) observers register with the relevant subject; when a state change occurs in the subject it issues a *notify ( )* message to all registered observers; and the observers then perform appropriate *update ( )* operations to obtain the details of the change that might affect them.

Patterns emphasise the use of composition and interfaces over inheritance. When identifying a pattern, the goal is therefore to identify the parts of a design that are likely to vary, and to encapsulate these so that these parts of the system can vary without affecting others [Freeman et al., 2004] This delegation of changeable elements then creates the required flexibility for patterns such as Observer. Observer also demonstrates the idea of loose coupling that is common to many patterns. It also means that the key information remains under the control of a single object.

4.3 Agile methods

For plan-driven design methods, the procedures involved nearly always assume that a fairly complete requirements specification is available at the start of the design process. They are therefore implicitly based upon a traditional 'waterfall' model of development. Since many such methods were developed with 'data processing' applications in mind, this is not wholly unreasonable. However, as the computing environment has evolved from mainframes through personal computers to the internet, expectations of software have changed and become more fluid.

The late 1980s and the 1990s saw the development of ideas about *rapid application development* (RAD), based at least in part upon the idea of developing a system through a series of incremental stages. Barry Boehm's *spiral model* of development [Boehm, 1988], typifies this move from a waterfall-based context to one in which both design and implementation evolve in a stepwise fashion, adjusting as greater insight into the needs of the eventual end-users emerge.

With the new century, these ideas coalesced into more radical thinking still (although still structured in its way), and led to the *Agile Manifesto* (see Figure 10). The basis of the agile movement was the idea that requirements were themselves 'emergent' [Truex et al., 1999] and likely to be in a constant state of change as they adapted to continual changes in business needs.

A number of *agile methods* have evolved from this, generally characterised by their use of incremental development models and close interaction with the end-user (customer). Probably the best known forms are *Extreme Programming* (XP) developed by Kent Beck [2004], and *Scrum* [Schwaber & Beedle, 2002] which seems to be increasingly attracting attention. Since

these two embody quite interestingly different approaches to addressing the ideas embodied in the Agile Manifesto, we provide a very brief outline of each of their main characteristics here.

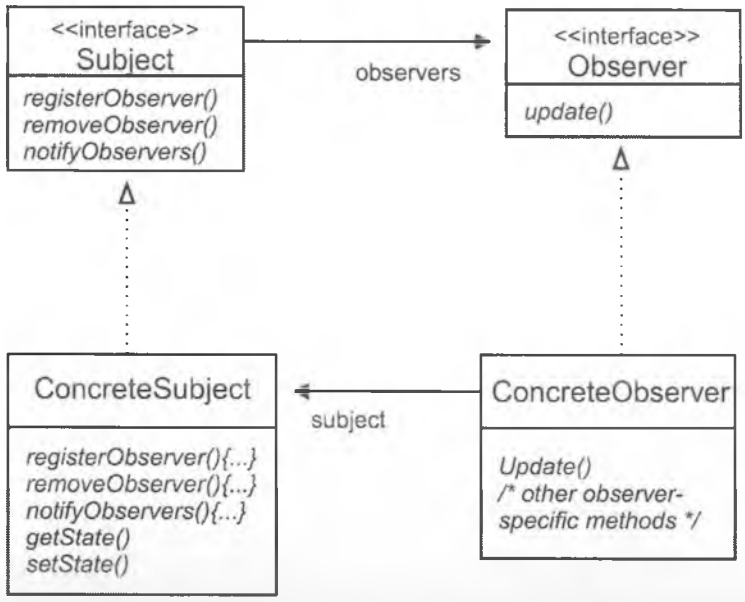


Figure 9: Structure of the Observer pattern

Extreme Programming (XP)

XP is characterised by an emphasis upon its 12 basic practices rather than by a specific form of process model. Space precludes discussing all of these in detail, but below we briefly outline some of those that are probably better known (and which help to distinguish XP from other methods).

- *Test-first programming.* The XP practice is to write the tests before writing the code and then test continuously, at the end of each day, after each increment in the design.
- *Pair programming.* Probably the best-known feature of XP. All code is written by two programmers working at a single machine, discussing their work as they go.
- *Collective ownership.* The code is owned by all of the team members, and they may make changes to it whenever they deem it necessary.
- *40-hour weeks.* Iterations should be sized so that overtime is not needed, on the basis that tired programmers make mistakes.

While empirical studies of design methods are difficult to perform, some of the features of XP have been studied empirically, in particular pair programming. A secondary study (aggregating primary experimental studies) in the form of a meta-analysis was performed by Hannay et al., [2009] The results of this were not strongly conclusive due to variations in the primary studies, but they did observe that for more complex problems, the use of pair programming did seem to result in higher quality software, whereas for simpler problems it tended to be more time-consuming than solo programming.

Scrum

Unlike XP, Scrum is less concerned with technical issues and more with the management of the overall development process. A characteristic that Scrum shares with a number of RAD and agile methods is the use of *time boxing*, by which emphasis is placed upon the use of development phases that use a fixed time interval, varying the resulting delivery of functionality as necessary. This is of course in contrast to plan-driven forms where functionality tends to be fixed, and delivery times are varied as necessary.

A Scrum project is therefore organized around a series of fixed-term *sprints*, usually of 2-4 weeks' duration. Each sprint generates a new increment, and increments are grouped to create releases of the product. The list of development tasks is termed the *product backlog*, and a portion of the backlog is usually addressed in each sprint. The team is self-organising and meets each day for a fifteen-minute *daily scrum* (the term comes from the huddle in the game of rugby football). During the scrum, they review what has been accomplished since the previous meeting, identify what is to be done that day, and note any possible obstacles. Scheduled around this is a further set of formalised meetings that occur at the beginning and end of a sprint.

Scrum also distinguishes between different roles, which are characterised as being those of *pigs* or *chickens* (from the traditional model of the cooked breakfast for which the pig is committed and the chicken merely involved). In this context, roles defined as pigs are those that carry responsibility, and hence chickens may not direct the activities of pigs.

4.4 Component-based development

In other domains, the role of the *component* has been highly successful, implying well-defined and thoroughly tested functionality and interfaces, so enabling the designer to *reuse* such components in other systems with confidence.

In the 1990s researchers began to explore how this concept might be employed for software. One problem that emerged was that of determining exactly what should be the key properties of a software component in order to enable the degree of reuse achieved in other domains. In (Budgen, 2003) one of the chapters examines this question, and discusses the evolution of the component concept—a process that has continued. Some key ideas about component-based software engineering (CBSE) that have emerged are:

- provision for *reuse* (implying a clear definition of interfaces needed to enable a 'plug and play' role in which a component could simply be viewed as a 'black box');
- *independence* of delivery (a component should not have any 'awareness' of its context);
- the existence of a *component model* that incorporates specific component interaction and composition standards;
- a *composition standard* that provides the necessary definitions of how components can be composed to form larger structures.

Elements of these are examined in the discussions of components provided in papers such as [Brown & Short, 1997] and books such as [Szyperski, 1998] and [Heineman & Councill, 2001].

However, while at one point it began to look as though a component 'market' was emerging, this has not really developed as far as many expected or hoped. The reasons for this probably include at least the following:

- the commercial potential of a component market was probably undermined by the emergence of open source systems (and components), so that vendors of commercial components were disinclined to invest too extensively;
- the lack of adequate standards, in the sense that beyond a few specialised areas such as Java APIs, there was no agreed framework that component vendors and users could depend upon;
- the emergence of the software service model and the related concept of the *service oriented architecture* (SOA), which we address below.

Service models (and particularly web services) began to develop in the early 2000s. Their use of standards for interaction (such as the use of SOAP – the *simple object access protocol*), simplicity of interface, and platform independence offered an (admittedly constrained) approach to component assembly. Architecturally, they provide a constrained form of distributed processes. However, because the computing work is undertaken by the service provider, they more readily lend themselves to creating commercial opportunities than components that the user has to execute on their own computer [Krafzig et al. 2004]. Emerging ideas about *cloud computing* take this aspect yet further.

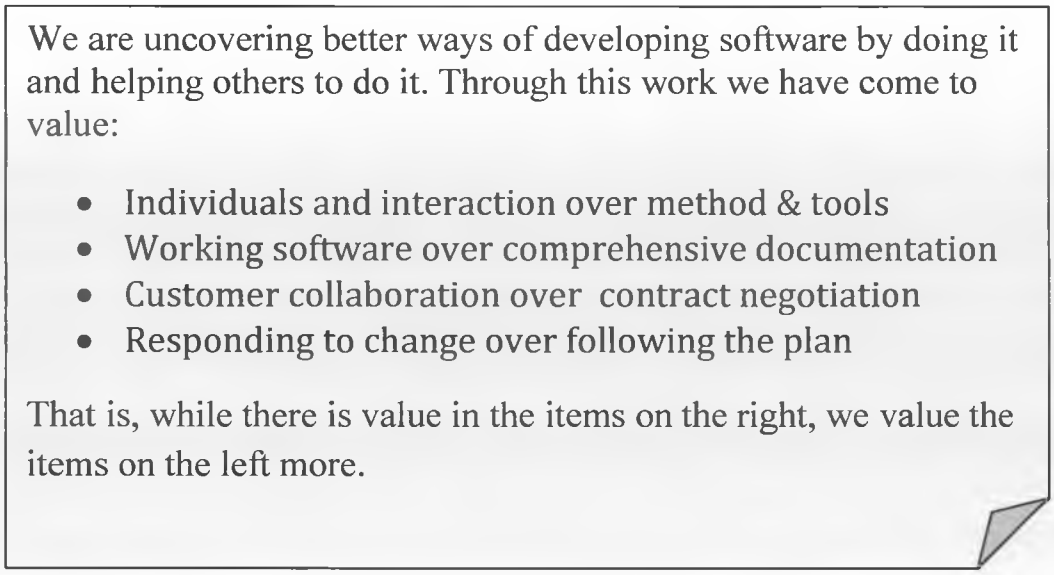


Figure 10: The Agile Manifesto

As so often, though, the service model, while appearing to overcome some of the limitations of preceding technologies, brings technical challenges of its own. Its distributed nature means that the end-user is dependent upon others to provide their computing resources; indexing and locating services offers something of a semantic challenge; and there is a multiplicity of standards and many differing definitions of SOA. Software service models do certainly offer potential and they change the computing paradigm—but they have yet to demonstrate their potential really effectively.

4.5 Design support tools

Software tools can provide support for creative activities in many domains. The word processor provides features that are useful to the author; musical composition can be made easier by score-

writing software that helps keep track of multiple parts; engineering has long had CAD (computer assisted design) software to help remove the more tedious drafting tasks.

Curiously, the one domain where design support tools have made little real progress is that of software engineering. There are a number of possible reasons for this: one is that the design of software involves manipulating abstractions, and in the early stages at least, this usually involves relatively little attention to detailed syntax and semantics. However, the examples of successful design support identified above are all from domains where design tends to take place using well-defined forms.

A second is the invisibility of our media. As observed above, our notations are essentially artificial and lack any well-defined or easily envisaged connections with the end product itself. Related to this is the need for well-established descriptive standards. Musical notation, text, 3-D engineering and drawing descriptions are all forms that have well-established conventions and standards. While for object-oriented design and for some other forms as well, the UML has at least partially met this need, as we noted earlier, in its present form at least, this may just be one step on the path towards this objective.

Software design tools have tended to provide the means of drawing diagrams using such forms as the UML. However, it is still not uncommon to find that their design makes it difficult to create diagrams with non-standard syntax or forms. Yet, as we have also observed, software designs rarely emerge 'fully fledged' and expressed in a well-defined syntax. Indeed, although the challenge of developing such tools has long been recognised [Guindon & Curtis. 1988; Reeves et al., 1995], progress with addressing this challenge seems to have been largely limited to developing tools for use in education (for example, see [Dranidis, 2007]).

5. Discussion

Software design is a large and complex topic and an overview paper such as this can only provide an outline description of some of the key issues and developments included in the topic, together with some pointers to where the reader can obtain more detail.

The software designer's repertoire of conceptual tools is both quite extensive and also needs some care in its use. As the discussion of knowledge schema indicates, each designer has their own set of models, based both on their own experiences, and experiences obtained from others, through whatever means are most appropriate. The sheer difficulty of undertaking empirical studies in this area has tended to limit our understanding of the effectiveness of our conceptual tools, although this situation is slowly changing. However, regardless of this, it is always important for the reader to be aware that software design does not lend itself to 'silver bullets'—and indeed, conceptual tools are just that: they are aids that assist the designer in performing their own creative task, not a source of solutions in themselves.

Acknowledgements

A review paper such as this draws upon many sources and past discussions with colleagues and collaborators and my thanks to all of them for their help with exploring this complex and fascinating topic.

References

Information sources for this article are:

- Adelson, B. & Soloway, E. (1985). "The Role of Domain Experience in Software Design," *IEEE Trans. Software Eng.*, 11(11), pp. 1351-1360.
- Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I. & Angel, S. (1977). *A Pattern Language*, Oxford University Press. Oxford, England.
- Avison, D.E. & Fitzgerald, G. (1995). *Information Systems Development Methodologies, Techniques & Tools*, 2nd edition, McGraw-Hill, New York.
- Beck, K. (2004). *Extreme Programming Explained: Embrace Change*, 2nd edition, Addison-Wesley, Reading, MA.
- Boehm, B.W. (1988). "A spiral model of software development and enhancement," *IEEE Computer* 21(5), pp. 61–72.
- Brooks, F.P. Jr. (1987). "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer*, Apr, pp. 10-19.
- Brown, A.W. & Short, K. (1997). "On components and objects: The foundations of component-based development," in *Proceedings of 5th International Symposium on Assessment of Software Tools and Technologies*, IEEE Computer Society Press, pp. 112-121.
- Budgen, D. (2003). *Software Design*, 2nd edition, Pearson Addison-Wesley, Reading, MA.
- Budgen, D., Burn, A.J., Brereton, O.P., Kitchenham, B.A. & Pretorius, R. (2011). "Empirical evidence about the UML: A Systematic Literature Review," to appear in *Software: Practice & Experience*.
- Buschmann, F., Meunier, R., Rohnert H., Sommerlad, P. & Stal, M. (1996). *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley.
- Curtis, B., Krasner, H. & Iscoe, N. (1988). "A field study of the software design process for large systems," *Comm. ACM*, 31(11), pp. 1268-1287.
- Détienne, F. (2002). *Software Design—Cognitive Aspects*, Springer Practitioner Series.
- Dranidis, D. (2007). "Evaluation of Student UML: An Educational Tool for consistent modelling with UML," in *Proceedings of the Informatics Europe II Conference*, pp. 248-256.
- Fenton, N.E. & Pfleeger, S. L. (1997). *Software Metrics: A Rigorous & Practical Approach*, 2nd edition, PWS Publishing Company, Boston, MA.
- Freeman, E., Sierra, K & Bates, B. (2004). *Head First Design Patterns*, O'Reilly.
- Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995). *Design Patterns—Elements of Reusable Object-Oriented Software*, Addison-Wesley Reading, MA,
- Garlan, D. & Perry, D.E. (1996). "Introduction to the special issue on software architecture," *IEEE Trans. on Software. Eng.*, 21(4), pp. 269-274.

- Guindon, R. & Curtis, B. (1988). "Control of cognitive processes during software design: what tools are needed," in *Proceedings of CHI'88*, ACM Press, pp. 263-268.
- Guindon, R. (1990). "Knowledge exploited by experts during software system design," *Int. J. Man-Machine Studies*, 33, pp. 279-304.
- Hannay, J. E., Dybå, T., Arisholm, E. & Sjøberg, D. I. K. (2009). "The effectiveness of pair programming: A meta-analysis," *Information & Software Technology* 51, pp. 1110–1122.
- Harel, D. (1987). "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, 8, 231-274.
- Hayes-Roth, B. & Hayes-Roth, F. (1979). "A Cognitive Model of Planning," *Cognitive Science*, 3, pp. 275-310.
- Heineman, G.T. & Councill, W.T. (2001). *Component-Based Software Engineering: Putting the Pieces Together*, Addison-Wesley, Reading, MA.
- Jacobson, I., Booch, G. & Rumbaugh, J. (1999). *The Unified Software Development Process*, Addison-Wesley, Reading, MA.
- Krafzig, D., Banke, K. & Slama, D. (2004). *Enterprise SOA: Service-Oriented Architecture Best Practices*, Prentice-Hall. Upper Saddle River, NJ
- Kruchten, P.B. (1994). "The 4+1 view model of architecture," *IEEE Software*, 12(6), 42-50.
- Moody, D.L. (2009). "The 'physics' of notations: Toward a scientific basis for constructing visual notations in software engineering," *IEEE Trans. on Software. Eng.*, 35(6), pp. 756-779.
- Page-Jones, M. (1988). *The Practical Guide to Structured Systems Design*, 2nd edition, Prentice-Hall International. Upper Saddle River, NJ
- Parnas, D.L. & Weiss, D.M. (1987). "Active Design Reviews: Principles and Practices," *J. Systems & Software*, 7, pp. 259-265.
- Reeves, A.C., Marashi, M. & Budgen, D. (1995). "A software design framework or how to support *real* designers," *Software Eng. Journal*, 10(4), pp. 141-155.
- Rittel, H.J. & Webber, M.M. (1984). "Planning Problems are Wicked Problems," in N. Cross, ed. *Developments in Design Methodology*, Wiley, pp. 135-144.
- Peter Rumbaugh, J., Jacobson, I. & Booch, G. (1999). *The Unified Modeling Language Reference Manual*, Addison-Wesley, Reading, MA.
- Schwaber, K. & Beedle, M. (2002). *Agile software development with Scrum*, Prentice-Hall. Upper Saddle River, NJ
- Shaw, M. & Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, Upper Saddle River, NJ
- Sommerville, I. (2007). *Software Engineering*, 8th edition, Addison-Wesley, Reading, MA.

- Szyperski, C. (1998). *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, Reading, MA.
- Taylor, R.N., Medvidovic, N. & Dashofy, E.M. (2010). *Software Architecture: Foundations, Theory and Practice*, Wiley & Sons, Hoboken, NJ.
- Truex, D., Baskerville, R. & Klein, H. (1999). "Growing systems in emergent organizations," *Comm. ACM,* 42(8), pp. 117–123.
- Wendorff, P. (2001). Peter Wendorff, "Assessment of design patterns during software reengineering: Lessons learned from a large commercial project," in *Proc. of Fifth Conference on Software Maintenance and Reengineering CSMR'01*, IEEE Computer Society Press, Los Alamitos, CA, pp. 77-84.
- Wieringa, R. (1998). "A survey of structured and object-oriented software specification methods and techniques," *ACM Computing Surveys*, 30(4), pp. 459-527.
- Williams, L. & Cockburn, A. (2003). "Agile software development: It's about feedback and change," *IEEE Computer*, 36(6), pp. 39–43.
- Zhang, C. & Budgen, D. (2010). "A Survey of Experience about Design Patterns," Submitted for publication.

Chapter 2.1b

Model-Based Software Design for Concurrent and Real-Time Systems

Hassan Gomaa
Department of Computer Science
George Mason University
Fairfax, Virginia 22030, USA

Abstract

When designing concurrent and real-time systems, it is essential to blend object-oriented concepts with the concepts of concurrent processing. This paper describes a model-based software design method for designing concurrent and real-time systems, which integrates object-oriented and concurrent processing concepts and uses the UML notation.

***Keywords**: real-time systems, UML, concurrency, real-time software, design method, software product lines, software modeling.*

1. Introduction

In model-based software design and development, software modeling is used as an essential part of the software development process. Models are built and analyzed prior to the implementation of the system, and are used to direct the subsequent implementation. A better understanding of a system can be obtained by considering the multiple views [Gomaa 2004, Gomaa 2006], such as requirements models, static models, and dynamic models of the system. A graphical modeling language such as UML helps in developing, understanding and communicating the different views.

Because real-time systems are reactive systems, control decisions are often state dependent, hence the importance of finite state machines in the design of these systems. Real-time systems typically need to process concurrent inputs from many sources, hence the importance of concurrent software design. They have real-time throughput and/or response time requirements, so there is a need to analyze the performance of real-time designs. Furthermore, there is a need to integrate real-time technology with modern software engineering concepts and methods.

This paper provides an overview of designing real-time embedded software systems. It starts by providing an overview of concurrent processing concepts in Section 2. In section 3, run-time support for concurrent and real-time systems is briefly discussed. Section 4 presents an overview of concurrent and real-time design methods. With this background, an overview of a model-based software design method for distributed and real-time embedded systems is given in Section 5. The COMET method [Gomaa 2000, Gomaa 2011] integrates object-oriented and concurrent processing concepts, and uses the Unified Modeling Language (UML) notation. Section 6 describes software architectural patterns for real-time control. Section 7 describes the performance analysis of real-time software designs. Section 8 describes the design of real-time embedded software product lines.

2. Concurrent Processing Concepts

A characteristic of all real-time embedded systems is that of concurrent processing; that is, many activities occurring simultaneously whereby, frequently, the order of incoming events is not predictable. Consequently, as real-time embedded systems deal with several concurrent activities, it is highly desirable for a real-time embedded system to be structured into concurrent tasks (also known as concurrent processes or threads). This section describes the concepts of the concurrent task, and the communication and synchronization between co-operating tasks. For more information, refer to [Bacon 2003, Magee and Kramer 2006, Silberschatz and Galvin 2008, Tanenbaum 2008].

A concurrent task (also known as concurrent process) represents the execution of a sequential program or sequential component of a concurrent program. A concurrent system consists of several tasks executing in parallel. Each task deals with one sequential thread of execution. Concurrency in a software system is obtained by having multiple asynchronous tasks, running at different speeds. From time to time, the tasks need to communicate and synchronize their operations with each other. The concurrent tasking concept has been applied extensively in the design of operating systems, real-time systems, interactive systems, distributed systems, parallel systems, and in simulation applications [Bacon 2003].

3. Run-Time Support for Concurrent Tasks

Runtime support for concurrent processing may be provided by:

- **Kernel of an operating system.** This has the functionality to provide services for concurrent processing. In some modern operating systems, a micro-kernel provides minimal functionality to support concurrent processing, with most services provided by system level tasks.
- **Runtime support system** for a concurrent language.
- **Threads package.** Provides services for managing threads (lightweight processes) within heavyweight processes.

For more information, refer to [Gomaa 2000].

4. Survey of Design Methods for Concurrent and Real-Time Systems

For the design of concurrent and real-time systems, a major contribution came in the late 1970s with the introduction of the MASCOT notation [Simpson 1979], and later the MASCOT design method [Simpson 1986]. Based on a data flow approach, MASCOT formalized the way tasks communicate with each other, via either channels for message communication or pools (information hiding modules that encapsulate shared data structures).

The 1980s saw a general maturation of software design methods, during which time several system design methods were introduced. Parnas's work with the Naval Research Lab, in which he explored the use of information hiding in large-scale software design, led to the development of the Naval Research Lab (NRL) Software Cost Reduction Method [Parnas, Clements, and Weiss 1984]. Work on applying Structured Analysis and Structured Design to concurrent and real-time systems led to the development of Real-Time Structured Analysis and Design (RTSAD) [Ward 1985, Hatley 1988] and the *Design Approach for Real-Time Systems* (DARTS) [Gomaa 1984] methods.

Another software development method to emerge in the early 1980s was Jackson System Development (JSD) [Jackson 1983]. JSD was one of the first methods to advocate that the design should model reality first and, in this respect, predated the object-oriented analysis methods. The system is considered a simulation of the real world and is designed as a network of concurrent tasks, where each real-world entity is modeled by means of a concurrent task. JSD also defied the then-conventional thinking of top-down design by advocating a scenario-driven behavioral approach to software design. This approach was a precursor of object interaction modeling, an essential aspect of modern object-oriented development.

The early object-oriented analysis and design methods emphasized the structural aspects of software development through information hiding and inheritance but neglected the dynamic aspects, and hence were less useful for real-time design. A major contribution by the Object Modeling Technique [Rumbaugh et al., 1991] was to clearly demonstrate that dynamic modeling was equally important. In addition to introducing the static modeling notation for the object diagrams, OMT showed how dynamic modeling could be performed with statecharts (hierarchical state transition diagrams originally conceived by Harel [1996, 1998] for showing behavior of active objects, and with sequence diagrams to show the sequence of interactions between objects.

The CODARTS (Concurrent Design Approach for Real-Time Systems) method [Gomaa 1993] built on the strengths of earlier concurrent design, real-time design, and early object-oriented design methods. These included Parnas's NRL Method, Booch's Object-Oriented Design [Booch 1994], JSD, and the DARTS method by emphasizing both information hiding module structuring and task structuring. In CODARTS, concurrency and timing issues are considered during task design while information hiding issues are considered during module design.

Octopus [Awad, Kuusela, and Ziegler 1996] is a real-time design method based on use cases, static modeling, object interactions, and statecharts. By combining concepts from Jacobson's use cases with Rumbaugh's static modeling and statecharts, Octopus anticipated the merging of the notations that is now the UML. For real-time design, Octopus places particular emphasis on interfacing to external devices and on concurrent task structuring.

ROOM (Real-Time Object-Oriented Modeling) [Selic, Gullekson, and Ward 1994], is a real-time design method that is closely tied in with a CASE (Computer Assisted Software Engineering) tool called ObjecTime. ROOM is based around actors, which are active objects that are modeled using a variation on statecharts called ROOMcharts. A ROOM model that has been specified in sufficient detail may be executed. Thus, a ROOM model is operational and may be used as an early prototype of the system.

Buhr [1996] introduced an interesting concept called the use case map (based on the use case concept) to address the issue of dynamic modeling of large-scale systems. Use case maps consider the sequence of interactions between objects (or aggregate objects in the form of subsystems) at a coarser grained level of detail than do communication diagrams.

For UML-based real-time software development, Douglass [1999, 2004] has provided a comprehensive description of how UML can be applied to real-time systems. The 2004 book describes applying the UML notation to the development of real-time systems. The 1999 book is a detailed compendium covering a wide range of topics in real-time system development, includ-

ing safety-critical systems, interaction with real-time operating systems, real-time scheduling, behavioral patterns, real-time frameworks, debugging, and testing.

5. A Model-Based Software Design Method for Concurrent and Real-Time Embedded Systems

Most books on object-oriented analysis and design only address the design of sequential systems or omit the important design issues that need to be addressed when designing real-time and distributed applications [Gomaa 2000, Bacon 2003].

It is essential to blend object-oriented concepts with the concepts of concurrent processing in order to successfully design these applications. This paper describes some of the key aspects of the COMET model-based software design method for real-time embedded and distributed systems. COMET integrates object-oriented and concurrent processing concepts and uses the Unified Modeling Language (UML) notation (Rumbaugh 2005). It also describes the decisions made regarding how to use the UML notation to address the design of concurrent, distributed, and real-time embedded systems. Examples are given from a Pump Monitoring and Control System, which is depicted using the UML 2 notation.

5.1 The COMET Method

COMET is a Concurrent Object Modeling and Architectural Design Method for the development of concurrent applications, in particular distributed and real-time embedded applications [Gomaa 2000, Gomaa 2011]. As the UML is now the standardized notation for describing object-oriented models [Booch et al. 2005, Rumbaugh et al. 2004, Jacobson et al. 2000], the COMET method uses the UML notation throughout.

The COMET Object-Oriented Software Life Cycle is highly iterative. In the Requirements Modeling phase, a use case model is developed in which the functional requirements of the system are defined in terms of actors and use cases.

In the Analysis Modeling phase, static and dynamic models of the system are developed. The static model defines the structural relationships among problem domain classes. Object structuring criteria are used to determine the objects to be considered for the analysis model. A dynamic model is then developed in which the use cases from the requirements model are refined to show the objects that participate in each use case and how they interact with each other. In the dynamic model, state dependent objects are defined using statecharts.

In the Design Modeling phase, an Architectural Design Model is developed. Subsystem structuring criteria are provided to design the overall software architecture. For distributed applications, a component-based development approach is taken, in which each subsystem is designed as a distributed self-contained component. The emphasis is on the division of responsibility between clients and servers, including issues concerning the centralization vs. distribution of data and control, and the design of message communication interfaces, including synchronous, asynchronous, brokered, and group communication. Each concurrent subsystem is then designed, in terms of active objects (tasks) and passive objects. Task communication and synchronization interfaces are defined. The performance of real-time designs is estimated using an approach based on rate monotonic analysis [SEI 1993].

Distinguishing features of the COMET method are the emphasis on:

- Structuring criteria to assist the designer at different stages of the analysis and design process: subsystems, objects, and concurrent tasks.
- Dynamic modeling, both object communication diagrams and statecharts, describing in detail how object communication diagrams and statecharts relate to each other.
- Distributed application design, addressing the design of configurable distributed components and inter-component message communication interfaces.
- Concurrent design, addressing in detail task structuring and the design of task interfaces.
- Performance analysis of real-time designs using real-time scheduling.

COMET emphasizes the use of structuring criteria at different stages in the analysis and design process. Object structuring criteria are used to help determine the objects in the system, subsystem structuring criteria are used to help determine the subsystems, and concurrent task structuring is used to help determine the tasks (active objects) in the system. UML stereotypes are used throughout to clearly show the use of the structuring criteria.

The UML Notation supports Requirements, Analysis, and Design concepts. The COMET method separates requirements, analysis, and design activities. Requirements modeling address defining the functional requirements of the system. COMET differentiates analysis from design as follows: analysis is breaking down or decomposing the problem so that it is better understood, while design is synthesizing or composing (putting together) the solution. These activities are now described in more detail.

5.2 Requirements Modeling with UML

In the Requirements Model, the system is considered as a black box. The Use Case Model is developed in which the functional requirements of the system are defined in terms of use cases and actors. This section describes the use of actors in real-time applications.

There are several variations on how actors are modeled [Jacobson 1992, Booch 2007, Fowler 2004, Gomaa 2011]. An actor is very often a human user. In many information systems, humans are the only actors. It is also possible in information systems for an actor to be an external system. In real-time and distributed applications, an actor can also be an external I/O device or a timer. External I/O devices and timer actors are particularly prevalent in real-time embedded systems, where the system interacts with the external environment through sensors and actuators.

A human actor may use various I/O devices to physically interact with the system. In such cases, the human is the actor and the I/O devices are not actors. In some cases, however, it is possible for an actor to be an I/O device. This can happen when a use case does not involve a human, as often occur in real-time applications.

An actor can also be a timer that periodically sends timer events to the system. Periodic use cases are needed when certain information needs to be output by the system on a regular basis. This is particularly important in real-time systems, although it can also be useful in information systems. Although some methodologists consider timers to be internal to the system, it is more useful in real-time application design to consider timers as logically external to the system and to treat them as primary actors that initiate actions in the system.

An example of a use case model from the Pump Monitoring and Control System is given in Figure 1, in which there are two use cases, Control Pump and View Pump Status. There are five

actors, three representing the three external sensors, one clock actor, and an external user actor – the Operator.

5.3 Analysis Modeling with UML

This section describes some of the interesting aspects of COMET for analysis modeling. In particular, this section describes static modeling of the system context, stereotypes to represent object structuring decisions made by the analyst, and consistency checking between multiple views of a dynamic model.

5.3.1 Static Modeling

For real-time applications, it is particularly important to understand the interface between the system and the external environment, which is referred to as the *system context*. In Structured Analysis [Yourdon 1989], the system context is shown on a *system context diagram*. The UML notation does not explicitly support a system context diagram. However, the system context may be depicted using either a static model or a communication model [Douglass 1999]. A *system context class diagram* provides a more detailed view of the system boundary than a use case diagram.

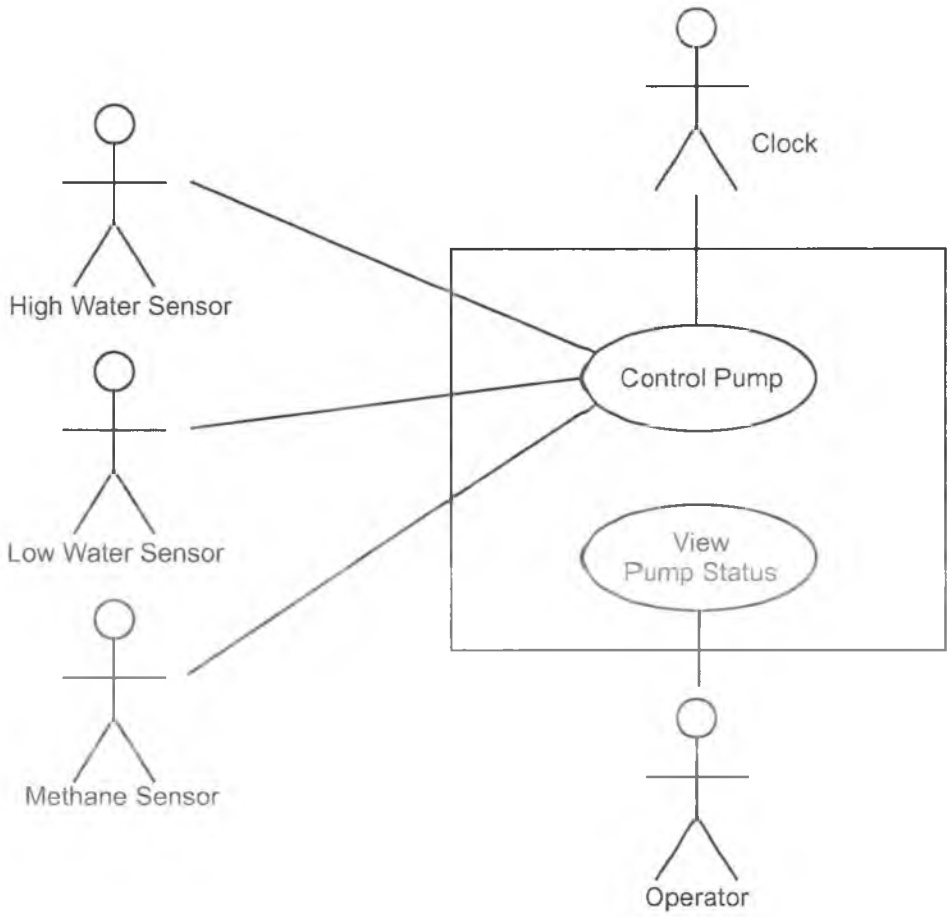


Figure 1: Use Case Model for Pump Monitoring and Control Systems

Using the UML notation for the static model, the system context is depicted showing the system as an aggregate class with the stereotype «software system», and the external environment is depicted as external classes to which the system must interface. External classes are categorized using stereotypes (see description in Section 5.4.2). An external class can be an «external input device», an «external output device», an «external I/O device», an «external user», an «external system», or an «external timer». For a real-time system, it is desirable to identify low-level external classes that correspond to the physical I/O devices to which the system must interface. These external classes are depicted with the stereotype «external I/O device».

An example of a system context class diagram from the Pump Monitoring and Control System is given in Figure 2. There are three external input device classes, namely the three sensors, one external output device class, the pump engine, one external timer class, and one external user class.

During the analysis modeling phase, static modeling is also used for modeling data-intensive classes [Rumbaugh 1991].

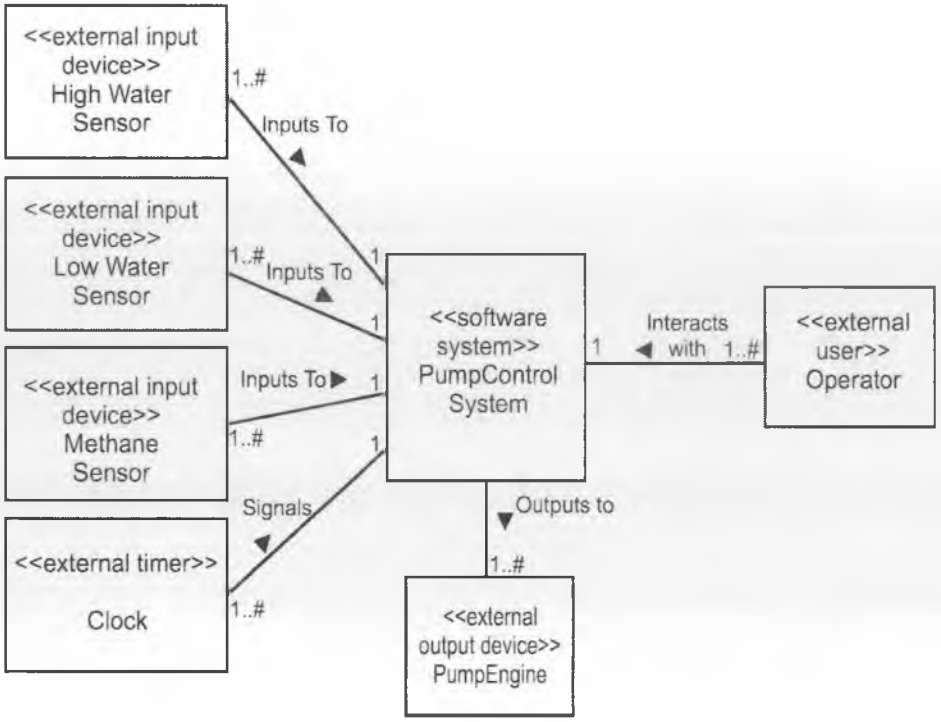


Figure 2: Pump Monitoring and Control System Class Context Diagram

5.3.2 Object Structuring

Object structuring criteria are provided to assist the designer in structuring a system into objects. Several object-based and object-oriented analysis methods provide criteria for determining objects in the problem domain [Booch 1994, Coad 1991, Gomaa 1993, Jacobson 1992, Parnas 1984, Shlaer, and Mellor 1988]. The COMET object structuring criteria build on these methods.

In object structuring, the goal is to categorize objects in order to group together those with similar characteristics. Whereas classification based on inheritance is an objective of object-oriented modeling, it is essentially tactical in nature. Categorization, however, is a strategic classification. The decision to organize classes into certain groups is made because most software systems have these kinds of classes, and categorizing classes in this way helps us understand the system we are to develop.

UML stereotypes are used to distinguish among the different kinds of application classes. A *stereotype* is a subclass of an existing modeling element, in this case an application class, which is used to represent a usage distinction, in this case the kind of class. A stereotype is depicted in guillemets, e.g., «control». An instance of a stereotype class is a stereotype object, which can also be shown in guillemets. Thus an application class can be categorized as an «entity» class, which is a persistent class that stores data, a «boundary» class, which interfaces to and communicates with the external environment, a «control» class, which provides the overall coordination for the objects that participate in a use case, or an «application logic» class, which encapsulates algorithms separately from the data being manipulated.

Real-time systems will have many device interface classes to interface with the various sensors and actuators. They will also have complex state-dependent control classes because these systems are highly state dependent.

5.3.3 Dynamic Modeling

For concurrent, distributed, and real-time applications, dynamic modeling is of particular importance. UML does not emphasize consistency checking between multiple views of the various models. Nevertheless, during dynamic modeling, it is important to understand how the finite state machine model, depicted using a statechart [Harel 1988, Harel 1996, Harel 1998] that is executed by a state-dependent control object, relates to the interaction model, which depicts the interaction of this object with other objects.

State Dependent Dynamic Analysis addresses the interaction among objects that participate in state-dependent use cases. A state-dependent use case has a state-dependent control object, which executes a statechart, providing the overall control and sequencing of the use case. The interaction among the objects that participate in the use case is depicted on a communication diagram or sequence diagram.

The statechart needs to be considered in conjunction with the communication diagram. In particular, it is necessary to consider the messages that are received and sent by the control object, which executes the statechart. An input event into the control object on the communication diagram must be consistent with the same event depicted on the statechart. The output event (which causes an action, enable or disable activity) on the statechart must be consistent with the output event shown on the communication diagram.

An example of the communication diagram for the Control Pump use case is given in Figure 3. An example of the statechart for the Pump Control object is shown in Figure 4. In Figure 3, there are two input objects, High Water Sensor Interface and Low Water Sensor Interface, both of which receive inputs from the external input devices. There is one output object, Pump Engine Interface, which outputs to the external output device. There is one state-dependent control object, Pump Control, which executes the statechart in Figure 4. Finally, there is one timer object. Message inputs to the Pump Control object, such as High Water Detected in Figure 3, are

the events that cause state changes on the statechart in Figure 4. Actions in Figure 4, such as Start Pump and Stop Pump, correspond to output messages from the Pump Control object in Figure 4.

5.4 Design Modeling

This section describes some of the interesting aspects of COMET for design modeling. In particular, this section describes the consolidation of communication diagrams to synthesize an initial software design, subsystem structuring using packages, distributed application design, concurrent task design, and the design of connectors using monitors.

5.4.1 The Transition from Analysis to Design

In order to transition from analysis to design, it is necessary to synthesize an initial software design from the analysis carried out so far. In the analysis model, a communication diagram is developed for each use case. The *integrated communication diagram* is a synthesis of all the communication diagrams developed to support the use cases. The consolidation performed at this stage is analogous to the robustness analysis performed in other methods [Jacobson 1992, Rosenberg 1999]. These other methods use the static model for robustness analysis, whereas COMET emphasizes the dynamic model, as this addresses the message communication interfaces, which is crucial in the design of real-time and distributed applications.

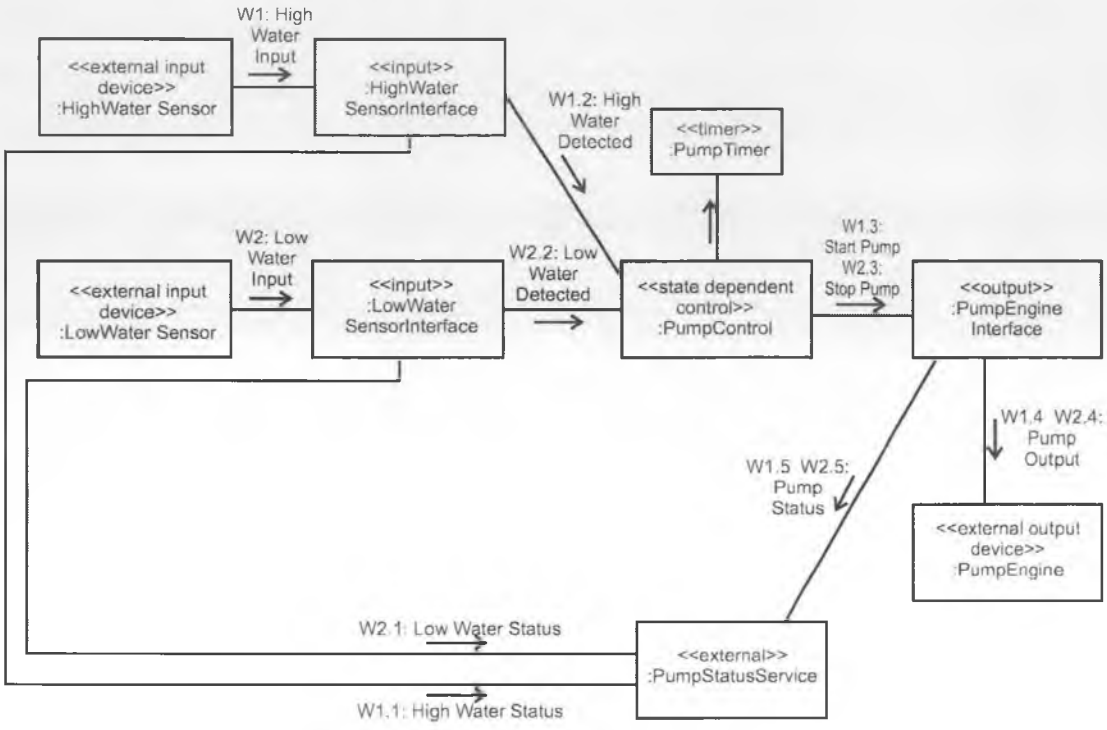


Figure 3: Communication diagram for Control Pump Use Case

5.4.2 Software Architectural Design

During Software Architectural Design, the system is decomposed into subsystems and the interfaces between the subsystems are defined [Shaw 1996, Taylor 2009]. A system is structured into subsystems, which contain objects that are functionally dependent on each other. The goal is to have objects with high coupling among each other in the same subsystem, while objects that are weakly coupled are placed in different subsystems. A subsystem can be considered a composite or aggregate object that contains the simple objects that compose that subsystem.

The integrated communication diagram, which depicts the objects and messages from all the use-case-based communication diagrams, can get very large for a large system and thus, it may not be practical to show all the objects on one diagram. This problem is addressed by developing an integrated communication diagram for each subsystem, and developing a higher level subsystem communication diagram to show the dynamic interactions between subsystems on a *subsystem communication diagram*, which depicts the overall software architecture, as shown in Figure 5. The structure of an individual subsystem is then depicted on an integrated communication diagram, which shows all the objects in the subsystem and their interconnections.

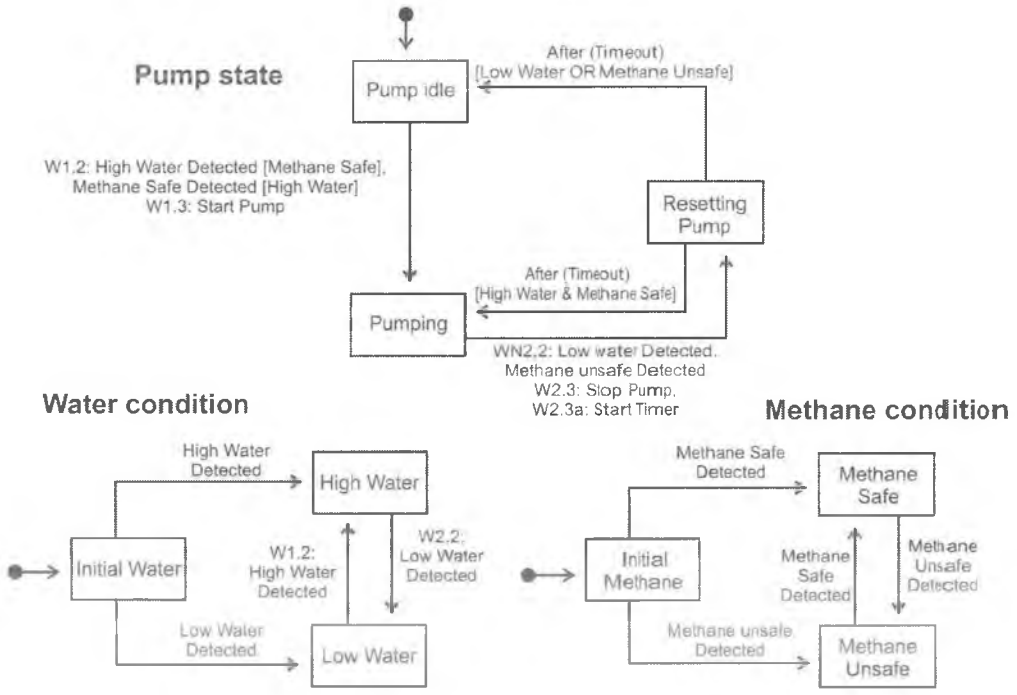


Figure 4: Pump Control Statechart

5.4.3 Concurrent Communication Diagrams

In the UML 2 notation, an active object or task is depicted as a box with two parallel lines on the left and right sides of the object box. An active object has its own thread of control and executes concurrently with other objects. This is in contrast to a passive object, which does not have a thread of control.

A passive object only executes when another object (active or passive) invokes one of its operations. In this paper, we refer to an active object as a task and a passive object as an object. Tasks are depicted on *concurrent communication diagrams*, which depict the concurrency concerns of the system [Douglass 2004, Gomaa 2011]. On a concurrent communication diagram, a task is depicted as a box with thick black lines while a passive object is depicted as a box with thin black lines. In addition, decisions are made about the type of message communication between tasks, asynchronous or synchronous, with or without reply.

5.4.4 Architectural Design of Distributed Real-Time Systems

Distributed real-time systems execute on geographically distributed nodes supported by a local or wide area network. With COMET, a distributed real-time system is structured into distributed subsystems, where a subsystem is designed as a configurable component and corresponds to a logical node. A subsystem component is defined as a collection of concurrent tasks executing on one logical node. As component subsystems potentially reside on different nodes, all communication between component subsystems must be restricted to message communication. Tasks in different subsystems may communicate with each other using several different types of message communication (Figure 5) including asynchronous communication, synchronous communication, client/server communication, group communication, brokered communication, and negotiated communication. The configuration of the distributed real-time system is depicted on a deployment diagram, as shown in Figure 6, which shows the three subsystems depicted as distributed nodes in a distributed configuration.

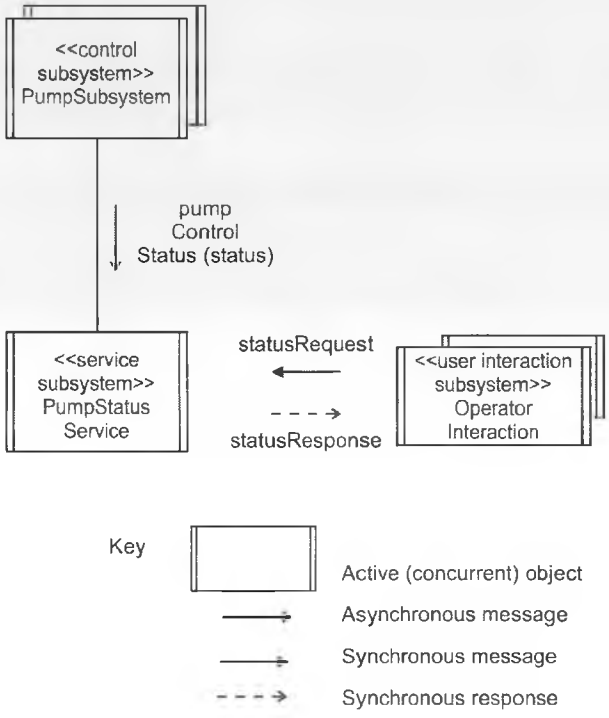


Figure 5: Distributed Software Architecture

5.4.5 Task Structuring

During the task structuring phase, each subsystem is structured into concurrent tasks and the task interfaces are defined. Task structuring criteria are provided to assist in mapping an object-oriented analysis model of the system to a concurrent tasking architecture. Following the approach used for object structuring, stereotypes are used to depict the different kinds of tasks. Each task is depicted with two stereotypes. The first is the object role criterion, determined during object structuring as described in Section 5.4.2. The second is used to depict the type of concurrency. During concurrent task structuring, if an object in the analysis model is determined to be active, it is categorized further to show its concurrent task characteristics. For example, an active «I/O» object is concurrent and is categorized further using a second stereotype as one of the following: an «event driven» task, a «periodic» task, or a «demand driven» task. Stereotypes are also used to depict the kinds of devices to which the concurrent tasks interface.

Thus, an «external input device» is further classified, depending on its characteristics, into an «event driven» external input device or a «passive» external input device. An event-driven I/O task is needed when there is an event-driven (also referred to as interrupt driven) I/O device to which the system has to interface. The event driven I/O task is activated by an interrupt from the event driven device.

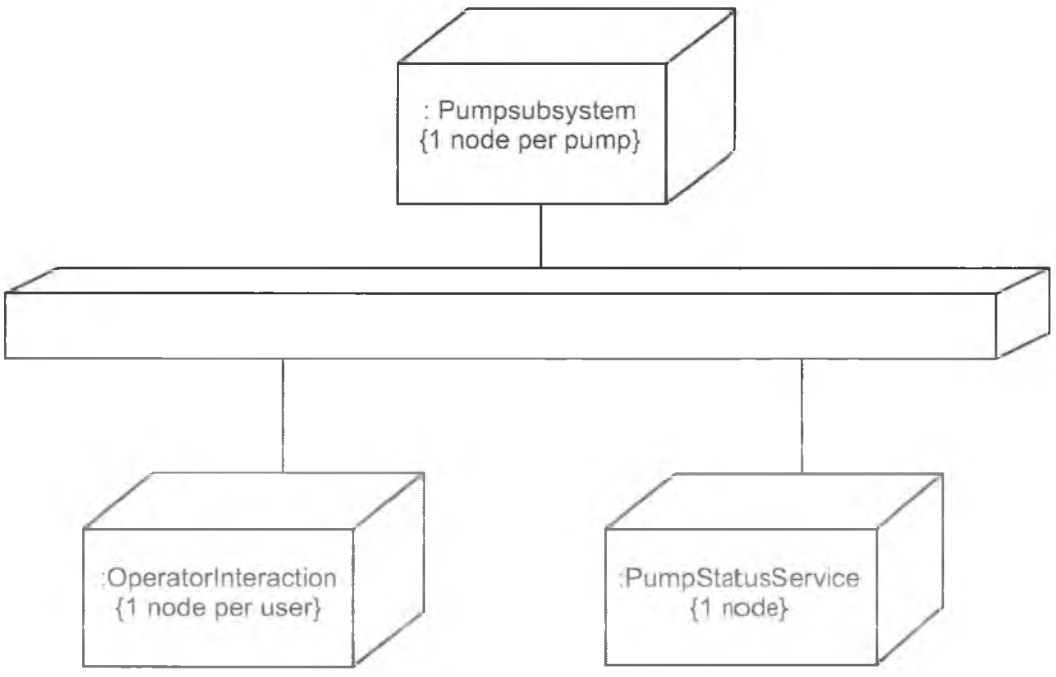


Figure 6: Distributed System Configuration

5.4.6 Detailed Software Design

If a passive class is accessed by more than one task, then the class's operations must synchronize the access to the data it encapsulates. Synchronization is achieved using the mutual exclusion or multiple readers and writers algorithms [Bacon 2003].

Connector classes encapsulate the details of inter-task communication, such as loosely and tightly coupled message communication. Some concurrent programming languages such as Ada and Java provide mechanisms for inter-task communication and synchronization. Neither of these languages supports loosely coupled message communication. In order to provide this capability, it is necessary to design a Message Queue connector class, which encapsulates a message queue and provides operations to access it. A connector is designed using a monitor, which combines the concepts of information hiding and task synchronization [Bacon 2003, Magee & Kramer 2006]. These monitors are used in a single processor or multiprocessor system with shared memory. Connectors may be designed to handle asynchronous message communication, synchronous message communication without reply, and synchronous message communication with reply.

6. Software Architectural Patterns for Real-Time Control

Software architectural patterns [Buschmann 1996] provide the skeleton or template for the overall software architecture or high-level design of an application. Basing the software architecture of a product line on one or more software architectural patterns helps in designing the original architecture, because it is based on a proven architecture and it evolves the architecture.

There are two main categories of software architectural patterns [Gomaa 2011]. Architectural structure patterns address the static structure of the software architecture. Architectural communication patterns address the message communication among distributed components of the software architecture.

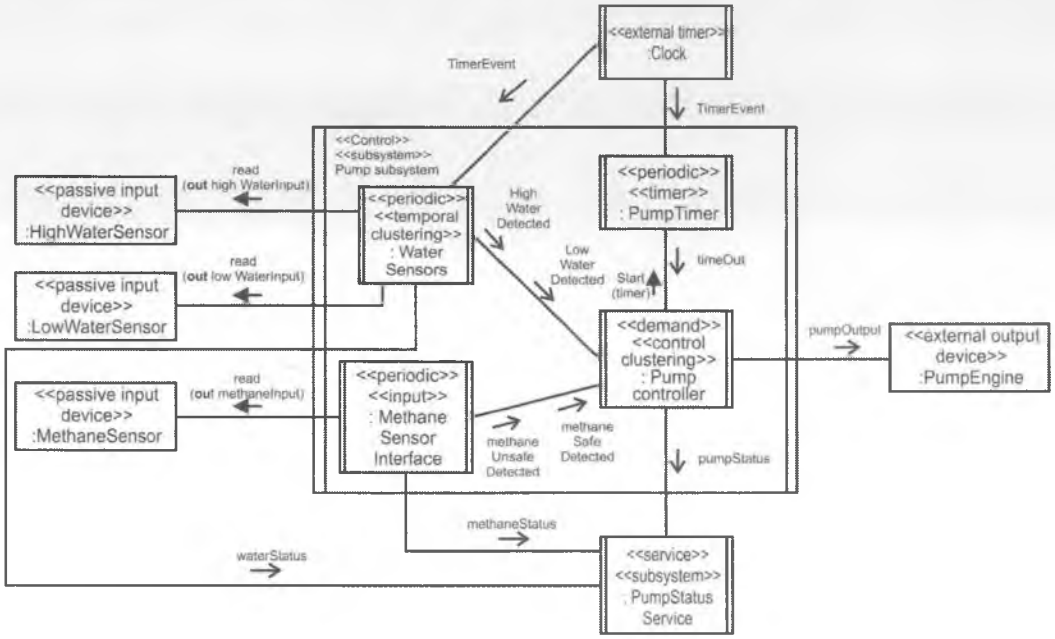


Figure 7: Pump Substation - Task Architecture

Most software systems can be based on well-understood overall software architectures. For example, the client/server software architecture is prevalent in many software applications. The basic client/server architecture has one server and many clients. However, there are also many variations on this theme, such as multiple client/multiple server architectures and brokered client/server architectures.

Many real-time systems provide overall control of the environment by providing either centralized control, decentralized control, or hierarchical control. Each of these control approaches can be modeled using a software architectural pattern. In a centralized control pattern, there is one control component, which executes a Statechart. It receives sensor input from input components and controls the external environment via output components, as shown in Figure 7 for the Pump Controller task. In a centralized control pattern, the control component executes a statechart, which is depicted for the Pump Controller in Figure 4. Another pattern used in the Pump Monitoring and Control System is the client/server pattern, as shown in Figure 5, where the Pump Subsystem is the client and the Pump Status Server is the server.

Architectural communication patterns for real-time systems include asynchronous communication and synchronous communication, both with and without reply. Other possible communication patterns include subscription/notification patterns and broker patterns. In the Pump Monitoring and Control System, both asynchronous and synchronous message communications are used as shown in Figures 5 and 7.

7. Performance Analysis of Real-Time Designs

Performance analysis of software designs is particularly important for real-time systems. The consequences of a real-time system failing to meet a deadline can be catastrophic.

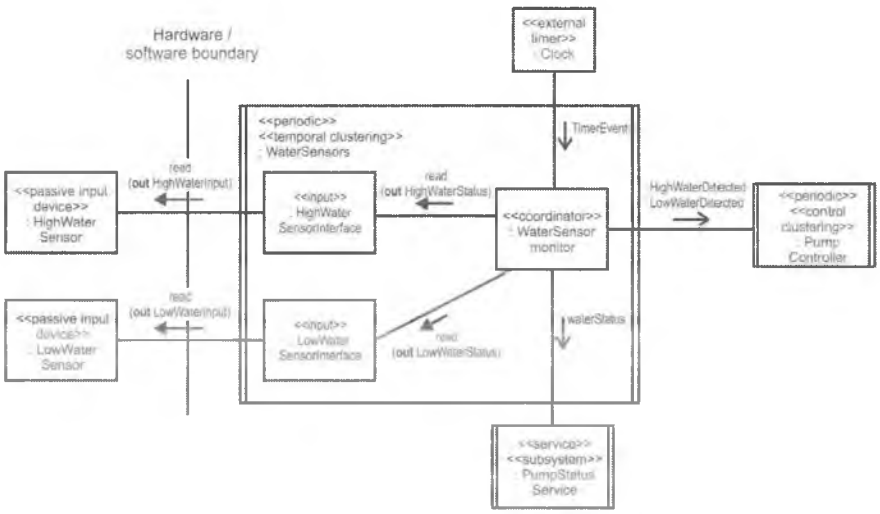


Figure 8: Water Sensors - Temporal Clustering with Nested Input Objects

The quantitative analysis of a real-time system design allows the early detection of potential performance problems. The analysis is for the software design conceptually executing on a given hardware configuration with a given external workload applied to it. Early detection of potential

performance problems allows alternative software designs and hardware configurations to be investigated.

In COMET, performance analysis of software designs is achieved by applying *real-time scheduling* theory. *Real-time scheduling* is an approach that is particularly appropriate for hard real-time systems that have deadlines that must be met [Gomaa 2000, SEI 1993]. With this approach, the real-time design is analyzed to determine whether it can meet its deadlines.

A second approach for analyzing the performance of a design is to use *event sequence analysis* and to integrate this with the *real-time scheduling* theory. Event sequence analysis considers scenarios of task collaborations and annotates them with the timing parameters for each of the tasks participating in each collaboration, in addition to system overhead for inter-object communication and context switching. The equivalent period for the active objects in the collaboration is the minimum inter-arrival time of the external event that initiates the collaboration.

8. Real-Time Embedded Software Product Line Design

A software product line (SPL) consists of a family of software systems that have some common functionality and some variable functionality [Parnas 1979, Clements 2002, Gomaa 2005]. Software product line engineering involves developing the requirements, architecture, and component implementations for a family of systems, from which products (family members) are derived and configured. The problems of developing individual software systems are scaled upwards when developing software product lines because of the increased complexity due to variability management.

A better understanding of a system or product line can be obtained by considering the multiple views, such as requirements models, static models, and dynamic models of the system or product line. A graphical modeling language such as UML helps in developing, understanding and communicating the different views. A key view in the multiple views of a software product line is the feature modeling view. The feature model is crucial for managing variability and product derivation as it describes the product line requirements in terms of commonality and variability, as well as defining the product line dependencies. Furthermore, it is necessary to have a development approach that promotes software evolution, such that original development and subsequent maintenance are both treated using feature-driven evolution.

The Evolutionary Software Product Line Engineering Process [Gomaa 2005] is a highly iterative software process that eliminates the traditional distinction between software development and maintenance. Furthermore, because new software systems are outgrowths of existing ones, the process takes a software product line perspective; as shown in Figure 9:

1. *Product Line (Domain) Engineering*. A product line multiple-view model is developed, which addresses the multiple views of a software product line. The product line multiple-view model, product line architecture, and reusable components are developed and stored in the product line reuse library.
2. *Software Application Engineering*. A software application multiple-view model is an individual product line member derived from the software product line multiple-view model. The user selects the required features for the individual product line member. Given the features, the product line model and architecture are adapted and tailored to derive the application architecture. The architecture determines which of the reusable components are needed for configuring the executable application.

Software product line concepts can also be applied to the design of embedded real-time software. Thus the COMET design method has been extended to the PLUS method (Product Line UML-Based Software Engineering) for designing embedded real-time software product lines as described in [Gomaa 2005].

9. Conclusions

This paper has described concepts and methods for the design of concurrent and real-time software systems. It is essential to blend object-oriented concepts with the concepts of concurrent processing. This paper has given an overview of the COMET model-based software design method for designing concurrent and real-time systems, which integrates object-oriented and concurrent processing concepts and uses the UML notation.

For software-intensive systems, in which the software is one component of a larger hardware/software system, systems modeling can be carried out before software modeling. A dialect of UML called SysML is a general purpose modeling language for systems engineering applications [Friedenthal et al. 2009]. More information on UML modeling for real-time and embedded systems is given in MARTE, the UML profile for Modeling and Analysis of Real-Time and Embedded Systems [Espinoza et al. 2009].

With the proliferation of low-cost workstations and personal computers operating in a networked environment, the interest in designing concurrent systems, particularly real-time and distributed systems, is growing rapidly. Furthermore, with the growing need for reusable designs, design methods for software product lines [Gomaa 2005] and service-oriented architectures [Gomaa 2011] are likely to be of increasing importance for future real-time embedded software systems.

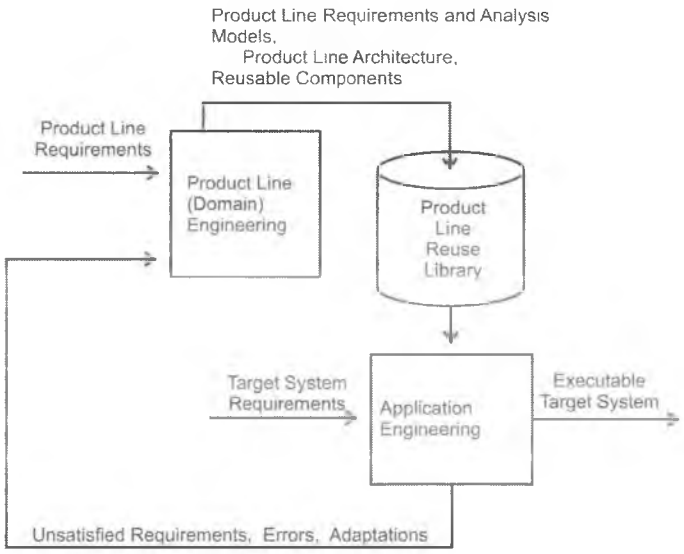


Figure 9: Process Model for Software Product Line Engineering

References

References for this chapter are:

- M. Awad, J. Kuusela, and J. Ziegler, "Object-Oriented Technology for Real-Time Systems," Prentice-Hall, Upper Saddle River, NJ, 1996.
- J. Bacon, *Concurrent Systems*, Third Edition, Addison-Wesley, Reading, MA, 2003.
- G. Booch, R. A. Maksimchuk, M. W. Engel, et al., "Object-Oriented Analysis and Design with Applications," 3rd ed., Addison-Wesley, Boston, 2007.
- G. Booch, J. Rumbaugh, I. Jacobson, "The Unified Modeling Language User Guide," 2nd ed., Addison-Wesley, Reading, MA. 2005.
- R. J. A. Buhr and R. S. Casselman, "Use Case Maps for Object-Oriented Systems," Prentice-Hall, Upper Saddle River, NJ, 1996.
- P. Clements and L. Northrop, "Software Product Lines: Practices and Patterns," Addison-Wesley, Reading, MA, 2002.
- P. Coad and E. Yourdon, "Object-Oriented Analysis," Prentice-Hall, Upper Saddle River, NJ, 1991.
- B. P. Douglass, "Doing Hard Time: UML, Objects, Frameworks, and Patterns in Real-Time Software Development," Addison-Wesley, Reading, MA, 1999.
- B. P. Douglass, "Real-Time UML," Third Edition, Addison-Wesley, Reading, MA, 2004.
- H. Espinoza, D. Cancila, B. Selic and S. Gérard, "Challenges in Combining SysML and MARTE for Model-Based Design of Embedded Systems." Lecture Notes in *Computer Science*, vol. 5562, pp. 98-113. Springer, Berlin, 2009.
- M. Fowler and K. Scott, "UML Distilled," Third Edition, Addison-Wesley, Reading, MA, 2004.
- S. Friedenthal, A. Moore, and R. Steiner, "A Practical Guide to SysML: The Systems Modeling Language," Morgan Kaufmann, Burlington, MA 2009.
- H. Gomaa, "A Software Design Method for Real Time Systems," *Communications ACM*, Vol. 27, No. 9, September 1984.
- H. Gomaa, "Software Design Methods for Concurrent and Real-Time Systems," Addison-Wesley, Reading, MA, 1993.
- H. Gomaa, "Designing Concurrent, Distributed, and Real-Time Applications with UML," Addison-Wesley, Reading, MA, 2000.
- H. Gomaa, "Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures," Addison-Wesley, Reading, MA, 2005.
- H. Gomaa, "A Software Modeling Odyssey: Designing Evolutionary Architecture-centric Real-Time Systems and Product Lines," Keynote Paper, *Proc. ACM/IEEE 9th International Conference on Model-Driven Engineering, Languages and Systems*, Springer Verlag LNCS 4199, Pages 1-15, Genova, Italy, October 2006.

- H. Gomaa, "Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures," Cambridge University Press, New York, 2011.
- H. Gomaa and M.E. Shin, "A Multiple-View Meta-Modeling Approach for Variability Management in Software Product Lines," *Proc. International Conference on Software Reuse*, Madrid, Spain, Springer LNCS 3107, July 2004.
- D. Harel, "On Visual Formalisms." *CACM 31*, 5 (May 1988), 514-530.
- D. Harel and E. Gary, "Executable Object Modeling with Statecharts," *Proc. 18th International Conference on Software Engineering*, Berlin. March 1996.
- D. Harel and M. Politi, "Modeling Reactive Systems with Statecharts," McGraw-Hill, 1998.
- D. Hatley and I. Pirbhai, "Strategies for Real Time System Specification," Dorset House, New York, 1988.
- M. Jackson, "System Development," Prentice-Hall, Upper Saddle River, 1983.
- I. Jacobson, "Object-Oriented Software Engineering," Addison-Wesley. Reading, MA, 1992.
- I. Jacobson, G. Booch, J. Rumbaugh, "The Unified Software Development Process," Reading, MA, Addison-Wesley, Reading, MA, 2.
- J. Magee and J. Kramer, "Concurrency, State Models & Java Programs," Second ed., John Wiley & Sons, 2006.
- D. Parnas, "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering*, March 1979.
- D. Parnas, P. Clements and D. Weiss, "The Modular Structure of Complex Systems," *Proc. Seventh IEEE International Conference on Software Engineering*, Orlando, Florida, March 1984.
- D. Rosenberg and K. Scott, "Use Case Driven Object Modeling with UML," Addison-Wesley, Reading, MA, 1999.
- J. Rumbaugh, J. Blaha, W. Premerlani, F. Eddy, W. Lorenson, "Object-Oriented Modeling and Design," Prentice-Hall, Upper Saddle River, NJ, 1991.
- J. Rumbaugh, G. Booch, I. Jacobson, "The Unified Modeling Language Reference Manual," 2nd ed., Addison-Wesley, Reading, MA, 2005.
- B. Selic, G. Gullekson, and P. Ward, "Real-Time Object-Oriented Modeling," Wiley & Sons, Hoboken, NJ, 1994.
- M. Shaw and D. Garlan, *Software Architecture, Perspectives on an Emerging Discipline*, Prentice-Hall, Upper Saddle River, NJ, 1996.
- S. Shlaer and S. Mellor, "Object-Oriented Systems Analysis," Prentice-Hall, Upper Saddle River, NJ, 1988.
- A. Silberschatz, P. Galvin, and G. Gagne, "Operating System Concepts," 8th ed., Addison-Wesley, Reading, MA, 2008.

- H. Simpson and K. Jackson, "Process Synchronization in MASCOT," *The Computer Journal*, vol.17, no. 4, 1979.
- H. Simpson, "The MASCOT Method," *IEE/BCS Software Engineering Journal*, 1(3), 1986, 103-120.
- SEI - Carnegie Mellon University Software Engineering Institute, "A Practitioner's Handbook for Real-Time Analysis - Guide to Rate Monotonic Analysis for Real-Time Systems," Kluwer Academic Publishers, Boston, 1993.
- A. S. Tanenbaum, "Modern Operating Systems," 3rd ed., PrenticeHall, Upper Saddle River, NJ, 2008.
- R. N. Taylor, N. Medvidovic, E. M. Dashofy, "Software Architecture, Foundations, Theory, and Practice," New York, Wiley & Sons, Hoboken, NJ 2009.
- P. Ward and S. Mellor, "Structured Development for Real-Time Systems," Vols. 1, 2 & 3, Yourdon Press, New York, 1985.
- E. Yourdon, "Modern Structured Analysis," Prentice Hall, Upper Saddle River, NJ, 1989.

Chapter 2.2

Essentials of Software Design

Richard Hall Thayer and Merlin Dorfman

This is the second chapter of a textbook to aid individual software engineers in a greater understanding of the IEEE SWEBOK [2013] and a guide book to aid software engineers in passing the IEEE CSDP and CSDA certification exams.

This module provides an introduction to the principles and concepts relevant to software design. It examines the role and context of the design activity as a form of the problem-solving process, describes how this is supported by current design methods, and considers the strategies, strengths, limitations, and main domains of application of these methods.

This list of exam specifications is reported to be the same list that the exam writers used to write the exam questions. Therefore it is the best source of help for the exam takers. Chapter 2 covers the following CSDP exam software design module [Software Exam Specification, Version 2, 18 March 2009]:

1. Software design fundamentals (general design concepts; the context of software design; the software design process; enabling techniques)
2. Key issues in software design (concurrency; control and handling of events; distribution of components; error and exception handling and fault tolerance; interaction and presentation; data persistence)
3. Software structure and architecture (architectural structures and viewpoints; architectural styles; design patterns; families of programs and frameworks; hardware issues in software architecture)
4. Human computer interface design (general HCI design principles; use of modes, navigation; coding techniques and visual design [color, icons, fonts, and so on]; response time and feedback; design modalities [menu-driven, forms, question-answering, and so on]; localization and internationalization; human computer interface design methods; multimedia [I/O, voice, natural language, web-page, sound]; metaphors and conceptual models; psychology of HCI)
5. Software design quality analysis and evaluation (quality attributes; quality analysis and evaluation techniques; measures)
6. Software design notations (structural descriptions [static view]; behavioral descriptions [dynamic view])
7. Software design strategies and methods (general strategies; function-oriented [structured] design; object-oriented design; data-structure-centered design; component-based design [CBD]; other methods; software design tools)

Software design is a process of defining the architecture, components, interfaces, and other characteristics of a system or component and planning for a software solution. After the purpose and specifications of software are determined, software developers will design or employ designers to develop a plan for a solution.

Viewed as a process, software design is the software engineering life-cycle activity in which software requirements are analyzed in order to produce a description of the software's internal structure that will serve as the basis for its construction. More precisely, a software design (the result) must describe the software components and the interfaces between those components. It must also describe the components at a level of detail that enable their construction [SWEBOK 2004].

2.1 Software Design Fundamentals

The concepts, notions, and terminology introduced here form an underlying basis for understanding the role and scope of software design.

2.1.1 General design concepts. Software is not the only field where design is involved. In the general sense, we can view design as a form of problem-solving. For example, the concept of a *wicked problem* is interesting in terms of understanding the limits of design. A number of other notions and concepts are also of interest in understanding design in its general sense: goals, constraints, alternatives, representations, and solutions [SWEBOK 2004].

A *wicked problem* is a phrase originally used in social planning to describe a problem that is difficult or impossible to solve because of incomplete, contradictory, and changing requirements that are often difficult to recognize. Moreover, because of complex interdependencies, the effort to solve one aspect of a wicked problem may reveal/create other problems [http://en.wikipedia .org /wiki/Wicked_problem].

2.1.2 Context of software design. To understand the role of software design, it is important to understand the context in which it fits, the software engineering life cycle. Thus, it is important to understand the major characteristics of software requirements analysis versus software design versus software construction versus software testing [SWEBOK 2004].

- ***Software requirements*** are a sub-field of software engineering that deal with the elicitation, analysis, specification, and validation of requirements for software [http://en.wikipedia.org/wiki/Software_requirements].
- ***Software design*** is a process of problem solving and planning for a software solution. After the purpose and specifications of software are determined, software developers will design or employ designers to develop a plan for a solution [http://en.wikipedia.org /wiki /Software design].
- ***Software construction*** (also known as software development, application development, software design, designing software, software application development, enterprise application development, or platform development) is the development of a software product. The term "software development" may be used to refer to the activity of computer programming, which is the process of writing and maintaining the source code. But in a broader sense of the term, it includes all that is involved between the conception of the desired software through to the final manifestation of the software, ideally in a planned and structured process [http://en.wikipedia.org/wiki/Software_development].
- ***Software testing*** is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. Software testing can also provide an objective, independent view of the software to allow the business to appreciate and un-

derstand the risks of software implementation [http://en.wikipedia.org/wiki/Software_testing].

2.1.3 Software design process. Software design is generally considered to be a two-step process [SWEBOK 2004]:

- ***Architectural design*** — *Architectural design* describes how software is decomposed and organized into components (the software architecture).
- ***Detailed design*** — *Detailed design* describes the specific behavior of these components. The output of this process is a set of models and artifacts that record the major decisions that have been taken.

2.1.4 Enabling techniques. *Software design principles*, also called *enabling techniques*, are key notions considered fundamental to many different software design approaches and concepts. According to the Oxford English Dictionary, a principle is "a basic truth or a general law … that is used as a basis of reasoning or a guide to action." Some enabling techniques are [SWEBOK 2004]:

- ***Abstraction*** — *Abstraction* is "the process of forgetting information so that things that are different can be treated as if they were the same" [Liskov & Guttag 2001]. In the context of software design, two key abstraction mechanisms are parameterization and specification.
  - ***Abstraction by parameterization*** — Rather than write code that mentions specific values on which computation is to occur, we write functions. Functions describe a computation that works on all acceptable values of the appropriate types. Thus, the detail of what specific values are to be used is removed. Parameterized types are another example of abstraction by parameterization, although there the parameters are *types* rather than *values*.
  - ***Abstraction by specification*** — A well-designed specification removes unnecessary detail about the actual type or value being specified. The specification serves as a contract between the implementer and the user (client), making the job of both parties simpler and making the code more extensible and maintainable. This idea is also known as information hiding or encapsulation in the object-oriented world [http://www.cs.cornell.edu/courses/cs312/2007sp/lectures/lec06.html].
- ***Coupling and cohesion*** — *Coupling* is defined as the strength of the relationships between modules, whereas *cohesion* is defined by how the elements making up a module are related.
- ***Decomposition and modularization*** — *Decomposing* and *modularizing* refers to the decomposition and modularization of a large element of software into a number of smaller independent ones, usually with the goal of placing different functionalities or responsibilities in different components.
- ***Encapsulation/information hiding*** — *Encapsulation/information hiding* means grouping and packaging the components and internal details of an abstraction and making those details inaccessible.

- ***Separation of interface and implementation*** —*Separating interface* and *implementation* involves defining a component by specifying a public interface, known to the clients, separate from the details of how the component is realized.
- ***Sufficiency, completeness and primitiveness*** —*Achieving sufficiency, completeness,* and *primitiveness* means ensuring that a software component captures all the important characteristics of an abstraction, and nothing more.

2.2 Key Issues in Software Design

A number of key issues must be dealt with when designing software. Some are quality concerns that all software must address—for example, performance. Another important issue is how to decompose, organize, and package software components. In contrast, other issues "deal with some aspect of software's behavior that is not in the application domain, but which addresses some of the supporting domains" [Bosch 2000]. Such issues, which often cross-cut the system's functionality, have been referred to as *aspects:* which "tend not to be units of software's functional decomposition, but rather to be properties that affect the performance or semantics of the components in systemic ways" [Kiczales et al. 1997]. The following are a number of these key, cross-cutting issues are the following (presented in alphabetical order) [SWEBOK 2004, Chapter 3]:

- ***Concurrency*** — How to decompose the software into processes, tasks, and threads and deal with related efficiency, atomicity, synchronization, and scheduling issues.
- ***Control and handling of events***— How to organize data and control flow, and how to handle reactive and temporal events through various mechanisms such as implicit invocation and call-backs.
- ***Distribution of components*** — How to distribute the software across the hardware, how the components communicate, how middleware can be used to deal with heterogeneous software.
- ***Error and exception handling and fault tolerance*** — How to prevent and tolerate faults and deal with exceptional conditions.
- ***Interaction and presentation*** — How to structure and organize the interactions with users and the presentation of information.
- ***Data persistence*** — How long-lived data are to be handled.

2.3 Software Structure and Architecture

In its strict sense, software architecture is "a description of the subsystems and components of a software system and the relationships between them." Architecture thus attempts to define the internal structure—according to the Oxford English Dictionary, "the way in which something is constructed or organized"— of the resulting software. During the mid-1990s, however, software architecture started to emerge as a broader discipline involving the study of software structures and architectures in a more generic way. This gave rise to a number of interesting ideas about software design at different levels of abstraction. Some of these concepts can be useful during the architectural design (for example, architectural style) of specific software, as well as during its detailed design (for example, lower-level design patterns). But they can also be useful for designing generic systems, leading to the design of families of programs (also known as product lines).

Interestingly, most of these concepts can be seen as attempts to describe, and thus reuse, generic design knowledge [SWEBOK 2004].

2.3.1 Architectural structures and viewpoints. Different high-level facets of a software design can and should be described and documented. These facets are often called views: "A view represents a partial aspect of a software architecture that shows specific properties of a software system" [Buschmann 1996]. These distinct views pertain to distinct issues associated with software design—for example, the logical view (satisfying the functional requirements) versus the process view (concurrency issues) versus the physical view (distribution issues) versus the development view (how the design is broken down into implementation units). Other authors use different terminologies, like behavioral versus functional versus structural versus data modeling views. In summary, a software design is a multi-faceted artifact produced by the design process and generally composed of relatively independent and orthogonal views [SWEBOK 2004].

An architectural style is "a set of constraints on an architecture [that] defines a set or family of architectures that satisfies them" [Bass 2003, Chapter 2]. An architectural style can thus be seen as a meta-model which can provide software's high-level organization (its macro architecture) [SWEBOK 2004].

- General structure (for example, layers, pipes, and filters, blackboard)
- Distributed systems (for example, client-server, three-tiers, broker)
- Interactive systems (for example, model-view-controller, presentation-abstraction-control)
- Adaptable systems (for example, micro-kernel, reflection)
- Others (for example, batch, interpreters, process control, rule-based)

2.3.2 Design patterns. Succinctly described, a pattern is "a common solution to a common problem in a given context" [Jacobson, Booch, & Rumbaugh 1999]. While architectural styles can be viewed as patterns describing the high-level organization of software (it is macro-architecture), other design patterns can be used to describe details at a lower, more local level (its micro-architecture). Types of design patterns include [http://www.oodesign.com/]:

- Creational patterns (for example: builder, factory, prototype, and singleton)
- Structural patterns (for example: adapter, bridge, composite, decorator, façade, flyweight, and proxy)
- Behavioral patterns (for example: command, interpreter, mediator, memento, observer, state, strategy, template, and visitor)

2.3.3 Families of programs and frameworks. One possible approach to allow the reuse of software designs and components is to design families of software, also known as software product lines. This can be done by identifying the commonalities among members of such families and by using reusable and customizable components to account for the variability among family members.

In object-oriented programming, a key related notion is that of the framework: a partially complete software subsystem that can be extended by appropriately instantiating specific plug-ins (also known as hot spots).

2.4 Human Computer Interface Design

Computer system design encompasses a spectrum of activities from hardware design to user interface design. While specialists are often employed for hardware design and for the graphic design of web pages, only large organizations normally employ specialist interface designers for their application software. Therefore, software engineers must often take responsibility for user interface design as well as for the design of the software to implement that interface [Sommerville 2006, p. 363].

2.4.1 General HCI design principles. The Association for Computing Machinery (ACM) defines human-computer interaction as "a discipline concerned with the design, evaluation and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them" [ACM SIGCHI 1996].

Human–computer interaction (HCI) is the study of interaction between people (users) and computers. Interaction between users and computers occurs at the user interface (or simply *interface*), which includes both software and hardware; for example, characters or objects displayed by software on a personal computer's monitor, input received from users via hardware peripherals such as keyboards and mice, and other user interactions with large-scale computerized systems such as aircraft and power plants [http://en.wikipedia.org/wiki /Human_computer _interaction].

HCI design principles vary greatly depending upon whoever is reporting. The list of design principles by Sommerville appears to be the best, the most understandable, and among the shortest [Sommerville 2006, p. 364].

- ***User familiarity*** — The interface should use terms and concepts drawn from the experience of those who will make the most use of the system.
- ***Consistency*** — The interface should be consistent in that, wherever possible, comparable operations should be activated in the same way.
- ***Minimal surprise*** — Users should never be surprised by the behavior of a system.
- ***Recoverability*** — The interface should include mechanisms to allow users to recover from errors.
- ***User guidance*** — The interface should provide meaningful feedback when errors occur and provide context-sensitive user help.
- ***User diversity*** — The interface should provide appropriate interaction facilities for different types of system users.

2.4.2 Use of modes. A *mode* is defined as a particular form or variation of something. A HCI mode is a distinct method of operation within a computer program, in which the same input can produce different perceived results depending on the state of the computer program. For example, 'caps lock' sets an input mode in which typed letters are uppercase by default; the same

typing produces lowercase letters when not in the caps lock mode. Heavy use of modes often reduces the usability of a user interface, as the user must expend effort to remember current mode states, and switch between mode states as necessary [http://en.wikipediawiki/User _interface].

2.4.3 Use of navigation. *Navigation* is the ability to move efficiently and effectively through a document, such as web pages that are linked together to find a particular word or paragraph. There are several standard navigational layouts — hierarchical, linear and webbed.

- *Linear navigation systems* only allow users to navigate through the interface in one way. Linear systems are normally non- or minimally interactive, with start, next, and back buttons but no others. Linear systems are the most structured of all the categories.
- A *hierarchical based site* is similar to a family tree in that every page has a parent page. Each page has only one page leading to it. The structure of sites of this type is very rigid, but can be useful for organizations with discrete departments requiring one section each. Breadcrumb trails can also be used in a hierarchical site.
- A *webbed topology* allows the user to navigate in a more fluid fashion. The range of webbed topologies is quite broad, ranging from almost hierarchical with additional links, to those in which each page will link to others of similar or related topics, but without an underlying categorization by topic.

2.4.4 Coding techniques and visual design. Coding and visual design include such techniques as the use of different colors, icons, and fonts.

2.4.4.1 Color. Color can improve user interfaces by helping users to understand and manage complexity. However, it is easy to misuse color and to create user interfaces that are visually unattractive and error-prone Shneiderman [1998] gives 14 key guidelines for the effective use of color in user interfaces. The most important of these are [Sommerville 2006, p. 16]:

- Limit the number of colors employed and be conservative as to how they are used. You should not use more than four or five separate colors in a window and no more than seven in a system interface. If you use too many, or if they are too bright, the display may be confusing. Some users may find masses of color disturbing and visually tiring. User confusion is also possible if colors are used inconsistently.
- Use color change to show a change in system status. If a display changes color, this should mean that a significant event has occurred. Thus, in a fuel gauge, you could use a change of color to indicate that fuel is running low. Color highlighting is particularly important in complex displays where hundreds of distinct entities may be displayed.
- Use color coding to support the task users are trying to perform. If they have to identify anomalous instances, highlight these instances; if similarities are also to be discovered, highlight these using a different color.
- Use color coding in a thoughtful and consistent way. For instance, if one part of a system displays error messages in red, all other parts should do likewise. Red should not be used for anything else. If it is, the user may interpret the red display as an error message.

- Be careful about color pairings. Because of the physiology of the eye, people cannot focus on red and blue simultaneously. Eyestrain is a likely consequence of a red on blue display. Other color combinations may also be visually disturbing or difficult to read.

In general, you should use color for highlighting, but you should not associate meanings with particular colors. About 10% of men are color-blind and may misinterpret the meaning. Human color perceptions are different, and there are different conventions in different professions about the meaning of particular colors. Users with different backgrounds may unconsciously interpret the same color in different ways. For example, to a driver, red usually means danger. However, to a chemist, red means hot.

2.4.4.2 Icons. Icons are pictographic representations of data or processes within a computer system, which have been used to replace commands and menus as the means by which the computer supports a dialogue with the end-user. They have been applied principally to graphics-based interfaces to operating systems, networks and document-processing software.

2.4.4.3 Fonts. A study of fonts was conducted at the Wichita State University [Bernard, Lial, & Mills 2001] to determine the impact of font size and style on legibility, reading time, and general preference when read by an older population. The study involved test volunteers reading text passages containing two serif and sans serif fonts at 12- and 14-point sizes. Two types of fonts were used, the serif fonts Georgia and Times New Roman, and the sans serif fonts Arial and Verdana. Both Times New Roman and Arial were originally developed for print and are the most common fonts of their respective font type used today. Georgia and Verdana, however, were developed specifically for optimized viewing on a computer screen.

Several observations can be made from these findings:

- First, 14-point fonts were found to be more legible and to promote faster reading, and were preferred to the 12-point fonts.
- Second, at the 14-point size, serif fonts tended to support faster reading. (Serif fonts, however, were generally not preferred over the sans serif fonts.)
- Third, there was essentially no difference between the computer fonts and the print fonts.

Thus, in light of these results, it is recommended to use 14-point sized fonts for presenting online text to older readers. However, a compromise must be made in deciding which font type to use. If speed of reading is paramount, then serif fonts are recommended. However, if font preference is important, then sans serif fonts are recommended.

2.4.5 Response time and feedback. Slow response times and difficult navigation are the most common complaints of Internet users. After waiting past a certain "attention threshold," users bail out to look for a faster site. Of course, exactly where that threshold is depends on many factors. How compelling is the experience? Is there effective feedback? The following is a set of response and feedback times proposed by Dr. Ben Shneiderman [1998]:

- Load in under 8.6 seconds (non-incremental display)
- Decrease these load times by 0.5 to 1.5 seconds for dynamic transactions

- Minimize the number of steps needed to accomplish tasks to avoid cumulative frustration from exceeding user time budgets
- Load in under 20 to 30 seconds (incremental display) with useful content within 2 seconds
- Provide performance information
- Equalize page download times to minimize delay variability

2.4.6 Design modalities. *Multimodal interfaces* are interfaces that support perceptual capabilities (e.g., auditory, speech, and visuals) as a means of facilitating human interaction with computers [Sears & Jacko 2007, p. 861]. There are a number of different HCI interface designs that focus on the human side of the interface, for example, menu-driven, command-line, and event-driven interfaces. Other interface support tools make interacting with computer systems easier, e.g., forms and question-answering interfaces.

- **Menu-driven interface** — This interface consists of a series of screens that are navigated by choosing options from lists, i.e. menus. (Here, "menu" is not used to refer to pull-down menus, but to lists of options on the screen that lead to other screens.) Because of their simplicity, menu-driven interfaces are commonly used for walk-up-and-use systems, such as information kiosks and ATMs. Websites are also often designed with the same basic navigation principle, where navigation bars substitute for "menus."
- **Command-line interface** — This interface is a means of operating a computer by typing a text command in response to an on-screen prompt and hitting the Enter or Return key to issue the command. The computer then processes the command, displays whatever output is appropriate, and presents another prompt for the next command. Typical commands are to run a program, enter a text editor, list files, and change directories. This mode of interaction is common, for instance, in the traditional DOS and UNIX operating systems.
- **Event-driven interface** — This kind of interface is common to most modern operating systems where the user can initiate actions at any time — the system responds to user "events," such as typing, mouse movements, or mouse clicks.
- **Forms dialogue boxes** — The user employs this interface to communicate with the system by filling in an on-screen form (e.g., a data entry form on a database). Design of the form must be clearly worded and presented, and color and highlights can be used. Form filling enables experienced users to enter data quickly and is user-friendly to the less experienced user.
- **Question-answer interface** — In this interface the application asks questions, and when the user provides answers containing all necessary data, the application gives the results. Sometimes these are called "walkthrough and use" or "interview" applications.

2.4.7 Localization and internationalization. In computing, *internationalization* and *localization* are means of adapting computer software (including HCI software) to different languages, regional differences and technical requirements of a target market. *Internationalization* is the process of designing a software application so that it can be adapted to various languages and regions without engineering changes. *Localization* is the adapting internationalized software for a specific region or language by adding locale-specific components and translating text.

2.4.8 Human-computer interface design methods. Shneiderman's "Eight Golden Rules of Interface Design" were documented in the text *Designing the User Interface* [Shneiderman 1998]. Shneiderman proposed this collection of principles that are derived heuristically from experience and are applicable in most interactive systems after being properly refined, extended, and interpreted.

1. ***Strive for consistency*** — Consistent sequences of actions should be required in similar situations; identical terminology should be used in prompts, menus, and help screens; and consistent commands should be employed throughout.
2. ***Enable frequent users to use shortcuts*** — As the frequency of use increases, so do the user's desires to reduce the number of interactions and to increase the pace of interaction. Abbreviations function keys, hidden commands, and macro facilities are very helpful to an expert user.
3. ***Offer informative feedback*** — For every operator action, there should be some system feedback. For frequent and minor actions, the response can be modest, while for infrequent and major actions, the response should be more substantial.
4. ***Design dialog to yield closure*** — Sequences of actions should be organized into groups with a beginning, middle, and end. The informative feedback at the completion of a group of actions gives the operators the satisfaction of accomplishment, a sense of relief, the signal to drop contingency plans and options from their minds, and an indication that the way is clear to prepare for the next group of actions.
5. ***Offer simple error handling*** — As much as possible, design the system so the user cannot make a serious error. If an error is made, the system should be able to detect the error and offer simple, comprehensible mechanisms for handling the error.
6. ***Permit easy reversal of actions*** — This feature relieves anxiety, since the user knows that errors can be undone; it thus encourages exploration of unfamiliar options. The units of reversibility may be a single action, a data entry, or a complete group of actions.
7. ***Support internal locus of control*** — Experienced operators strongly desire the sense that they are in charge of the system and that the system responds to their actions. Design the system to make users the initiators of actions rather than the responders.
8. ***Reduce short-term memory load*** — The limitation of human information processing in short-term memory requires that displays be kept simple, multiple page displays be consolidated, window-motion frequency be reduced, and sufficient training time be allotted for codes, mnemonics, and sequences of actions.

2.4.9 Multimedia. (I/O, voice, natural language, web-page, sound.). *Multimedia* is media and content that uses a combination of different content forms. The term can be used as a noun (a medium with multiple content forms) or as an adjective describing a medium as having multiple content forms. The term is used in contrast to media which only use traditional forms of printed or hand-produced material. Multimedia includes a combination of *text, audio, still images, animation, video,* and *interactivity content forms.*

Multimedia is usually recorded and played, displayed or accessed by information content processing devices, such as computerized and electronic devices, but can also be part of a live

performance. *Multimedia* (as an adjective) also describes electronic media devices used to store and experience multimedia content [http://en.wikipedia.org/wiki/Multimedia].

- ***Input and Output*** — In computing, *input/output,* or *I/O,* refers to the communication between an information processing system (such as a computer), and the outside world, possibly a human, or another information processing system. Inputs are the signals or data received by the system, and outputs are the signals or data sent from it. The term can also be used as part of an action; to "perform I/O" is to perform an input or output operation. I/O devices are used by a person (or other system) to communicate with a computer. For instance, a keyboard or a mouse may be an input device for a computer, while monitors and printers are considered output devices for a computer. Devices for communication between computers, such as modems and network cards, typically serve for both input and output.

  Note that the designation of a device as either input or output depends on the perspective. Mice and keyboards take as input physical movement that the human user outputs and converts this movement into signals that a computer can understand. Hence the output from these devices is input for the computer. Similarly, printers and monitors take as input signals that a computer outputs. They then convert these signals into representations that human users can see or read. For a human user, the process of reading or seeing these representations comprises the reception of input. These interactions between computers and humans are studied in a field called human–computer interaction (HCI) [http://en .wikipedia.org/wiki/Input/output].

  Devices must be constructed for mediating between humans and machines.

  - ***Input devices*** — mechanics of particular devices, performance characteristics (human and system), devices for the disabled, handwriting and gestures, speech input, eye tracking, exotic devices (e.g., EEG and other biological signals).
  - ***Output devices*** — mechanics of particular devices, vector and raster devices, frame buffers and image stores, canvases, event handling, performance characteristics, devices for the disabled, sound and speech output, 3D displays, motion (e.g., flight simulators), exotic devices.
  - ***Characteristics of I/O devices*** — (e.g., weight, portability, bandwidth, sensory modality).

- ***Natural languages*** — *Natural language* deals with computer systems that interpret the languages that humans use. The ultimate goal is to eventually be able to communicate with your computer as you would with another person. Unfortunately, natural language, which is the easiest for humans to learn, is the hardest for computers to learn. The Redmond-based Natural Language Processing Group described at [http://research.microsoft .com/en-us/groups/nlp/] is working towards developing algorithms and statistical models that can interpret natural language efficiently. The group's advancements have been integrated into applications including information recovery, text critiquing, question answering, gaming, and many others. As the group's work progresses, they anticipate it will enable people to communicate with computers through natural language [https://wiki spaces .psu.edu /display/331Grp1/Natural+language+HCI+Information].

- ***Sound*** — *Sound recording and reproduction* is an electrical or mechanical inscription and re-creation of sound waves, such as spoken voice, singing, instrumental music, or sound effects. The two main classes of sound recording technology are analog recording and digital recording. Acoustic analog recording is achieved by a small microphone diaphragm that can detect changes in atmospheric pressure (acoustic sound waves) and record them as a graphic representation of the sound waves on a medium such as a phonograph. Digital recording and reproduction converts the analog sound signal picked up by the microphone to a digital form by a process of digitization, allowing it to be stored and transmitted by a wider variety of media [http://en.wikipedia.org/wiki/Sound-_recording _and _reproduction].

- ***Text*** — *Text* (writing) is the representation of language in a textual medium through the use of a set of signs or symbols (known as a writing system). It is distinguished from illustration, such as cave drawing and painting, and non-symbolic preservation of language via non-textual media, such as magnetic tape audio [http://en.wikipedia.org/wiki/Writing].

- ***Voice*** — *Voice* (or vocalization) is the sound produced by humans and other vertebrates using the lungs and the vocal folds in the larynx, or voice box. Voice is not always produced as speech, however. Infants babble and coo; animals bark, moo, whinny, growl, and meow; and adult humans laugh, sing, and cry. Voice is generated by airflow from the lungs as the vocal folds are brought close together. When air is pushed past the vocal folds with sufficient pressure, the vocal folds vibrate. If the vocal folds in the larynx did not vibrate normally, speech could only be produced as a whisper. Your voice is as unique as your fingerprint. It helps define your personality, mood, and health [http:// www.nidcd.nih.gov/health/voice/whatis _vsl.html].

  Voice is a major means of multimedia. Voice can be applied to many other multimedia types to increase the understanding by the receiver (listener).

- ***Web Page*** — Each web page (also known as a webpage) represents various types of information presented to the visitor in an aesthetic and readable manner. Most of the web pages are available on the World Wide Web, which makes them widely accessible to the Internet public. The information on a web page is displayed online with the help of a web browser, which connects with the server where the website's contents are hosted through the Hypertext Transfer Protocol (HTTP) [http://www.ntchosting.com/internet/webpage .html].

2.4.10 Metaphors and conceptual models. *Metaphors* are linguistic devices that express an abstract concept through analogy. The use of metaphors allows unfamiliar and abstract concepts to be more readily grasped and understood.

An example of the use of a metaphor in everyday speech is when we talk about time as if it is money or currency. Time is an abstract concept and by using a metaphor to make it more familiar and understandable, we can talk about it more freely. By using this money metaphor in relation to time, it has become normal for us to save, spend, give, waste, and borrow time.

Metaphors are important within HCI because they allow users to apply their understanding of everyday objects and situations to help them understand concepts within a computing environment.

The desktop metaphor is one which has been used from an early stage by the Mac Windows System which then of course led to the Windows Operating System by Microsoft. It is important to note that not all the functionality of a real-world desktop can be transformed into a virtual counterpart. This is a situation in which novice users who expect a certain behavior based on the real world but are surprised when things aren't quite the same, e.g., "icons" — some users expect religious objects.

A *conceptual model* describes the way a system is meant to be understood. A good conceptual model that is applied properly in the design of a system will enable a user to develop a good mental model associated with the system. There are typically many metaphors and mental models of users that can be used in ISD [Instructional System Design] to help users gain a good understanding of the system. The user's mental model of a system is developed by viewing and/or experiencing the system and its visible functionality and structure.

For an interactive systems designer, it is good practice to start with a desired mental model and then develop the interface with the intention of conveying that mental model explicitly to the user through a conceptual model. Parts of the system that may clash with the conceptual model can be hidden from the user in order to maintain a good conceptual model, which will hopefully lead to ease of use for the user [http://www.computingstudents.com/notes/interactive _systems /metaphors_conceptual_models.php].

2.4.11 Psychology of HCI. *Cognitive psychologists* who work in the software industry typically find themselves designing and evaluating complex software systems to aid humans in a wide range of problem domains, like word processing, interpersonal communications, information access, finance, remote meeting support, air traffic control, or even gaming situations. In these domains, the technologies and the users' tasks are in a constant state of flux, evolution and co-evolution. Cognitive psychologists working in human-computer interaction design may try to start from first principles developing these systems, but they often encounter novel usage scenarios for which no guidance is available. For this reason, we believe that there is not as much application of theories, models, and specific findings from basic psychological research to user interface (UI) design as one would hope. However, several analysis techniques and some guidelines generated from the literature are useful [Dumais and Czerwinski 2001].

2.5 Software Design Quality Analysis and Evaluation

This section includes a number of quality and evaluation topics that are specifically related to software design [SWEBOK 2004].

2.5.1 Quality attributes. Various attributes are generally considered important for obtaining a software design of good quality [e.g., various "ilities" (maintainability, portability, testability, traceability), various "nesses" (correctness, robustness), including "fitness for purpose."]

An interesting distinction is the one between quality attributes discernable at run-time (performance, security, availability, functionality, usability), those not discernable at run-time (modifiability, portability, reusability, integrability, and testability), and those related to the architecture's intrinsic qualities (conceptual integrity, correctness, and completeness, build ability) [SWEBOK 2004].

2.5.2 Quality analysis and evaluation techniques. Various tools and techniques can help ensure a software design's quality [SWEBOK 2004].

- ***Software design reviews*** — Informal or semiformal, often group-based, techniques to verify and ensure the quality of design artifacts.
- ***Static analysis*** — Formal or semiformal static (non-executable) analysis that can be used to evaluate a design (for example, fault-tree analysis or automated cross-checking).
- ***Simulation and prototyping*** — Dynamic techniques to evaluate a design (for example, performance simulation or feasibility prototype).

2.5.3 Measures. Measures can be used to assess or to quantitatively estimate various aspects of a software design's size, structure, or quality. Most measures that depend on the approach used for producing the design. These measures are classified into two broad categories [SWEBOK 2004]:

- *Function-oriented (structured) design* measures the design's structure, obtained mostly through functional decomposition and generally represented as a structure chart (sometimes called a hierarchical diagram) on which various measures can be computed.
- *Object-oriented design* measures the design's overall structure, often represented as a class diagram, on which various measures can be computed. Measures of the properties of each class's internal content can also be computed.

2.6 Software Design Notations

Many notations and languages exist to represent software design artifacts. Some are used mainly to describe a design's structural organization, others to represent software behavior. Certain notations are used mostly during architectural design and others mainly during detailed design, although some notations can be used in both steps. In addition, some notations are used mostly in the context of specific methods. Here, they are categorized into notations for describing the structural (static) view versus the behavioral (dynamic) view [SWEBOK 2004].

2.6.1 Structural Descriptions (static view). The following notations, mostly (but not always) graphical, describe and represent the structural aspects of a software design — that is, they describe the major components and how they are interconnected (static view) [SWEBOK 2004]:

- ***Architecture description languages*** (ADLs) — Textual, often formal, languages used to describe software architecture in terms of components and connectors.
- ***Class and object diagrams*** — Used to represent a set of classes (and objects) and their interrelationships.
- ***Component diagrams*** — Used to represent a set of components ("physical and replaceable part[s] of a system that [conform] to and [provide] the realization of a set of interfaces") [Booch, Rumbaugh, & Jacobson 1999] and their interrelationships.
- ***Class responsibility collaborator (CRC) models*** — Used to denote the names of components (class), their responsibilities, and their collaborating components' names [http://www.agilemodeling.com/artifacts/crcModel.htm].
- ***Deployment diagrams*** — Used to represent a set of (physical) nodes and their interrelationships, and thus to model the physical aspects of a system.
- ***Entity-relationship diagrams (ERDs)*** — Used to represent conceptual models of data stored in information systems.

- ***Interface description languages (IDLs)*** — Programming-like languages used to define the interfaces (names and types of exported operations) of software components.
- ***Jackson structure diagrams (JSDs)*** — Used to describe the data structures in terms of sequence, selection, and iteration.
- ***Structure charts*** — Used to describe the calling structure of programs (which module calls, and is called by, which other modules).

2.6.2 Behavioral descriptions (dynamic view). The following notations and languages, some graphical and some textual, are used to describe the dynamic behavior of software and components. Many of these notations are useful mostly, but not exclusively, during detailed design [SWEBOK 2004]:

- ***Activity diagrams*** — Used to show the control flow from activity ("ongoing non-atomic execution within a state machine") to activity.
- ***Collaboration diagrams*** — Used to show the interactions that occur among a group of objects, where the emphasis is on the objects, their links, and the messages they exchange on these links.
- ***Data flow diagrams (DFDs)*** — Used to show data flow among a set of processes.
- ***Decision tables and diagrams*** — Used to represent complex combinations of conditions and actions.
- ***Flowcharts and structured flowcharts*** — Used to represent the flow of control and the associated actions to be performed.
- ***Sequence diagrams*** — Used to show the interactions among a group of objects, with emphasis on the time-ordering of messages.
- ***State transition and state-chart diagrams*** —Used to show the control flow from state to state in a state machine.
- ***Formal specification languages*** — Textual languages that use basic notions from mathematics (for example, logic, set, and sequence) to rigorously and abstractly define software component interfaces and behavior, often in terms of pre- and post-conditions.
- ***Pseudocode and program design languages (PDLs)*** — Structured-programming-like languages used to describe, generally at the detailed design stage, the behavior of a procedure or method.

2.7 Software Design Strategies and Methods

There exist various general strategies to help guide the design process. In contrast to general strategies, methods are more specific in that they generally suggest and provide a set of notations to be used with the method, a description of the process to be used when following the method, and a set of guidelines in using the method. Such methods are useful as a means of transferring knowledge and as a common framework for teams of software engineers.

2.7.1 General strategies. Some often-cited examples of general strategies useful in the design process are divide-and-conquer and stepwise refinement, top-down versus bottom-up strategies, data abstraction and information hiding, use of heuristics, use of patterns and pattern languages, and use of an iterative and incremental approach [SWEBOK 2004].

2.7.2 Function-oriented (structured) design. This is one of the classical methods of software design, where decomposition centers on identifying the major software functions and then elaborating and refining them in a top-down manner. Structured design is generally used after structured analysis, thus producing, among other things, data flow diagrams and associated process descriptions. Researchers have proposed various strategies (for example, transformation analysis, transaction analysis) and heuristics (for example, fan-in/fan-out, scope of effect versus scope of control) to transform a DFD into a software architecture generally represented as a structure chart [SWEBOK 2004].

2.7.3 Object-oriented design. Numerous software design methods based on objects have been proposed. The field has evolved from the early object-based design of the mid-1980s (noun = object; verb = method; adjective = attribute) through OO design, where inheritance and polymorphism (See Chapter 3.2, Paragraph 3.5.3 for definitions) play a key role, to the field of component-based design, where meta-information can be defined and accessed. Although OO design's roots stem from the concept of data abstraction, responsibility-driven design has also been proposed as an alternative approach to OO design [SWEBOK 2004; http://en.wikipedia .org/wiki /Responsibility-driven_design].

2.7.4 Data-structure-centered design. Data-structure-centered design (for example, Jackson, Warnier-Orr) starts from the data structures a program manipulates rather than from the function it performs. The software engineer first describes the input and output data structures (using Jackson's structure diagrams, for instance) and then develops the program's control structure based on these data structure diagrams. Various heuristics have been proposed to deal with special cases—for example, when there is a mismatch between the input and output structures.

2.7.5 Component-based design (CBD). A software component is an independent unit, having well-defined interfaces and dependencies that can be composed and deployed independently. Component-based design addresses issues related to providing, developing, and integrating such components in order to improve reuse [SWEBOK 2004].

2.7.6 Other methods. Other interesting but less mainstream approaches also exist, for example, formal and rigorous approaches (VDM, Z, etc.); SADT; RUP; state charts.

References

Additional information on the software design KA can be found in the following documents:

- **[ACM SIGCHI 1996]** Thomas T. Hewett, Ronald Baecker, Tom Carey, Jean Gasen, Marilyn Mantei, Gary Perlman, Gary Strong, and William Verplank, *Curricula for Human-Computer Interaction* ACM, New York, 1996.
- **[Bass 2003]** L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice,* 2nd ed., Addison-Wesley, Reading, MA, 2003.
- **[Bernard, Lial, & Mills 2001]** M. Bernard, C.H. Liao, and M. Mills. "The Effects of Font Type and Size on the Legibility and Reading Time of Online Text by Older Adults," Department of Psychology, Wichita State University, Wichita, KS, 2001.
- **[Booch, Rumbaugh, & Jacobson 1999]** G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide,* Addison-Wesley, Reading, MA, 1999.
- **[Bosch 2000]** J. Bosch, *Design & Use of Software Architectures: Adopting and Evolving a Product-Line Approach,* 1st ed., ACM Press. New York 2000.

- **[Budgen 2003]** David Budgen, *Software Design* (Hardcover), 2nd Edition, Addison-Wesley, Reading, MA, 400 pages, ISBN-13: 978-0201722192. (Recommended as a CSDP exam reference book by the IEEE Computer Society.)
- **[Buschmann 1996]** F. Buschmann *Pattern-Oriented Software Architecture: A System of Patterns,* John Wiley & Sons, New York, 1996.
- **[Clements et al. 2002]** Paul Clements, Felix Bachmann, Len Bass and David Garlan, *Documenting Software Architectures: Views and Beyond* (Hardcover), Addison-Wesley, Reading, MA, 2002, 560 pages, ISBN-13: 978-0201703726. (Recommended as a CSDP exam reference book by the IEEE Computer Society.)
- **[Draper 1998]** Steve W. Draper, *Computer Supported Cooperative Lecture Notes,* University of Glasgow, Glasgow, Scotland, 1998.
- **[Dumais & Czerwinski 2001]** Susan Dumais and Mary Czerwinski, *Building Bridges from Theory to Practice.* Microsoft Research, One Microsoft Way, Redmond, WA, 2001.
- **[Gamma et al. 1994]** Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st edition, Boston: Addison-Wesley, Reading, MA, 1994, 16 pages, ISBN-13: 978-0201633610 (Recommended as a CSDP exam reference book by the IEEE Computer Society).
- **[Jacobson, Booch, & Rumbaugh 1999]** I. Jacobson. G. Booch, and J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, Reading, MA, 1999.
- **[Kiczales et al 1997]** Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin, "Aspect-Oriented Programming," Presented at *ECOOP '97 – Objected-Oriented Programming*, 1997.
- **[Liskov & Guttag 2001]** B. Liskov and J. Guttag, *Program Development in Java: Abstraction, Specification, and Object-Oriented Design,* Addison-Wesley, Reading, MA, 2001.
- **[Nielsen 1994]** Jakob Nielsen, *Usability Engineering* (Paperback), 1st Edition, Morgan Kaufmann, Burlington, MA, 1994, 362 pages, ISBN-13: 978-0125184069 (Recommended as a CSDP exam reference book by the IEEE Computer Society).
- **[Page-Jones 1999]** Meilir Page-Jones, *Fundamental of Object-Oriented Design in UML* (Paperback), 2nd Edition, Addison-Wesley, Reading, MA, 1999, 480 pages, ISBN-13: 978-0201699463. (Recommended as a CSDP exam reference book by the IEEE Computer Society.)
- **[Sears & Jacko 2007]** Andrew Sears and Julie A. Jacko, Editors, *The Human-Computer Interaction Handbook: Fundamentals, Evolving Technologies and Emerging Applications,* Second Edition (Human Factors and Ergonomics), Lawrence Erlbaum Associates, New York, 2007.
- **[Shneiderman 1998]** Ben Shneiderman, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 3d ed., Addison-Wesley, Reading, MA, 1998.
- **[Silberschatz 2008]** Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, *Operating System Concepts* (Hardcover), 8th Edition, John Wiley & Sons, Hoboken, NJ, 2008,

ISBN-13: 978-0470128725. (Recommended as a CSDP exam reference book by the IEEE Computer Society.)

- **[Sommerville 2006]** Ian Sommerville, *Software Engineering,* 8th edition, Addison-Wesley, Harlow, England 2006. (Recommended as a CSDP exam reference book by the IEEE Computer Society.)
- **[SWEBOK 2004]** E. Bourque and R. Dupuis, Editors, *Software Engineering Body of Knowledge (SWEBOK).* IEEE Computer Society Press, Los Alamitos, CA, 2004.
- **[Wikipedia]** is a free web based encyclopedia enabling multiple users to freely add and edit online content.

Chapter 3.1

Welcome to Software Construction[8]

Steven McConnell
Construx Software
Bellevue, WA 98004

You know what "construction" means when it's used outside software development. "Construction" is the work "construction workers" do when they build a house, a school, or a skyscraper. When you were younger, you built things out of "construction paper." In common usage, "construction" refers to the process of building. The construction process might include some aspects of planning, designing, and checking your work, but mostly "construction" refers to the hands-on part of creating something.

1. What Is Software Construction?

Developing computer software can be a complicated process, and in the last 25 years, researchers have identified numerous distinct activities that go into software development. They include:

- Problem definition
- Requirements development
- Construction planning
- Software architecture, or high-level design
- Detailed design
- Coding and debugging
- Unit testing
- Integration testing
- Integration
- System testing
- Corrective maintenance

If you've worked on informal projects, you might think that this list represents a lot of red tape. If you've worked on projects that are too formal, you know that this list represents a lot of red tape. It's hard to strike a balance between too little and too much formality.

If you've taught yourself to program or worked mainly on informal projects, you might not have made distinctions among the many activities that go into creating a software product. Mentally, you might have grouped all of these activities together as "programming." If you

8. This paper is an extract from McConnell's book, *Code Complete: A practical handbook of software construction*, 2nd ed., Microsoft Press, Redmond, WA, 2004. Used with permission of Microsoft, Inc. (This book is recommended by the IEEE Computer Society as a reference book for the CSDP exam.)

work on informal projects, the main activity you think of when you think about creating software is probably the activity the researchers refer to as "construction."

This intuitive notion of "construction" is fairly accurate. but it suffers from a lack of perspective. Putting construction in its context with other activities helps keep the focus on the right tasks during construction and appropriately emphasizes important non-construction activities.

Construction is mostly coding and debugging but also involves detailed design, construction planning, unit testing, integration, integration testing, and other activities. If this were a paper about all aspects of software development, it would feature nicely balanced discussions of all activities in the development process. Because this is a paper about construction techniques, however, it places a lopsided emphasis on construction and only touches on related topics. If this paper were a dog, it would muzzle up to construction, wag its tail at design and testing, and bark at the other development activities.

Construction is also sometimes known as "coding" or "programming." "Coding" isn't really the best word because it implies the mechanical translation of a preexisting design into a computer language, but construction is not at all mechanical and involves substantial creativity and judgment. Throughout the article, I use "programming" interchangeably with "construction."

Here are some of the specific tasks involving construction:

- Verifying that the groundwork has been laid so that construction can proceed successfully
- Determining how your code will be tested
- Designing and writing classes and routines
- Creating and naming variables and named constants
- Selecting control structures and organizing blocks of statements
- Unit testing, integration testing, and debugging your own code
- Reviewing other team members' low-level designs and code and having them review yours
- Polishing code by carefully formatting and commenting it
- Integrating software components that were created separately
- Tuning code to make it faster and use fewer resources

With so many activities at work in construction, you might say, "OK, Jack, what activities are *not* parts of construction?" That's a fair question. Important non-construction activities include management, requirements development, software architecture, user-interface design, system testing, and maintenance. Each of these activities affects the ultimate success of a project as much as construction—at least the success of any project that calls for more than one or two people and lasts longer than a few weeks. You can find good books on each activity; many are listed in the "Additional Resources" sections of *Code Complete: A practical handbook of software construction* [2004].

2. Why Is Software Construction Important?

You probably agree that improving software quality and developer productivity is important. Many of today's most exciting projects use software extensively. The Internet, movie special effects, medical life-support systems, space programs, aeronautics, high-speed financial analysis, and scientific research are a few examples. These projects and more conventional projects can all benefit from improved practices because many of the fundamentals are the same.

If you agree that improving software development is important in general, the question for you as a reader of this document becomes—why is construction an important focus?

Here's why:

- *Construction is a large part of software development.* Depending on the size of the project, construction typically takes 30 to 80 percent of the total time spent on a project. Anything that takes up that much project time is bound to affect the success of the project.

- *Construction is the central activity in software development.* Requirements and architecture are done before construction so that you can do construction effectively. System testing (in the strict sense of independent testing) is done after construction to verify that construction has been done correctly. Construction is at the center of the software-development process.

- *With a focus on construction, the individual programmer's productivity can improve enormously.* A classic study by Sackman, Erikson, and Grant [1968] showed that the productivity of individual programmers varied by a factor of 10 to 20 during construction. Since their study, their results have been confirmed by numerous other studies [Curtis 1981, Mills 1983, Curtis et al. 1986, Card 1987, Valett & McGarry 1989, DeMarco & Lister 1999, Boehm et al. 2000]. This paper helps all programmers learn techniques that are already used by the best programmers.

- *Construction's product, the source code, is often the only accurate description of the software*. In many projects, the only documentation available to programmers is the code itself. Requirements specifications and design documents can go out of date, but the source code is always up to date. Consequently, it's imperative that the source code be of the highest possible quality. Consistent application of techniques for source-code improvement makes the difference between a Rube Goldberg contraption and a detailed, correct, and therefore informative program. Such techniques are most effectively applied during construction.

- *Construction is the only activity that's guaranteed to be done.* The ideal software project goes through careful requirements development and architectural design before construction begins. The ideal project undergoes comprehensive, statistically controlled system testing after construction. Imperfect, real-world projects, however, often skip requirements and design to jump into construction. Testing is dropped because developers have too many errors to fix and they've run out of time. But no matter how rushed or poorly planned a project is, you can't drop construction: it's where the rubber meets the road. Improving construction is thus a way of improving any software-development effort, no matter how abbreviated.

3. Software Construction: Building Software

The image of "building" software is more useful than that of "writing" or "growing" software. It's compatible with the idea of software accretion and provides more detailed guidance. Building software implies various stages of planning, preparation, and execution that vary in kind and degree depending on what's being built. When you explore the metaphor, you find many other parallels.

Building a four-foot tower requires a steady hand, a level surface, and 10 undamaged beer cans. Building a tower 100 times that size doesn't merely require 100 times as many beer cans. It requires a different kind of planning and construction altogether.

If you're building a simple structure—a doghouse, say—you can drive to the lumber store and buy some wood and nails. By the end of the afternoon, you'll have a new house for Fido. If you forget to provide for a door, or make some other mistake, it's not a big problem; you can fix it or even start over from the beginning. All you've wasted is part of an afternoon. This loose approach is appropriate for small software projects too. If you use the wrong design for 1000 lines of code, you can start over completely without losing much.

If you're building a house, the building process is more complicated, and so are the consequences of poor design. First you have to decide what kind of house you want to build—analogous in software development to problem definition. Then you and an architect have to come up with a general design and get it approved. This is similar to software architectural design. You draw detailed blueprints and hire a contractor. This is similar to detailed software design. You prepare the building site, lay a foundation, frame the house, put siding and a roof on it, and plumb and wire it. This is similar to software construction. When most of the house is done, the landscapers, painters, and decorators come in to make the best of your property and the home you've built. This is similar to software optimization. Throughout the process, various inspectors come to check the site, foundation, frame, wiring, and other inspectables. This is similar to software reviews and inspections.

Greater complexity and size imply greater consequences in both activities. In building a house, materials are somewhat expensive, but the main expense is labor. Ripping out a wall and moving it six inches is expensive not because you waste a lot of nails but because you have to pay the people for the extra time it takes to move the wall. You have to make the design as good as possible so that you don't waste time fixing mistakes that could have been avoided. In building a software product, materials are even less expensive, but labor costs just as much. Changing a report format is just as expensive as moving a wall in a house because the main cost component in both cases is people's time.

What other parallels do the two activities share? In building a house, you won't try to build things you can buy already built. You'll buy a washer and dryer, dishwasher, refrigerator, and freezer. Unless you're a mechanical wizard, you won't consider building them yourself. You'll also buy prefabricated cabinets, counters, windows, doors, and bathroom fixtures. If you're building a software system, you'll do the same thing. You'll make extensive use of high-level language features rather than writing your own operating-system-level code. You might also use prebuilt libraries of container classes, scientific functions, user interface classes, and database-manipulation classes. It generally doesn't make sense to code things you can buy ready-made.

If you're building a fancy house with first-class furnishings, however, you might have your

cabinets custom-made. You might have a dishwasher, refrigerator, and freezer built in to look like the rest of your cabinets. You might have windows custom-made in unusual shapes and sizes. This customization has parallels in software development. If you're building a first-class software product, you might build your own scientific functions for better speed or accuracy. You might build your own container classes, user interface classes, and database classes to give your system a seamless, perfectly consistent look and feel.

Both building construction and software construction benefit from appropriate levels of planning. If you build software in the wrong order, it's hard to code, hard to test, and hard to debug. It can take longer to complete, or the project can fall apart because everyone's work is too complex and therefore too confusing when it's all combined.

Careful planning doesn't necessarily mean exhaustive planning or over-planning. You can plan out the structural supports and decide later whether to put in hardwood floors or carpeting, what color to paint the walls, what roofing material to use, and so on. A well-planned project improves your ability to change details later. The more experience you have with the kind of software you're building, the more details you can take for granted. You just want to be sure that you plan enough so that lack of planning doesn't create major problems later.

The construction analogy also helps explain why different software projects benefit from different development approaches. In building, you'd use different levels of planning, design, and quality assurance if you're building a warehouse or a toolshed than if you're building a medical center or a nuclear reactor. You'd use still different approaches for building a school, a skyscraper, or a three-bedroom home. Likewise, in software you might generally use flexible, lightweight development approaches, but sometimes you'll need rigid, heavyweight approaches to achieve safety goals and other goals.

Making changes in the software brings up another parallel with building construction. To move a wall six inches costs more if the wall is load-bearing than if it's merely a partition between rooms. Similarly, making structural changes in a program costs more than adding or deleting peripheral features.

Finally, the construction analogy provides insight into extremely large software projects. Because the penalty for failure in an extremely large structure is severe, the structure has to be overengineered. Builders make and inspect their plans carefully. They build in margins of safety; it's better to pay 10 percent more for stronger material than to have a skyscraper fall over. A great deal of attention is paid to timing. When the Empire State Building was built, each delivery truck had a 15-minute margin in which to make its delivery. If a truck wasn't in place at the right time, the whole project was delayed.

Likewise, for extremely large software projects, planning of a higher order is needed than for projects that are merely large. Capers Jones [1998] reports that a software system with one million lines of code requires an average of 69 *kinds* of documentation. The requirements specification for such a system would typically be about 4000-5000 pages long, and the design documentation can easily be two or three times as extensive as the requirements. It's unlikely that an individual would be able to understand the complete design for a project of this size—or even read it. A greater degree of preparation is appropriate.

The building-construction metaphor is quite useful and can be extended in a variety of other directions to represent a variety of other constructions.

4. Combining Metaphors

Because metaphors are heuristic rather than algorithmic, they are not mutually exclusive. You can use both the accretion and the construction metaphors. You can use writing if you want to, and you can combine writing with driving, hunting for werewolves, or drowning in a tar pit with dinosaurs. Use whatever metaphor or combination of metaphors stimulates your own thinking or communicates well with others on your team.

Using metaphors is a fuzzy business. You have to extend them to benefit from the heuristic insights they provide. But if you extend them too far or in the wrong direction, they'll mislead you. Just as you can misuse any powerful tool, you can misuse metaphors; but their power makes them a valuable part of your intellectual toolbox.

5. Selection of Major Construction Practices

Part of preparing for construction is deciding which of the many available good practices you'll emphasize. Some projects use pair programming and test-first development, while others use solo development and formal inspections. Either combination of techniques can work well, depending on specific circumstances of the project.

6. Checklist: Major Construction Practices

The following major construction practices apply independently to coding, team work, quality assurance, and tools.

6.1 Coding

- Have you defined how much design will be done up front and how much will be done at the keyboard, while the code is being written?
- Have you defined coding conventions for names, comments, and layout?
- Have you defined specific coding practices that are implied by the architecture, such as how error conditions will be handled, how security will be addressed, what conventions will be used for class interfaces, what standards will apply to reused code, how much to consider performance while coding, and so on?

6.2 Teamwork

- Have you defined an integration procedure—that is, have you defined the specific steps a programmer must go through before checking code into the master sources?
- Will programmers program in pairs, or individually, or some combination of the two?

6.3 Quality assurance

- Will programmers write test cases for their code before writing the code itself?
- Will programmers write unit tests for their code regardless of whether they write them first or last?
- Will programmers step through their code in the debugger before they check it in?
- Will programmers integration-test their code before they check it in?"
- Will programmers review or inspect each other's code?

6.4 Tools

- Have you selected a revision control tool?
- Have you selected a language and language version or compiler version?
- Have you selected a framework such as J2EE or Microsoft.NET or explicitly decided not to use a framework?
- Have you decided whether to allow use of nonstandard language features?
- Have you identified and acquired other tools you'll be using—editor, refactoring tool, debugger, test framework, syntax checker, and so on?

8. Key Points

Several key points in software construction are:

- Software construction is the central activity in software development; construction is the only activity that's guaranteed to happen on every project.
- The main activities in construction are detailed design, coding, debugging, integration, and developer testing (unit testing and integration testing).
- Other common terms for construction are "coding" and "programming."
- The quality of the construction substantially affects the quality of the software.
- In the final analysis, your understanding of how to do construction determines how good a programmer you are, and that's the subject of *Code Complete: A practical handbook of software construction* [2004].
- Every programming language has strengths and weaknesses. Be aware of the specific strengths and weaknesses of the language you're using.
- Establish programming conventions before you begin programming. It's nearly impossible to change code to match them later.
- More construction practices exist than you can use on any single project. Consciously choose the practices that are best suited to your project.

References:

Additional information on the *software construction* KA can be found in the following documents:

- **[Boehm et al. 2000]** Barry W. Boehm, Chris Abts, A. Winsor Brown, Sunita Chulani, Bradford K. Clark, Ellis Horowitz, Ray Madachy, Donald J. Reifer, Bert Steece, *Software Cost Estimation with COCOMO II.* Addison-Wesley, Boston, MA, 2000.
- **[Card 1987]** D. Card, "A Software Technology Evaluation Program," *Information and Software Technology,* vol. 29, no. 6, 1987, pp. 291-300.
- **[Curtis 1981]** B. Curtis, "Substantiating Programmer Variability," *Proceedings of the IEEE*, Vol. 69, no. 7, 1981, p. 846.

- **[Curtis et al. 1986]** B. Curtis, E.M. Soloway, R.E. Brooks, J.B. Black, K. Ehrlich, H.R. Ramsey, "Software Psychology: The need for an Interdisciplinary Program," *Proceedings of the IEEE,* vol. 74, no. 8, 1986, pp. 1092-1106.
- **[DeMarco & Lister 1999]** T. DeMarco and T. Lister, *Peopleware: Productive and Terms*, 2nd, edition, Dorset House, New York, 1999.
- **[Jones 1998]** C. Jones, *Estimating Software Costs,* McGraw-Hill, New York, 1998.
- **[McConnell 2004]** S. McConnell, *Code Complete: A practical handbook of software construction*, 2nd ed., Microsoft Press, Redmond, WA, 2004.
- **[Mills 1983]** H. Mills, *Software Productivity*, Little Brown, Boston MA, 1983.
- **[Sackman, Erikson, & Grant 1968]** H. Sackman, W.J. Erikson, and E.E. Grant, "Experimental Studies Comparing Online and Offline Programming Performance," *Communications of the ACM,* Vol. 1, No. 1 (January), 1968, pp. 188-204.
- **[Valett & McGarry 1989]** J. Valett and F.E. McGarry, "A Summary of Software Measurement Experience in the Software Engineering Laboratory," *Journal of Systems and Software,* vol. 9, no. 2 (February), 1989, pp. 137-138.

Chapter 3.2
Essentials of Software Construction

Richard Hall Thayer and Merlin Dorfman

This is the third chapter of a textbook to aid individual software engineers in a greater understanding of the IEEE SWEBOK [2013] and a guide book to aid software engineers in passing the IEEE CSDP and CSDA certification exams.

This chapter also introduces concepts and problems of software construction. The term software construction refers to the detailed creation of working, meaningful software through a combination of coding, verification, unit testing, integration testing, and debugging.

Software construction is linked to all the other software engineering efforts, most strongly to software design and software testing. This is because the software construction process itself involves significant software design and test activity. It also uses the output of design and provides one of the inputs to testing. Detailed boundaries between design, construction, and testing (if any) will vary depending upon the software life-cycle processes and methods that are used in a project [SWEBOK 2013].

Chapter 3 covers the CSDP exam specifications for the software construction module [Software Exam Specification, Version 2, 18 March 2009]:

1. Software construction fundamentals (minimizing complexity; anticipating change; constructing for verification; standards in construction)
2. Managing construction (construction models; construction planning; construction measurement)
3. Practical consideration (construction design; construction languages; coding; construction testing; refuse; construction quality; integration; executable models)
4. Construction tools (development environments; GUI builders; unit testing tools; application oriented languages [for example, scripting, visual, domain-specific, markup, and macros]; profiling, performance analysis and slicing tools)
5. Construction technologies Part 1 (API design and use; code reuse and libraries; object-oriented run-time issues [for example, polymorphism and dynamic binding]; parameterization and generics; assertions, design by contract, defensive programming; error handling, exception handling, and fault tolerance; state-based and table driven construction techniques; run-time configuration and internationalization)
6. Construction technologies Part 2 (grammar-based input processing [parsing]; concurrency primitives [such as semaphores and monitors]; middleware [components and containers]; construction methods for distributed software; constructing heterogeneous systems [hardware and software]; hardware-software co-design; performance analysis and tuning; platform standards [Posix, etc.]; test-first programming)

3.1 Software Construction Fundamentals

The fundamentals of software construction include [SWEBOK 2004]:

- Minimizing complexity
- Anticipating change
- Constructing for verification
- Standards in construction

The first three concepts apply to design as well as to construction. The following sections define these concepts and describe how they apply to construction.

3.1.1 Minimizing complexity. A major factor in how people convey intent to computers is the severely limited ability of people to hold complex structures and information in their working memories, especially over long periods of time. This leads to one of the strongest drivers in software construction: *minimizing complexity*. The need to reduce complexity applies to essentially every aspect of software construction, and is particularly critical to the process of verification and testing of software constructions.

In software construction, *reduced complexity* is achieved through emphasizing the creation of code that is simple and readable rather than clever.

3.1.2 Anticipating change. Most software will change over time, and the anticipation of change drives many aspects of software construction. Software is unavoidably part of changing external environments, and changes in those outside environments affect software in diverse ways.

The need to *anticipate change* is supported by many specific construction techniques:

- Communication methods (for example, standards for document formats and contents)
- Programming languages (for example, language standards for languages like Java and C++)
- Platforms (for example, programmer interface standards for operating system calls)
- Tools (for example, diagrammatic standards for notations like UML (Unified Modeling Language)

3.1.3 Constructing for verification. *Constructing for verification* means building software in such a way that faults can be ferreted out readily by the software engineers writing the software, as well as during independent testing and operational activities. Specific techniques that support constructing for verification include following coding standards to support code reviews, unit testing, organizing code to support automated testing, and restricted use of complex or hard-to-understand language structures, among others.

3.1.4 Standards in construction. Standards that directly affect construction issues include:

- ***Use of external standards*** — Construction depends on the use of external standards for construction languages, construction tools, technical interfaces, and interactions between software construction and other knowledge areas (KAs). Standards come from numerous sources, including hardware and software interface specifications such as the Object Management Group (OMG) and international organizations such as the IEEE or ISO.

- ***Use of internal standards*** — Standards may also be created on an organizational basis at the corporate level or for use on specific projects. These standards support coordination of group activities, minimizing complexity, anticipating change, and constructing for verification.

3.2 Managing Construction

The fundamentals of managing construction include [SWEBOK 2004]:

- Construction models
- Construction planning
- Construction measurement

3.2.1 Construction models. Numerous models have been created to develop software, some of which emphasize construction more than others.

Some models are more linear from the construction point of view, such as the waterfall and staged-delivery life-cycle models. These models treat construction as an activity that occurs only after significant prerequisite work has been completed - including detailed requirements, extensive design, and detailed planning. The more linear approaches tend to emphasize the activities that precede construction (requirements and design), and tend to create more distinct separations between the activities. In these models, the main emphasis of construction may be coding.

Other models are more iterative, such as evolutionary prototyping, Extreme Programming, and Scrum. These approaches tend to treat construction as an activity that occurs concurrently with other software development activities, including requirements, design, and planning, or overlaps them. These approaches tend to mix design, coding, and testing activities, and they often treat the combination of activities as construction.

Consequently, what is considered to be "construction" depends to some degree on the life-cycle model used.

3.2.2 Construction planning. The choice of construction method is a key aspect of the construction planning activity. The choice of construction method affects the extent to which construction prerequisites are performed, the order in which they are performed, and the degree to which they are expected to be completed before construction work begins.

The approach to construction affects the project's ability to reduce complexity, anticipate change, and construct for verification. Each of these objectives may also be addressed at the process, requirements, and design levels—but they will also be influenced by the choice of construction method.

Construction planning also defines the order in which components are created and integrated, the software quality management processes, the allocation of task assignments to specific software engineers, and the other tasks, according to the chosen method.

3.2.3 Construction measurement. Numerous construction activities and artifacts can be measured, including code developed, code modified, code reused, code destroyed, code complexity, code inspection statistics, fault-fix and fault-find rates, effort, and scheduling. These measurements can be useful for purposes of managing construction, ensuring quality during construction, improving the construction process, as well as for other reasons.

3.3 Practical Considerations

Construction is an activity in which the software has to come to terms with arbitrary and chaotic real-world constraints, and to do so exactly. Due to its proximity to real-world constraints, construction is more driven by practical considerations than some other knowledge areas, and software engineering is perhaps most craft-like in the construction area.

Some practical considerations in construction design include [SWEBOK 2004]:

- Construction design
- Construction languages
- Coding
- Construction testing
- Reuse
- Construction quality
- Integration
- Executable models

3.3.1 Construction design. Some projects allocate more design activity to construction; others to a phase explicitly focused on design. Regardless of the exact allocation, some detailed design work will occur at the construction level, and that design work tends to be dictated by immovable constraints imposed by the real-world problem that is being addressed by the software. Just as construction workers building a physical structure must make small-scale modifications to account for unanticipated gaps in the builder's plans, software construction workers must make modifications on a smaller or larger scale to flesh out details of the software design during construction [SWEBOK 2004].

3.3.2 Construction languages. *Construction languages* include all forms of communication by which a human can specify an executable problem solution to a computer [SWEBOK 2004].

3.3.2.1 Configuration languages. The simplest type of construction language is a configuration language, in which software engineers choose from a limited set of predefined options to create new or custom software installations. The text-based configuration files used in both the Windows and UNIX operating systems are examples of this, and the menu style selection lists of some program generators constitute another.

Toolkit languages are used to build applications out of toolkits (integrated sets of application-specific reusable parts), and are more complex than configuration languages. Toolkit languages may be explicitly defined as application programming languages (for example, scripts), or may simply be implied by the set of interfaces of a toolkit.

Programming languages are the most flexible type of construction languages. They also contain the least amount of information about specific application areas and development processes, and so require the most training and skill to use effectively.

There are three general kinds of notation used for programming languages, namely Linguistic, Formal, and Visual [SWEBOK 2004].

- *Formal notations* rely less on intuitive, everyday meanings of words and text strings and

more on definitions backed up by precise, unambiguous, and formal (or mathematical) definitions. Formal construction notations and formal methods are at the heart of most forms of system programming, where accuracy, time behavior, and testability are more important than ease of mapping into natural language. Formal constructions also use precisely defined ways of combining symbols that avoid the ambiguity of many natural language constructions.

- *Visual notations* rely much less on the text-oriented notations of both linguistic and formal construction, and instead rely on direct visual interpretation and placement of visual entities that represent the underlying software. Visual construction tends to be somewhat limited by the difficulty of making "complex" statements using only movement of visual entities on a display. However, it can also be a powerful tool in cases where the primary programming task is simply to build and "adjust" a visual interface to a program, the detailed behavior of which has been defined earlier.

3.3.2.2 Object-oriented languages. There are almost two dozen major object-oriented programming languages in use today. However, here are far fewer object-oriented languages in commercial use [http://www.computerclub.i8.com/eoo.htm]. These are:

- C++
- Smalltalk
- Java

3.3.2.2.1 C++. *C++* is an object-oriented version of C. It is compatible with C (it is actually a superset), so that existing C code can be incorporated into C++ programs. C++ programs are fast and efficient, qualities that helped make C an extremely popular programming language. It sacrifices some flexibility in order to remain efficient, however. C++ uses compile-time binding, which means that the programmer must specify the specific class of an object, or at the very least, the most general class that an object can belong to. This makes for high run-time efficiency and small code size, but it trades off some of the power to reuse classes.

C++ has become so popular that most new C compilers are actually C/C++ compilers. However, to take full advantage of object-oriented programming, one must program (and think!) in C++, not C. This can often be a major problem for experienced C programmers. Many programmers think they are coding in C++, but instead are only using a small part of the language's object-oriented power.

3.3.2.2.2 Smalltalk. *Smalltalk* is a pure object-oriented language. While C++ makes some practical compromises to ensure fast execution and small code size, Smalltalk makes none. It uses run-time binding, which means that nothing about the type of an object need be known before a Smalltalk program is run.

Smalltalk programs are considered by most to be significantly faster to develop than C++ programs. A rich-class library that can be easily reused via inheritance is one reason for this. Another reason is Smalltalk's dynamic development environment. It is not explicitly compiled, like C++. This makes the development process more fluid, so that "what if" scenarios can be easily tried out, and classes' definitions easily refined. But being purely object-oriented, programmers cannot simply put their toes in the o-o waters, as with C++. For this reason, Smalltalk generally takes longer to master than C++. But most of this time is actually spent learning object-

oriented methodology and techniques, rather than details of a particular programming language. In fact, Smalltalk is syntactically very simple, much more so than either C or C++.

Unlike C++, which has become standardized, The Smalltalk language differs somewhat from one implementation to another. The most popular commercial "dialects" of Smalltalk are:

- ***VisualWorks*** — *VisualWorks* is arguably the most powerful of Smalltalks. VisualWorks was developed by Parc Place, which grew out of the original Xerox PARC project that invented the Smalltalk language.
- ***Smalltalk/V and Visual Smalltalk*** — Digitalk's versions of Smalltalk are somewhat smaller and simpler, and are specifically tailored to IBM-compatible PCs.
- ***VisualAge*** — IBM's version of Smalltalk. VisualAge is comparable to Smalltalk/V.

3.3.2.2.3 Java. *Java* is a programming language originally developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995 as a core component of Sun Microsystems's Java platform. The language derives much of its syntax from C and C++ but has a simpler object model and fewer low-level facilities. Java applications are typically compiled to byte code (class file) that can run on any Java Virtual Machine (JVM) regardless of computer architecture. Java is a general-purpose, concurrent, class-based, object-oriented language that is specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere." Java is currently one of the most popular programming languages in use, and is widely used from application software to web applications [http://en.wikipedia.org/wiki/Java_(programming_language)].

Java is a curious mixture of C++ and Smalltalk. While it has the syntax of C++, making it easy (or difficult) to learn, depending on your experience, it has improved on C++ in some important areas. For one thing, it has no pointers (pointers are low-level programming constructs that can make for error-prone programs). Like Smalltalk, it has garbage collection, a feature that frees the programmer from explicitly allocating and de-allocating memory. And it runs on a Smalltalk-style virtual machine, software built into your web browser that executes the same standard compiled Java byte codes no matter what type of computer you have.

Java development tools are being rapidly deployed, and are available from such major software companies as IBM, Microsoft, and Symantec.

3.3.3 Coding. The following considerations apply to the software construction coding activity:

- Techniques for creating understandable source code, including naming and source code layout
- Use of classes, enumerated types, variables, named constants, and other similar entities
- Use of control structures
- Handling of error conditions—both planned errors and exceptions (input of bad data, for example)
- Prevention of code-level security breaches (buffer overruns or array index overflows, for example)
- Resource usage via use of exclusion mechanisms and discipline in accessing serially reusable resources (including threads or database locks)

- Source code organization (into statements, routines, classes, packages, or other structures)
- Code documentation
- Code tuning

3.3.4 Construction testing. Construction involves two forms of testing, which are often performed by the software engineer who wrote the code:

- Unit testing
- Integration testing

The purpose of *construction testing* is to reduce the gap between the time at which faults are inserted into the code and the time those faults are detected. It is traditional to develop the construction tests after the code itself is written, but some practitioners and some methods advocate writing the tests before coding.

Construction testing typically involves a subset of types of testing. For instance, construction testing does not typically include system testing, alpha testing, beta testing, stress testing, configuration testing, usability testing, or other, more specialized kinds of testing.

Two standards have been published on the topic: IEEE Std 829-1998, IEEE Standard for Software Test Documentation and IEEE Std 1008-1987, IEEE Standard for Software Unit Testing.

3.3.5 Reuse. Implementing software reuse entails more than creating and using libraries of assets. It requires formalizing the practice of reuse by integrating reuse processes and activities into the software life cycle. However, reuse is important enough in software construction that it is included here as a topic.

The tasks related to reuse in software construction during coding and testing are:

- The selection of the reusable units, databases, test procedures, or test data
- The evaluation of code or test reusability
- The reporting of reuse information on new code, test procedures, or test data

3.3.6 Construction quality. Numerous techniques exist to ensure the quality of code as it is constructed. The primary techniques used for construction include:

- ***Code stepping*** — One of the most common debugger usage scenarios is Code Stepping. When you are debugging under this scheme, you are able to examine the state of the program, variables, and related data before and after executing a particular line of code. This allows you to evaluate the effects of an instruction in isolation and to understand the behavior of the program [http://blogs.msdn.com/b/philpenn/archive /2010/08/31/practical-for-analyzing-concurrency-code-stepping.aspx].
- ***Debugging*** — *Debugging is* a methodical process of finding and reducing the number of bugs, or defects, in a computer program or a piece of electronic hardware, thus making it behave as expected. In software, the debugging process is normally done at unit level by the unit programmer. Debugging at this level can include fixing the error that is identified by the debugging process.

- ***Static analysis*** — *Formal or semiformal static* (non-executable) analysis that can be used to evaluate a design (for example, fault-tree analysis or automated cross-checking).

- ***Technical reviews*** — The purpose of a *technical review* is to evaluate a software product to determine its suitability for its intended use. The objective is to identify discrepancies from approved specifications and standards, i.e., to find technical problems and not to (1) suggest solutions or remedial action or (2) cast blame. The results should provide management with evidence confirming (or not) that the product meets the specifications and adheres to standards and that changes are controlled [IEEE1028-1997].

- ***Test-first development*** — This concept suggests that writing test cases first will minimize the amount of time between when a defect is inserted into the code and when the defect is discovered and removed [McConnell 2004].

- **Testing and integration testing** — *Software testing* consists of the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the specified expected behavior. *Integration testing* is the process of verifying the interaction (interfaces) between software components. Classical integration testing strategies, such as top-down or bottom-up, are used with traditional, hierarchically structured software.

- ***Use of assertions*** — In computer programming, an *assertion* is a predicate (for example a true–false statement) placed in a program to indicate that the developer *thinks* that the predicate is always true at that place [http://en.wikipedia.org/wiki/Assertion_(computing)].

The specific technique or techniques selected depend on the nature of the software being constructed, as well as on the skills set of the software engineers performing the construction.

Construction quality activities are differentiated from other quality activities by their focus. Construction quality activities focus on code and on artifacts that are closely related to code: small-scale designs - as opposed to other artifacts that are less directly connected to the code, such as requirements, high-level designs, and plans.

3.3.7 Integration. A key activity during construction is the integration of separately constructed routines, classes, components, and subsystems. In addition, a particular software system may need to be integrated with other software or hardware systems.

Concerns related to construction integration include planning the sequence in which components will be integrated, creating scaffolding to support interim versions of the software, determining the degree of testing and quality work performed on components before they are integrated and determining points in the project at which interim versions of the software are tested.

3.3.8 Executable models. Executable models are software products such as software requirements specifications and software design descriptions that can be run on a computer and produce an answer that satisfies the requirements or design. A simple executable model tool is a compiler that can convert source code to executable code.

3.4 Construction Tools

A *programming tool or software development* tool (i.e., a construction tool) is a program or application that software developers use to create, debug, maintain, or otherwise support other programs and applications.

3.4.1 Development environments. A *software development environment* (SDE) is the entire environment (applications, servers, network) that provides comprehensive facilities to computer programmers for software development.

Typically an SDE is dedicated to a specific programming language, allowing a feature set that most closely matches the programming paradigms of the language. SDEs typically present a single program in which all development is done. This program typically provides many features for authoring, modifying, compiling, deploying and debugging software. The aim is to abstract the configuration necessary to piece together command line utilities in a cohesive unit, which theoretically reduces the time to learn a language, and increases developer productivity. It is also thought that the tight integration of development tasks can further increase productivity [http://en.wikipedia.org/wiki /Software _development _environment].

An SDE can contain but is not limited to such tools as:

- Requirements management tools
- Design modeling tools
- Documentation generation tools
- Integrated development environment (IDE)
- Code analysis tools
- Code referencing tools
- Code inspection tools
- Software building tools (compile, link)
- Source repository (configuration management)
- Problem reporting / tracking tools

3.4.2 GUI builder. A *graphical user interface (GUI) builder*, also known as GUI designer, is a software development tool that simplifies the creation of GUIs by allowing the designer to arrange widgets using a drag-and-drop WYSIWYG (What You See Is What You Get) editor. Without a GUI builder, a GUI must be built by manually specifying each widget's parameters in code, with no visual feedback until the program is run.

User interfaces are commonly programmed using an event-driven architecture, so GUI builders also simplify creating event-driven code. This supporting code connects widgets with the outgoing and incoming events that trigger the functions providing the application logic.

A *widget* (or *control*) is an element of a GUI that displays an information arrangement changeable by the user, such as a window or a text box [http://en.wikipedia.org/wiki/Graphical _user_interface_builder].

3.4.3 Unit testing tools. *Unit testing* is a method by which individual units of source code are tested to determine if they are fit for use. A *unit* is the smallest testable part of an application. In procedural programming a unit may be an individual function or procedure. Unit tests are created by programmers or occasionally by white box testers.

White-box testing (a.k.a. clear box testing, glass box testing, transparent box testing, or structural testing) is a method of testing software that tests internal structures or workings of an application, as opposed to its functionality (i.e. black-box testing). In white-box testing an internal perspective of the system, as well as programming skills, are required and used to design test cases. The *white-box tester* chooses inputs to exercise paths through the code and determine the appropriate outputs [http://en.wikipedia.org/wiki/White-box_testing].

Ideally, each test case is independent from the others: substitutes like method stubs, mock objects, fakes and test harnesses can be used to assist testing a module in isolation. Unit tests are typically written and run by software developers who developed the code to ensure that code meets its design and behaves as intended. Its implementation can vary from being very manual (pencil and paper) to being formalized as part of build automation.

3.4.4 Application oriented languages. An *application oriented language* is a computer language whose statements resemble terminology of the user [http://www.thefreedictionary.com /application-oriented+language].

Some examples are:

- **Scripting language** — A *scripting language* is a programming language that allows control of one or more software applications. "Scripts" are distinct from the core code of the application, as they are usually written in a different language and are often created or at least modified by the end-user. Scripts are often interpreted from source code or byte code, whereas application software is typically first compiled to a native machine code or to an intermediate code [http://en.wikipedia.org/wiki/Scripting_language].

- **Visual language** — An image that communicates an idea presupposes the use of a visual language. Just as people can "verbalize" their thinking, they can "visualize" it. A diagram, a map, and a painting are all examples of uses of visual language. Its structural units include line, shape, color, form, motion, texture, pattern, direction, orientation, scale, angle, space and proportion.

  The elements in an image represent concepts in a spatial context, rather than the linear form used for words. Speech and visual communication are parallel and often interdependent means by which humans exchange information [http://en.wikipedia.org/wiki /Visual_language].

- **Domain-specific language** — A domain-specific language (DSL) is a programming language or specification language dedicated to a particular problem domain, a particular problem representation technique, and/or a particular solution technique. The concept isn't new—*special-purpose programming languages* and all kinds of modeling/specification languages have always existed, but the term has become more popular due to the rise of domain-specific modeling [http://en.wikipedia.org/wiki/Domain_specific_language].

- **Markup language** — A *markup language* is a modern system for annotating a text in a way that is syntactically distinguishable from that text. The idea and terminology evolved from the "*marking up*" of manuscripts, i.e. the revision instructions by editors, traditionally written with a blue pencil on authors' manuscripts. Examples are typesetting instructions such as those found in *troff* and *Latex*, and structural markers such as XML tags. Markup is typically omitted from the version of the text that is displayed for end-user consumption [http://en.wikipedia.org/wiki/Markup_language].
- **Macros** — A macro is a rule or pattern that specifies how a certain input sequence (often a sequence of characters) should be mapped to an output sequence (also often a sequence of characters) according to a defined procedure. The mapping processes that instantiates (transforms) a macro into a specific output sequence is known as *macro expansion* [http://en.wikipedia.org/wiki/Macro_(computer_science)].

3.4.5 Profiling. In software engineering, *software profiling* or simply *profiling*, a form of dynamic program analysis (as opposed to static code analysis), is the investigation of a program's behavior using information gathered as the program executes. The usual purpose of this analysis is to determine that sections of a program to optimize - to increase its overall speed, decrease its memory requirement or sometimes both.

For example, a *code profiler* is a performance analysis tool that, most commonly, measures only the frequency and duration of function calls, but there are other specific types of profilers (e.g. memory profilers) in addition to more comprehensive profilers, capable of gathering extensive performance data [http://en.wikipedia.org/wiki/Software_profiling].

3.4.6 Performance analysis. *Performance analysis* involves gathering formal and informal data to help customers and sponsors define and achieve their performance goals. Performance analysis uncovers several perspectives on a problem or opportunity, determining any and all drivers towards or barriers to successful performance, and proposing a solution system based on what is discovered.

3.4.7 Slicing tools. In computer programming, *program slicing* is the computation of the set of programs statements, the program slice that may affect the values at some point of interest, referred to as a *slicing criterion*. Program slicing can be used in debugging to locate source of errors more easily. Other applications of slicing include software maintenance, optimization, program analysis, and information flow control [http://en.wikipedia.org/wiki/Program_slicing].

3.5 Construction Technologies

Technology can be most broadly defined as the entities, both material and immaterial, created by the application of mental and physical effort in order to achieve some value. In this usage, technology refers to tools and machines that may be used to solve real-world problems. It is a far-reaching term that may include simple tools, such as a crowbar or wooden spoon, or more complex machines, such as a space station or particle accelerator. Tools and machines need not be material; virtual technology, such as computer software and business methods, falls under this definition of technology.

The word "technology" can also be used to refer to a collection of techniques. In this context, it is the current state of humanity's knowledge of how to combine resources to produce desired products, to solve problems, fulfill needs, or satisfies wants; it includes technical methods, skills, processes, techniques, tools and raw materials. When combined with another term, such as

"medical technology" or "space technology," it refers to the state of the respective field's knowledge and tools. "State-of-the-art technology" refers to the high technology available to humanity in any field [http://en.wikipedia.org/wiki/Technology].

3.5.1 API design and use. An API (application programming interface) is a language and message format used by an application program to communicate with the operating system or some other control program such as a database management system. An API implies that some program module is available in the computer to perform the operation or that it must be linked into the existing program to perform the tasks [PC Magazine Encyclopedia].

3.5.2 Code reuse and libraries. *Code reuse*, also called *software reuse*, is the use of existing software, or software knowledge, to build new software. Code reuse is the idea that a partial or complete computer program written at one time can be. should be, or is being used in another program written at a later time. The reuse of programming code is a common technique that attempts to save time and energy by reducing redundant work.

The software library is a good example of code reuse. Programmers may decide to create internal abstractions so that certain parts of their program can be reused, or may create custom libraries for their own use. Some characteristics that make software more easily reusable are modularity, loose coupling, high cohesion, information hiding and separation of concerns [http://en .wikipedia.org/wiki/Code_reuse].

3.5.3 Object-oriented run-time issues. *Object-oriented programming (OOP) is a* programming paradigm that uses "objects" – data structures consisting of data and methods together with their interactions – to design applications and computer programs. Programming techniques may include features such as data abstraction, encapsulation. modularity, polymorphism, and inheritance. Many modern programming languages now support OOP [http://en.wikipediaorg/wiki /Object_oriented_programming].

- ***Data abstraction*** — *Abstraction* is the process by which data and programs are defined with a representation similar in form to its meaning (semantics), while hiding away the implementation details.
- ***Encapsulation*** — *Encapsulation* is used to refer to one of two related but distinct notions, and sometimes to the combination: (1) a language mechanism for restricting access to some of the object's components, and (2) a language construct that facilitates the bundling of data with the methods (or other functions) operating on that data.
- ***Modularity*** — Also known as m*odular programming* (also known as top down design and stepwise refinement) is a software design technique that increases the extent to which software is composed of separate, interchangeable components by breaking down program functions into *modules*, each of which accomplishes one function and contains everything necessary to accomplish this function.
- ***Polymorphism*** — In computer science, *polymorphism* is a programming language feature that allows values of different data types to be handled using a uniform interface. The concept of parametric polymorphism applies to both data types and functions. A function that can evaluate to or be applied to values of different types is known as a *polymorphic function.* A data type that can appear to be of a generalized type (e.g., a list with elements of arbitrary type) is designated *polymorphic data type* like the generalized type from which specializations are made [htpp://en.wikipedia.org/wiki/Type_polymorphism].

- ***Inheritance*** — In object-oriented software engineering, inheritance is the capability for classes to inherit attributes from pre-existing classes (called base classes, superclasses, parent classes or ancestor classes). The resulting classes are known as derived classes, subclasses or child classes. The relationship of classes through inheritance gives rise to a hierarchy. By default, the subclass inherits all the attributes and properties of the superclass but usually they may be redefined in the subclass if desired. Inheritance may apply to objects as well as to classes. Note: this is a general definition and individual languages may have specific inheritance definitions that differ [http://en.wikipedia.org/wiki/Inheritance_(object-oriented.programming); Sherman 2012].

- ***Dynamic binding*** — In object-oriented programming, *dynamic binding or late binding* means determining the exact implementation of a request based on both the request (operation) name and the receiving object at run-time. It often happens when invoking a derived class's member function using a pointer to its base class. The implementation of the derived class will be invoked instead of that of the base class. It allows substituting a particular implementation using the same interface and enables polymorphism [http://www.ask.com/wiki/Dynamic_binding_(computer_science)].

3.5.4 Parameterization and generics. A *parameter* is a special kind of variable, used in a subroutine to refer to one of the pieces of data provided as input to the subroutine. These pieces of data are called *arguments*. An ordered list of parameters is usually included in the definition of a subroutine, so that, each time the subroutine is called, its arguments for that call can be assigned to the corresponding parameters. The term "argument" is often (incorrectly) used in place of "parameter," [http://en.wikipedia.org/wiki/Parameter_(computer_programming)].

Two types of parameters are frequently used — dependent and independent variables. The *independent variable* is typically the variable representing the value being manipulated or *changed and the dependent variable is the observed result of the independent variable being manipulated. In Boehm's equations on software costs, the size of the computer program was the* independent and the cost of developing the software was the dependent variable [Boehm 1981].

Generic programming is a style of computer programming in which algorithms are written in terms of *to-be-specified-later* types that are then *instantiated* when needed for specific types provided as parameters. This approach, pioneered by Ada in 1983, permits writing common functions or types that differ only in the set of types on which they operate when used, thus reducing duplication. Software entities created using generic programming are known as generics [http:// en.wikipedia.org/wiki/Generic programming].

3.5.5 Assertions. In computer programming, an *assertion* is a predicate (for example a true–false statement) placed in a program to indicate that the developer *thinks* that the predicate is always true at that place [http://en.wikipedia.org/wiki/Assertion_(computing)].

- ***Design by contract***™ — *Design by Contract* or *Programming by Contract* is an approach to designing computer software. It prescribes that software designers should define formal, precise and verifiable interface specifications for software components, which extend the ordinary definition of abstract data types with preconditions, postconditions and invariants. These specifications are referred to as "contracts," in accordance with a conceptual metaphor with the conditions and obligations of business contracts. The term was coined by Bertrand Meyer in connection with his design of the Eiffel programming language [http://en.wikipedia.org/wiki/Design_by_contract].

- ***Defensive programming*** — Defensive programming is a form of defensive design intended to ensure the continuing function of a piece of software in spite of unforeseeable usage of said software. The idea can be viewed as reducing or eliminating the prospect of Murphy's Law having effect. Defensive programming techniques are used especially when a piece of software could be misused mischievously or inadvertently to catastrophic effect [http://en.wikipedia.org/wiki/Defensive_programming].

3.5.6 Error handling. Error handling refers to the programming practice of anticipating and coding for error conditions that may arise when your program runs. Errors in general come in three flavors: compiler errors such as undeclared variables that prevent your code from compiling; user data entry error such as a user entering a negative value where only a positive number is acceptable; and run time errors, that occur when your program cannot correctly execute a program statement [http://www.c.com/excel/errorhandling.htm].

For example:

- ***Exception handling*** — *Exception handling* is a programming language construct or computer hardware mechanism designed to handle the occurrence of exceptions, special conditions that change the normal flow of program execution. Programming languages differ considerably in their support for exception handling (as distinct from error checking, which is normal program flow that codes for responses to adverse contingencies such as invalid state changes or the unsuccessful termination of invoked operations) [http://en.wikipedia.org/wiki/Exception_handling].

- ***Fault tolerance*** — Fault-tolerance (a.k.a. graceful degradation) is the property that enables a computer system to continue operating properly in the event of the failure of (or one or more faults within) some of its components. If its operating quality decreases at all, the decrease is proportional to the severity of the failure, as compared to a naïvely-designed system in which even a small failure can cause total breakdown. Fault-tolerance is particularly sought-after in military systems or life-critical systems [http://en.wikipedia.org/wiki /Fault-tolerant system].

3.5.7 State-based and table-driven construction techniques. *State-based construction techniques* are most commonly represented by *state diagrams* which are also referred to as *state transition diagrams*. A state diagram is a directed graph in which each vertex represents a state and each edge represents a transition between two states.

A state transition table presents a common representation of Finite State Machines (FSMs). (See Figure 3.1) Every column included in the table corresponds to a state. Each row corresponds to an event category. Values contained in table cells provide states resulting from respective transitions. Table cells also can be used for specifying actions related to transitions.

A *finite-state machine* (FSM) or finite-state automaton (plural: *automata*), or simply a state machine, is a mathematical abstraction sometimes used to design digital logic or computer programs. It is a behavior model composed of a finite number of states, transitions between those states, and actions, similar to a flow graph in which one can inspect the way logic runs when certain conditions are met. It has finite internal memory, an input feature that reads symbols in a sequence, one at a time without going backward; and an output feature, which may be in the form of a user interface, once the model is implemented. The operation of an FSM begins from one of the states (called a *start state*), goes through transitions depending on input to different

states and can end in any of those available, however only a certain set of states mark a successful flow of operation (called *accept states*) [http://en.wikipedia.org/wiki/Finite-state_machine].

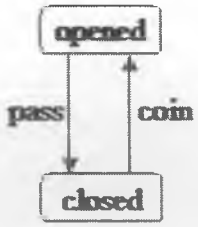
3.5.8 Run-time configuration and internationalization. These items are defined as follows:

3.5.8.1 Run-time configuration. Using run-time configuration lets you create and delete data services, adapters, and destinations, even after the server has been started.

There are many reasons why you might want to create components dynamically. For example, consider the following use cases:

- You want a separate destination for each of your offices that use an application. Instead of manually creating destinations in the configuration files, you want to create them dynamically based on information in a database.
- You want to dynamically create, delete, or modify destinations in response to some user input.

There are two primary ways to perform dynamic configuration. The first way is to use a custom bootstrap service class. This is the preferred way to perform dynamic configuration. The second way is to call a remote object on the server that performs dynamic configuration.



	opened	closed
pass	closed	
Coin		opened

Figure 3.1: State transition tables [Sakharov 2005]

3.5.8.2 Internationalization. In computing, *internationalization* and *localization* are means of adapting computer software to different languages, regional differences and technical requirements of a target market. *Internationalization* is the process of designing a software application so that it can be adapted to various languages and regions without engineering changes. *Localization* is the process of adapting internationalized software for a specific region or language by adding locale-specific components and translating text. Some companies use the term "globalization" for the combination of internationalization and localization [http://en.wikipedia.org/wiki/Internationalization_and_localization].

This concept is also known as NLS (National Language Support or Native Language Support).

3.5.9 Grammar-based input processing (parsing). *Parsing* is to break down (a sentence) into its component parts of speech with an explanation of the form, function, and syntactical relationship of each part.

3.5.10 Concurrency primitives. In computer science, *concurrency* is a property of systems in which several computations are executing simultaneously, and potentially interacting with each other. The computations may be executing on multiple cores in the same chip, preemptively time-shared threads on the same processor, or executed on physically separated processors.

Concurrency primitives are a series of processes that can be used to develop a computer program. Examples are:

- ***Semaphores*** — In computer science, *a semaphore* is a protected variable or abstract data type that provides a simple but useful abstraction for controlling access by multiple processes to a common resource in a parallel programming environment.

  A useful way to think of a semaphore is as a record of how many units of a particular resource are available, coupled with operations to *safely* (i.e. without race conditions) adjust that record as units are required or become free, and if necessary wait until a unit of the resource becomes available. Semaphores are a useful tool in the prevention of race conditions and deadlocks; however their use is by no means a guarantee that a program is free from these problems. Semaphores that allow an arbitrary resource count are called counting semaphores, whilst semaphores that are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called binary semaphores [http://en.wikipedia.org /wiki/Semaphore_(programming)].

- ***Monitors*** — In concurrent programming, a *monitor* is an object or module intended to be used safely by more than one thread. The defining characteristic of a monitor is that its methods are executed with mutual exclusion. That is, at each point in time, at most one thread may be executing any of its methods. This mutual exclusion greatly simplifies reasoning about the implementation of monitors compared with code that may be executed in parallel.

  Monitors also provide a mechanism for threads to temporarily give up exclusive access, in order to wait for some condition to be met, before regaining exclusive access and resuming their task. Monitors also have a mechanism for signaling other threads that such conditions have been met [http://en.wikipedia.org/wiki/Monitor_(synchronization)].

3.5.11 Middleware (components and containers). *Middleware* is used to describe a broad array of tools and data that help applications use networked resources and services. Some tools. such as authentication and directories, are in all categorizations. Other services, such as cost scheduling of networked resources, secure multicast, and object brokering and messaging, are the major middleware interests of particular communities, such as scientific researchers or business systems vendors. One definition that reflects this breadth of meaning is "Middleware is the intersection of the stuff that network engineers don't want to do with the stuff those applications developers don't want to do" [http://middleware.internet2.edu/overview/middleware-faq.html].

- ***Components*** — The *components* of middleware are an executable unit of functionality. One can buy or download it, deploy it, and it works. It is a software black box.

- ***Containers*** — *Containers* are used in application servers to plug components into application servers [The Internet Encyclopedia, p. 611].

3.5.12 Construction methods for distributed software. The following steps might be used to initiate construction approaches for developing a customer-oriented distributed software system (DSS):

1. Draft a requirements specification for the DSS.
2. Establish with the potential user a draft front-end interface.

3. Inventory the technical capabilities of the software engineering development team.
4. Schedule training for gaps in the technical knowledge of the assigned software engineers.
5. Estimate the schedule and cost for developing the system.
6. Inventory the existing system to determine what part of the existing system can be re-used.
7. Establish the degree of authority that the project management, the user, and sponsor have over the project.
8. Develop a prototype system. Reanalyze the requirements and user interface.
9. Look again at the requirements. Are they realistic?
10. Is the budget realistic?
11. Do the benefits outweigh the costs and potential problems?
12. Start the project.

3.5.13 Constructing heterogeneous systems (hardware and software). In information technology heterogeneity means a network comprising different types of computers, potentially with vastly differing memory sizes, processing power and even basic underlying architecture, or a data resource with multiple types of formats [http://en.wikipedia.org/wiki/Homogeneity_and_heterogeneity].

3.5.14 Hardware-software co-design. Current methods for designing embedded systems require hardware and software to be specified and designed separately. A specification, often incomplete and written in non-formal languages, is developed and sent to the hardware and software engineers. Hardware-software partition is decided *a priori* and is adhered to as much as is possible, because any changes in this partition may necessitate extensive redesign. Designers often strive to make everything fit in software, and off-load only some parts of the design to hardware to meet timing constraints [Pederson 2011].

Lockheed Martin [2006] defines co-design as a simultaneous consideration of hardware and software within the design process. It emphasizes that it consists of the "co-development and co-verification of hardware and software through the use of simulation and/or emulation."

Co-design includes [Assimakopoulos 1998]:

- Co-specification, where the roles of software and hardware in implementing system functionality are considered and, based on the evaluation, the implementation is assigned to either of the two.
- Co-development, where the software, hardware and interfaces are developed.
- Co-verification to further optimize and refine the SW/HW partitioning, i.e. to aid design space exploration.
- Co-management that covers coordination, project management, requirements management and configuration management throughout system specification, development and verification.

3.5.15 Performance analysis and tuning. *Performance analysis*, commonly known as *profiling*, is the investigation of a program's behavior using information gathered as the program executes. Its goal is to determine which sections of a program to optimize.

A *profiler* is a performance analysis tool that measures the behavior of a program as it executes, particularly the frequency and duration of function calls. Performance analysis tools existed at least from the early 1970s. Profilers may be classified according to their output types, or their methods for data gathering.

Tuning (i.e., *code tuning*) is the practice of modifying correct code in ways that make it run more efficiently. "Tuning" refers to small-scale changes that affect a single class, a single routine, or, more commonly, a few lines of code. "Tuning" does not refer to large-scale design changes or other higher-level means of improving performance. There is an argument that "code tuning" can make dramatic improvements at each level from system design through code tuning. Jon Bentley [1982] cites an argument that in some systems the improvements at each level can be multiplied by a factor of 10. Because you can achieve a 10-fold improvement in each of six levels, that implies a potential performance improvement of a million fold. Although such a multiplication of improvements requires a program in which gains at one level are independent of gains at other levels, which is rare, the potential is inspiring.

However, code tuning is not the most effective way to improve performance—program architecture, class design, and algorithm selection usually produce more dramatic improvements. Nor is it the easiest way to improve performance—buying new hardware or a compiler with a better optimizer is easier. And it's not the cheapest way to improve performance either—it takes more time to hand-tune code initially, and hand-tuned code is harder to maintain later.

3.5.16 Platform standards (Posix, etc.). A *computing platform* is some sort of hardware architecture and software framework (including application frameworks) that allows software to run. Typical platforms include a computer's architecture, operating system, programming languages and related user interface (runtime libraries or graphical user interface).

A platform is a crucial element in software development. A platform might be simply defined as a place to launch software. It is an agreement that the platform provider gave to the software developer that logic code will interpret consistently as long as the platform is running on top of other platforms. Logic code includes byte code, source code, and machine code. It actually means execution of the program is not restricted by the type of operating system provided. It has mostly replaced the machine independent languages [http://en.wikipedia.org/wiki/Computing_platform].

POSIX (Portable Operating System Interface) is a family of standards, specified by the IEEE, to clarify and make uniform the application programming interfaces (and ancillary issues, such as command line shell utilities) provided by Unix-like operating systems. When you write your programs to rely on POSIX standards, you can be pretty sure to be able to port them easily among a large family of UNIX derivatives (including Linux, but not limited to it!) [Martelli 2009].

Java refers to a number of computer software products and specifications from Sun Microsystems, a subsidiary of Oracle Corporation, that together provide a system for developing application software and deploying it in a *cross-platform environment*. (In order for software to be considered *cross-platform,* it must be able to function on more than one computer architecture

or operating system.) *Java* is used in a wide variety of *computing platforms* from embedded devices and mobile phones on the low end, to enterprise servers and supercomputers on the high end.

The *Java platform* is the name for a bundle of related programs from Sun which allow for developing and running programs written in the Java programming language. The Java platform is not specific to any one processor or operating system, but rather an execution engine (called a virtual machine) and a compiler with a set of libraries that are implemented for various hardware and operating systems so that Java programs can run identically on all of them [http://en .wikipedia.org/wiki/Java_(software_platform)]. Note the distinction between Java as a language (Section 3.3.2.2.3) and as a platform.

3.5.17 Test-first programming. Don Wells (consultant, Extreme Programming) states that *"When you create your tests first, before the code, you will find it much easier and faster to create your code."* The combined time it takes to create a unit test and create some code to make it pass is about the same as just coding it up straight away. If you already have the unit tests you don't need to create them after the code saving you some time now and lots later. Creating a unit test helps a developer to really consider what needs to be done. There can be no misunderstanding a specification written in the form of executable code" [http://www.extremeprogramminging .org/rules/testfirst.html].

References:

Additional information on the *software construction* KA can be found in the following documents:

- **[Assimakopoulos 1998]** N.A. Assimakopoulos, "Systemic industrial management of HW/SW co-design," *The Journal of High Technology Management Research*, Vol. 9, No. 2, 1998, pp. 271–284.
- **[Bentley 1982]** Jon Bentley, *Writing Efficient Programs,* Prentice-Hall, Englewood Cliffs, NJ, 1982.
- **[Boehm 1981]** Barry W. Boehm. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ, 1981.
- **[Clements et al. 2002]** Paul Clements, Felix Bachmann, Len Bass, David Garlan, Paulo Merson, James Ivers, Reed Little, Robert Nord, and Judith Stafford, *Documenting Software Architectures: Views and Beyond* (Hardcover), Pearson, Boston, 2002, 560 pages, ISBN-13: 978-0201703726. (Recommended as a reference book by the IEEE Computer Society).
- **[IEEE Standard 1028-2008]** IEEE Standard 1028, *Standard for Software Reviews*, IEEE Inc., 2008.
- **[Lockheed Martin 2006]** Lockheed Martin, "Hardware/software co-design," http:// www.atl.lmco.com/projects/rassp/RASSP_legacy/appnotes/HWSW/APNOTE_HWSW _INDEX.HTM, 2006.
- **[Martelli 2009]** Alex Martelli, http://stackoverflow.com/questions/1780599/i-never-really–understood-what-is-posix, 2009.

- **[McConnell 2004]** Steve McConnell, *Code Complete*, (Paperback), 2nd Edition, revised, Microsoft Press, 2004, 960 pages, ISBN-13: 978-0735619678. (Recommended as a reference book by the IEEE Computer Society.)
- **[Null & Lobur 2006]** Linda Null and Julia Lobur, *The Essentials of Computer Organization and Architecture* (Hardcover), 2nd Edition, Jones & Bartlett, 2006, ISBN-13: 978-763737696, Chapters 1-4, 9-12, also sections 8.1-8.4, 8.6, 8.7. (Recommended as a reference book by the IEEE Computer Society.)
- **[PC Magazine Encyclopedia]** *PC Magazine Encyclopedia,* Ziff Davis. Inc., New York, 1996.
- **[Pederson 2011]** Donald O. Pederson, "A Framework for Hardware-Software Co-Design of Embedded Systems. UCB Electronic Systems Design Publications, University of California, Berkeley, CA, 2011.
- **[Sakharov 2005]** Finite State Machines, http://sakharov.net/fsmtutorial.html, 2005.
- **[Sherman 2012]** Dr. S. Sherman, Pers. comm., 2012.
- **[Silberschatz, Galvin, & Gagne 2008]** J. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, *Operating System Concepts* (Hardcover), 8th Edition, John Wiley, Hoboken, NJ, 2008, ISBN-13: 978-0470128725, Chapter 3-6, 16, 18. (Recommended as a reference book by the IEEE Computer Society.)
- **[The Internet Encyclopedia 2003]** Hossein Bidgoli, "*The Internet Encyclopedia*," Volume 1, John Wiley, Hoboken, NJ. 2003.
- **[SWEBOK 2004]** E. Bourque and R. Dupuis, Editors, *Software Engineering Body of Knowledge (SWEBOK)*. IEEE Computer Society Press, Los Alamitos, CA, 2004.
- **[Webster 1913]** Webster's Revised Unabridged Dictionary, G & C. Merriam Co., Springfield, MA, 1913.
- **[Wikipedia]** Wikipedia is a free web based encyclopedia enabling multiple users to freely add and edit online content. Definitions cited on Wikipedia and their related sources have been verified by the authors and other peer reviewers.

Chapter 4.1

Software Testing
Fundamentals, techniques and related concepts

Antonia Bertolino and Eda Marchetti
Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo"
Consiglio Nazionale delle Ricerche,
Pisa, Italy

Abstract—*Nowadays, a common consideration is that testing is a fundamental and effort-consuming activity of the software development process, influencing the quality and reliability of the released products. Performing software testing does not mean only the detection of "bugs" in the software, but also assuring the necessary confidence in the functioning of the product developed and assessing its properties.*

Thus the various steps of a testing process, which evolves all along in parallel with the entire development process, need to be specified. In this chapter we provide a survey of the fundamental concepts of the software testing discipline, focusing in particular on criteria and techniques, test levels, process, measurements and tools. Due to the vastness of the topic and the impossibility to be all-embracing, we try to highlight the most important concepts and approaches for each covered subject, and provide plenty of references for further reading.

Index Terms — *D.2.4 Software/Program Verification, D.2.5 Testing and Debugging.1.*

1. Introduction

Testing is a crucial activity all along software development. Whereas several development process models exist which position it in different stages, by no means are testing activities to be treated as a last-minute concern. Considering a traditional waterfall process, testing should start at the requirements specification stage, while planning ahead for test strategies and procedures, and propagate down, with derivation and refinement of test cases, all along the various development steps after the code-level stage, at which time the test cases are executed. It should continue even after deployment, with logging and analysis of operational usage data and customer's reported failures. In the more recent Test-Driven Development approach, testing is moved ahead with tests written and executed before any specification or coding begins.

Testing is an expensive and challenging activity that involves several high-demanding tasks: at the forefront is the task of deriving an adequate suite of test cases, according to a feasible and cost-effective test selection technique. However, test selection is just a starting point, and many other critical tasks face test practitioners with technical and conceptual difficulties: the ability to launch the selected tests (in a controlled host environment, or worse in the tight target environment of an embedded system); deciding whether the test outcome is acceptable or not (which is referred to as the *test oracle* problem); if not, evaluating the impact of the failure and finding its direct cause (the fault), and the indirect one (via Root Cause Analysis); judging whether testing is sufficient and can be stopped, which in turn would require having at hand measures of the effectiveness of the tests. Each one of the above tasks presents tough challenges both to testing

practitioners, whose skill and expertise always remain of topmost importance, and to testing researchers, who in four decades of the software testing discipline [11] have gained great advances in test automation and formalization.

The *Guide to the Software Engineering Body of Knowledge* (SWEBOK) [14] provides a compendium of the generally accepted knowledge in software engineering, divided into 10 *knowledge areas* (KA). Among them, the Software Testing KA summarizes basic concepts and includes a detailed reference list. A roadmap of achievements and open challenges for the software testing research discipline with an extensive bibliography can also be found in [10].

In this document, we provide a broad overview of the current state-of-art of the software testing discipline, spanning test levels, test techniques, including usability testing, test-related measures, test process and supporting tools. In an attempt to cover all these testing-related topics, we can only briefly expand on each argument, and provide references throughout for further reading.

The remainder of the document is organized as follows: we present some basic concepts in Section 2, and an overview of V&V approaches (static and dynamic) in Section 3. In Section 4, we focus on the test levels (unit, integration and system test), the role of regression testing and the objectives of testing. In Section 5, we list the most commonly adopted techniques for test selection. Section 6 is dedicated to the aspects of usefulness and usability, while Section 7 highlights test measurements that can be adopted during the software lifecycle. Going on, in Section 8, we present the test process, which includes test planning, design, execution and documentation, and then in Section 9 we summarise management concerns. Finally, a survey of testing tools is discussed in Section 10 and conclusions are drawn in Section 11.

2. Terminology and Basic Concepts

We provide in this section some introductory notions on testing-related terminology and key issues.

2.1 Foreword

Generally speaking, test techniques can be divided into two classes:

- *Static analysis techniques* (expanded in Section 3.1), where the term "static" does not refer to the techniques themselves (they could use automated analysis tools), but is used to mean that they do not involve the execution of the system under test (SUT). Static techniques are applicable throughout the lifecycle to the various developed artifacts for different purposes, such as to check the adherence of the implementation to the specifications or to detect flaws in the code via inspection or review.
- *Dynamic analysis techniques* (further discussed in Section 3.2), which exercise the SUT in order to expose possible failures and to observe the behavioral and non-functional properties of the program.

  Static and dynamic analysis are complementary techniques [4]: the former yield generally valid results, but they may be weak in precision; the latter are efficient and provide more precise results, but only holding for the examined executions. The focus of this chapter is mainly on dynamic test techniques, and, where not otherwise specified, testing is meant to be synonymous with "dynamic testing."

2.2 A General Definition

For a general definition of software testing, we refer to the definition provided in the SWEBOK [14]:

> *Software testing consists of the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the specified expected behavior.*

This short definition attempts to include all essential testing concerns: the term *dynamic* means, as previously stated, that testing implies executing the program on real numbers (as opposed to symbolic inputs); *finite* indicates that only a limited number of test cases can be executed during the testing phase, chosen from the whole test set, that can generally be considered infinite; *selected* refers to the test techniques adopted for selecting the test cases (and testers must be aware that different selection criteria may yield vastly different effectiveness); *expected* points to the decision process (see Test Oracles below) adopted for establishing whether the observed outcomes of program execution are acceptable or not.

2.3 Fault vs. Failure

To fully understand the facets of software testing, it is important to clarify the terms "fault," "error"[9] and "failure:" indeed, although their meanings are strictly related, there are important distinctions between these three concepts.

A *failure* is the manifested inability of the program to perform the function required, i.e., a system malfunction evidenced by incorrect output, abnormal termination or unmet time and space constraints. The cause of a failure, e.g., a missing or incorrect piece of code, is a *fault*. A fault may remain undetected for a long time, until some event activates it. When this happens, it first brings the program into an intermediate unstable state, called *error,* which, if and when it propagates to the output, eventually causes the failure. The process of failure manifestation can be therefore summed up as a chain [44]:

$$\text{Fault} \rightarrow \text{Error} \rightarrow \text{Failure}$$

which can recursively iterate: *a fault in turn can be caused by the failure of some other interacting system.*

What testing reveals are the failures, and a consequent analysis stage (fault localization) is needed to identify the faults that caused them.

The notion of a fault, however, is ambiguous and difficult to grasp, because no precise criteria exist to definitively determine the cause of an observed failure. It would be more practical to speak about *failure-causing inputs*, that is, those sets of inputs that when exercised can result into a failure, since they can be unambiguously identified.

2.4 Test Criteria

A decision procedure stating what a suitable set of test cases should be is called a *test criterion.* A more precise definition for a test criterion is provided below [14]:

---

9. Note that we are using the term "error" with the commonly used meaning within the Software Dependability community, which is stricter than its general definition in [28].

A test criterion C is a decision predicate defined on triples (P, RM, T), where P is a program, RM is a reference model related to P, and T is a test suite. When C (P, RM, and T) holds, it is said that T satisfies criterion C for P and RM.

In particular, a test criterion used for selecting the test cases is said to be a *test selection* criterion, whereas if used for checking whether a selected test suite is adequate, that is, to decide whether the testing can be stopped, it is said to be a *test adequacy* or a stopping criterion.

Testers can use the same test criterion for guiding in a proactive way the selection of test cases (so that when the selection terminates, the criterion is automatically fulfilled), or for checking after the fact if the executed (and however else selected) suite is sufficient. For instance, a tester could execute a test suite manually derived from the analysis of the requirements specification document, and use a coverage analyzer tool during test execution for measuring the percentage of program branches covered, stopping the testing as soon as this percentage reaches a fixed threshold.

A broad class of test criteria is referred to as *partition testing*. The underlying idea is that the program input domain is divided into subdomains within which it is assumed that the program behaves the same, i.e., for every point within a subdomain the program either succeeds or fails: we also call this the "test hypothesis." Therefore, thanks to this assumption, only one or a few points within each subdomain need to be checked, and this is what allows for getting a finite set of tests out of the potentially infinite domain.

Hence a partition testing criterion essentially provides a way to derive the subdomains. It is contrasted with *random testing* ([19], [20]), by which test inputs are blindly drawn from the entire domain. (Random testing is briefly discussed in Section 5.3.)

A test criterion yielding the assumption that all test cases within a subdomain either succeed or fail is only an ideal, and would guarantee that any fulfilling set of test cases always detects the same failures; in practice, the assumption is rarely satisfied, and different sets of test cases fulfilling the same criterion may show varying effectiveness depending on how the test cases are picked within each subdomain.

With reference to the above definition for a test criterion, the partitioning of the program input domain into subdomains is induced by the adopted reference model (RM). Test criteria can be classified according to the kind of *RM*: it can be as informal as "tester intuition," or strictly formalized, as in the case of conformance testing from a formal specification or of code-coverage criteria. The advantages of a formalized *RM* are evident: the selection of test cases, or the adequacy evaluation of test case adequacy, can be automated.

2.5 About Testing Effectiveness

Given that test resources are limited, how the test cases are selected or decided is of crucial importance. Indeed, effective testing requires strategies to trade-off between the two opposing needs of amplifying testing thoroughness on the one side (for which a high number of test cases would be desirable), and reducing times and costs on the other (for which the fewer the test cases the better).

As we will see in the remainder of this chapter, there exist many types of testing and many test strategies; however, all of them share the same ultimate purpose: increasing the software engineer's confidence in the proper functioning of the software. Towards this general goal, a piece of software can be tested to achieve various more direct objectives, all meant in fact to

increase confidence, such as exposing potential design flaws or deviations from user's requirements, measuring the operational reliability, evaluating the performance characteristics, and so on (we further expand on test objectives in Section 4.5).

To serve each specific objective, different techniques can be adopted. However, it is only in light of the objective pursued that the effectiveness of the test set can be evaluated.

Hence, if there are many factors of relevance when a test selection criterion has to be chosen, an important point to always keep in mind is that what makes a test a "good" one does not have a unique answer, but varies depending on the context, on the specific application, and on the goal for testing.

The most common interpretation for "good" would be "able to detect many failures." For example, in testing for defect identification, a successful test is obviously one that causes the system to fail. However, again precision would require specifying what kinds of failures are most undesirable, as it is well known and experimentally observed that different test criteria trigger different types of faults ([8], [63]). For this reason, it is always preferable to spend the test budget to apply a combination of diverse techniques rather than concentrating on just one, even if it is shown to be the most effective.

Testing for identification of defects is quite different from testing to demonstrate that the software meets its specifications or other desired properties, in which case testing is successful if no (significant) failures are observed.

Whatever the test objective, reducing the testing cost remains a crucial concern. Two main approaches are taken to control the test set dimension: *test suite minimization*, which, given a test suite T, attempts to identify a reduced test suite T' that yields the same properties (e.g., in terms of code coverage) of T; and *test suite prioritization*, which orders the tests in T according to some criteria, such as fault-detection effectiveness.

2.6 Test Oracles

An important component of testing is the *oracle*. Indeed, a test is meaningful only if it is possible to decide about its outcome. The difficulties inherent in this task, often oversimplified, had been articulated early on in [62].

An oracle is any (human or mechanical) agent that decides whether the program behaved correctly on a given test. The oracle is specified to output a *reject* verdict if it observes a failure (or even an error, for smarter oracles), and an *approve* verdict otherwise. The oracle is not always able to reach a decision: in these cases the test output is classified as *inconclusive*.

In a scenario in which a limited number of test cases is executed, sometimes even derived manually, the oracle can be the tester himself/herself, who can either inspect the test log a posteriori, or even decide a priori, during test planning, the conditions that make a test successful and code these conditions into the employed test driver.

When the tests cases are automatically derived, or also when their number is quite high, in the order of thousands, or millions, a manual log inspection or codification is not thinkable. Automated oracles must then be implemented.

Generally speaking, an oracle could be derived from a specification of the expected behavior. Thus, in principle, the automated derivation of test cases from specifications, as is done in model-based testing (see Section 5.2), has the advantage that we get an abstract oracle specifica-

tion as well. However, the gap between the abstract level of specifications and the concrete level of executed tests only allows for partial oracle implementations, i.e., only necessary (but not sufficient) conditions for correctness can be derived. On the other hand, if we had available a mechanism that knows the correct results in advance and infallibly, it would not be necessary to develop the system under test: we could use the oracle instead--hence the need for approximate solutions.

Different approaches for approximate oracles include ([5], [58]): assertions could be embedded into the program so as to provide run-time checking capability; conditions expressly specified to be used as test oracles could be developed, in contrast with using the same specifications (i.e., written to model the system behavior and not for run-time checking); the produced execution traces could be logged and analyzed.

In some cases, the oracle can be an earlier version of the system that we are going to replace with the one under test. A particular instance of this situation is regression testing (see Section 4.4), in which the test outcome is compared with earlier version executions (which however, in turn had to be judged as passed or failed).

In view of these considerations, it should be evident that the oracle might not always judge correctly. So the notion of *coverage*[10] of an oracle is introduced to measure its accuracy. Oracle coverage could be measured, for instance, by the probability that the oracle rejects a test (on an input chosen at random from a given probability distribution of inputs), given that it should reject it [13]. Thus, a perfect oracle exhibits 100 percent coverage, while a less than perfect oracle may yield different measures of accuracy.

2.7 Testability

The term "software testability" has two related but different meanings: on the one hand, it refers to the degree to which it is easy for software to fulfil a given test coverage criterion. as in [3]; on the other hand, it is defined as the likelihood, possibly measured statistically, that the software will expose a failure under testing, *if* it is faulty, as in [13].

Both meanings are important and both ultimately refer to the effort required for successfully completing the testing of a piece of software. The testability of a software component depends on its properties of *observability* and *controllability* [48]. Informally, a software component is observable if distinct outputs are observed for distinct inputs, and is controllable if any desired output can be produced from a specified input.

2.8 Limitations of Testing

Unfortunately, there are few mathematical certainties on which software testing foundations can be laid. The firmest one, as everybody now recognizes, is that, even after successful completion of an extensive testing campaign, the software can still contain faults. The obvious reason is that complete testing is not feasible in real software. Because of this, testing must be driven based on risk and can be seen as a risk management strategy.

As first stated by Dijkstra as early as thirty years ago [22], *testing can never prove the absence of defects*; it can only possibly reveal the presence of faults by provoking malfunctions. In

10. The usage with a quite different meaning of the same term "coverage" adopted for test criteria is just a coincidence.

the decades since, a lot of progress has been made both in our knowledge of how to scrutinize a program's executions in rigorous and systematic ways, and in the development of tools and processes that can support the tester's tasks.

Yet, the more the discipline progresses, the clearer it becomes that it is only by means of *rigorous empirical studies* that software testing can increase its maturity level [36]. Testing is in fact an engineering discipline, and as such it calls for evidence and proven facts, to be collected either from experience or from controlled experiments, based on which testers can make predictions and decisions.

3. Other V&V Approaches

Testing is one among many approaches to software verification and validation (V&V). Verification refers to the *process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase,* whereas validation is the *process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements* [28].

Before describing test techniques, we provide a brief overview of other V&V approaches.

3.1 Static Techniques

A first distinction can be made between dynamic and static techniques, depending on whether the software is executed or not. Static techniques are based solely on the (manual or automated) examination of project documentation, of software models and code, and of other related information about requirements and design. Thus, static techniques can be employed throughout development, and their early usage is of course highly desirable. Considering a generic development process, they can be applied ([52], [2]):

- At the *requirements stage* for checking language *syntax*, consistency, and completeness as well as the adherence to established conventions.
- At the *design phase* for evaluating the implementation of requirements, and detecting inconsistencies (for instance between the inputs and outputs used by high level modules and those adopted by sub-modules).
- During the *implementation phase* for checking that the form adopted for the implemented products (e.g., code and related documentation) adheres to the established standards or coding conventions, and that interfaces and data types are correct.

Traditional static techniques include [53]:

- ***Software inspection*** — The step-by-step analysis of the deliverables (documents, code and so on) produced, against a compiled checklist of common and historical defects.
- ***Software reviews*** — The process by which different aspects of the work product are presented to project personnel (managers, users, customers, etc.) and other interested stakeholders for comment or approval. The process may focus in particular on the evaluation of the compliance to standards, procedures, and guidelines. Different types of review include: code review, design review, formal qualification review, requirements review, and test readiness review.

- ***Code reading*** — The desktop analysis of the produced code for discovering typing errors that do not violate style or syntax. This process includes the checking of typographical errors, data structures, control flow, and processing.
- ***Algorithm analysis and tracing*** — The process in which the complexity of algorithms employed and the worst-case, average-case and probabilistic analysis evaluations can be derived.

The processes implied by the above techniques are heavily manual, error-prone, and time consuming. To overcome these problems, researchers have proposed static analysis techniques relying on the use of formal methods [65]. The goal is to automate as much as possible the verification of the properties of the requirements and the design. Towards this goal, it is necessary to enforce a rigorous and unambiguous formal language for specifying the requirements and the software architecture. In fact, if the language used for specification has a well-defined semantics, algorithms and tools can be developed to analyze the statements written in that language.

The basic idea of using a formal language for modeling requirements or design is recognized as a foundation for software verification. *Formal verification* techniques today attract quite a lot of attention from both research institutions and industry, and it is foreseeable that proofs of correctness will be increasingly applied, especially for the verification of critical systems.

One of the most promising approaches for formal verification is *model checking* [21]. Essentially, a model checking tool takes as input a *model* (a description of system functional requirements or design) and a *property* that the system is expected to satisfy. The model checker performs an automated analysis, and then either proves that the given model satisfies the stated property, or generates a *counterexample*. The latter details why the model doesn't satisfy the specification. By studying the counterexample, the source of the error in the model can also be identified. Model checking has proven to be a successful technology defined to check properties over a variety of real-time embedded and safety-critical systems, and with the computing power of modern machines, its wide-scale application is becoming a concrete prospect [34].

Between static and dynamic analysis techniques is *symbolic execution* [16], which executes a program by replacing variables with symbolic values. Thus it produces a set of expressions relative to the various output variables. Research on tools for performing symbolic execution was rather active in the late 70's, at which time the goal was to derive test data for coverage testing in a completely automated way, but the idea was then abandoned because of its limited applicability to real complex programs. Problems were classically the handling of arrays, pointers, and procedure calls. Recently the automated generation of test data for coverage testing is again attracting a lot of interest, and advanced tools are being developed based on a similar approach to symbolic execution exploiting *constraint solving* techniques [6]. A flowgraph path to be covered is translated into a path constraint, whose solution provides the desired input data. In *concolic testing* [56], symbolic execution is applied in parallel with concrete test executions, which help to reduce the space of solutions.

We conclude this section considering the alternative application of static techniques in producing values of interest for controlling and managing the testing process. Different estimates can be obtained by observing specific properties of present or past products, and/or parameters of the development process. For instance, during the testing phase, static techniques may be applied to estimate the total number of defects and provide other useful measures. Static defect models

can be applied, for instance, to identify higher-risk modules and consequently to re-allocate testing resources or to modify design. Thus, static techniques could also be very attractive to managers not only for checking, but also for prediction purposes, because they provide "numbers," which the managers are eager for, very early in the process compared to dynamic techniques. The latter can only be used late in the lifecycle, when it may be too late to efficaciously re-direct development efforts.

3.2 Dynamic Techniques

Dynamic techniques [4] obtain information of interest about a program by observing some executions. Standard dynamic analysis includes testing (on which we focus in the rest of the chapter) and *profiling*. Essentially, a program profile records the number of times some entities of interest occur during a set of controlled executions. Profiling tools are increasingly used today to derive measures of coverage, for instance in order to dynamically identify control flow invariants, as well as measures of frequency, called *spectra*, which are diagrams providing the relative execution frequencies of the monitored entities. In particular, *path spectra* refer to the distribution of (loop-free) paths traversed during program profiling. Specific dynamic techniques also include simulation, sizing, and timing analysis, and prototyping [52].

Testing is based on the execution of the code on inputs. Of course, although the set of input values can be considered infinite, those that can be run effectively during testing are finite. It is in practice impossible, due to the limitations of the available budget and time, to exhaustively exercise every input of a specific set even when not infinite. In other words, by testing we observe some samples of the program's behavior.

A test strategy, therefore, must be adopted to find a trade-off between the number of chosen inputs and overall time and effort dedicated to testing purposes. Different techniques can be applied depending on the target and the effect that should be reached. We will describe test selection techniques in Section 5.

In the case of concurrent, non-deterministic systems, the results obtained by testing depend not only on the input provided but also on the state of the system. Therefore, when speaking about test input values, it is implied that the definition of the parameters and environmental conditions that characterize a system state must be included when necessary.

Once the tests are selected and run, another crucial aspect of this phase is the already introduced oracle problem, which means deciding whether or not the observed outcomes are acceptable (see Section 2.6).

4. Test Levels

During the development lifecycle of a software product, testing is performed at different levels and can involve the whole system or parts of it. Depending on the process model adopted, then, software testing activities can be articulated in different phases, each one addressing specific needs relative to different portions of a system. Whatever the process adopted, we can at least distinguish in principle between *unit, integration,* and *system test* [54]. These are the three testing stages of a traditional phased process (such as the classical waterfall). However, even considering different, more modern process models, a distinction between these three test levels remains useful to emphasize three logically different stages in the verification of a complex software system.

None of these levels is more relevant than another, and, more importantly, one stage cannot stand in for another, because each addresses different typologies of failure.

4.1 Unit Test

A unit is the smallest testable piece of software, which may consist of hundreds or even just a few lines of source code, and generally represents the result of the work of one programmer. The unit test's purpose is to ensure that the unit satisfies its functional specification and/or that its implemented structure matches the intended design structure [54].

Unit tests can also be applied to check interfaces (parameters passed in correct order, number of parameters equal to number of arguments, parameter and argument matching), local data structure (improper typing, incorrect variable name, inconsistent data type) or boundary conditions. A good reference for unit test is [30].

4.2 Integration Test

Generally speaking, integration is the process by which software pieces or components are aggregated to create a larger component. Integration testing is specifically aimed at exposing the problems that can arise at this stage. Even though the single units are individually acceptable when tested in isolation, in fact they could still result in incorrect or inconsistent behavior when combined in order to build complex systems. For example, there could be an improper call or return sequence between two or more components [48]. Integration testing is thus aimed at verifying that components interact according to the specifications as defined during preliminary design. In particular, it mainly focuses on the communication interfaces among integrated components.

There are not many formalized approaches to integration testing in the literature, and practical methodologies rely essentially on good design sense and the testers' intuition. Integration testing of traditional systems was done substantially in either a non-incremental or an incremental approach. In a non-incremental approach, the components are linked together and tested all at once: this is the so-called, and not advisable, "big-bang" testing [35]. In the preferable incremental approaches, we find the classical "top-down" strategy, in which the modules are integrated one at a time, from the main program down to the subordinated ones, or "bottom-up," in which the tests are constructed starting from the modules at the lowest hierarchical level and then are progressively linked together upwards, to construct the whole system. Usually in practice, a mixed approach is applied, as determined by external project factors (e.g., availability of modules, release policy, availability of testers, and so on) [54].

In object-oriented, distributed systems, approaches such as top-down or bottom-up integration and their practical derivatives are no longer usable, as no "classical" hierarchy between components can be generally identified. Some other criteria for integration testing imply integrating the software components based on identified functional threads [35]. In this case, the test is focused on those classes used in response to a particular input or system event (thread-based testing); or by testing together those classes that contribute to a particular use of the system.

Finally, some authors have used the dependency structure between classes as a reference structure for guiding integration testing, i.e., their static dependencies [42], or even the dynamic relations of inheritance and polymorphism [41]. Such proposals are interesting when the number of classes is not too big; however, test planning in those approaches can begin only at a mature stage of design, when the classes and their relationships are already stable.

A different branch of the literature is testing based on the *Software Architecture*: this specifies the high level, formal specification of a system structure in components and their connectors, as well as the system dynamics. The way in which the description of the Software Architecture could be used to drive the integration test plan is currently under investigation, e.g., [46].

4.3 System Test

System test involves the whole system embedded in its actual hardware environment and is mainly aimed at verifying that the system behaves according to user requirements. In particular, it attempts to reveal bugs that cannot be attributed to components as such, to the inconsistencies between components, or to the planned interactions of components and other objects (which are the subject of integration testing). Summarizing, the primary goals of system testing can be [48]:

- Discovering the failures that manifest themselves only at the system level and hence were not detected during unit or integration testing;
- Increasing the confidence that the developed product correctly implements the required capabilities;
- Collecting information useful for making decisions about the release of the product.

System testing should therefore ensure that each system function works as expected, that any failures are exposed and analyzed, and additionally, that interfaces for input and output routines behave as required.

System testing makes available information about the actual status of development that other verification techniques such as review or inspections of models and code cannot provide.

Generally, system testing includes testing for performance, security, reliability, stress testing, and recovery [35, 54]. In particular, tests and data collected during system testing can be used for defining an operational profile necessary to support a statistical analysis of system reliability [44].

A further test level, called *Acceptance Test*, is often added to the above subdivision. This is, however, more an extension of system test, rather than a new level. It is in fact a test session conducted over the whole system, which mainly focuses on the customer requirements more than on the compliance of the implementation against some specification. It may also include Usability Testing with the intent of verifying that the effort required from end-users to learn to use and fully exploit the system functionalities is acceptable. (See also Section 5.3.)

4.4 Regression Test

Properly speaking, *regression test* is not a separate level of testing (we listed it in fact among test objectives in Section 4.5 below), but may refer to the retesting of a unit, a combination of components or a whole system (see Fig. 1) after modification, in order to ascertain that the change has not introduced new faults [54].

As software produced today is constantly undergoing evolution, driven by market forces and technology advances, regression testing is the predominant portion of testing effort in industry.

Since both corrective and perfective modifications may be performed quite often, to re-run all previously executed test cases after each change would be prohibitively expensive. Therefore, various types of techniques have been developed to reduce the costs and increase the effectiveness of regression testing.

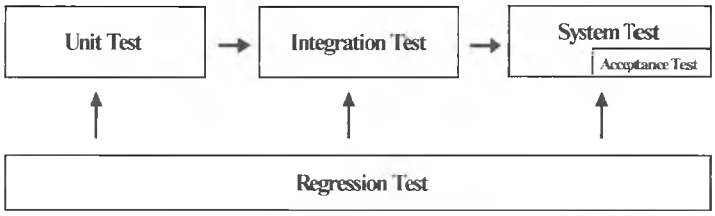


Fig. 1: Logical schema of software testing levels.

We already spoke of test suite minimization and prioritization (Sec. 2.5): both can be applied in regression testing. *Selective regression test* techniques [67] help in selecting a (minimized) subset of the existing test cases by examining the modifications (for instance at code level, using control flow and data flow analysis). Other approaches instead *prioritize* the test cases according to some specified criterion (for instance, maximizing the fault detection power or the structural coverage); so that the test cases judged the most effective with regard to the adopted criterion can be taken first, up to the available budget.

4.5 Objectives of Testing

Software testing can be applied for different purposes, such as verifying that the functional specifications are implemented correctly, or that the system shows specific non-functional properties such as performance, reliability, and usability. A (certainly not complete) list of relevant testing objectives includes [48]:

- ***Acceptance/Qualification testing***: — The final test action prior to deploying a software product. Its main goal is to verify that the software meets the customer's requirements. Generally, it is run by or with the end-users to perform those functions and tasks for which the software was built [54].
- ***Installation testing*** — The system is verified upon installation in the target environment. Installation testing can be viewed as system testing conducted once again according to hardware configuration requirements. Installation procedures may also be verified [54].
- ***Alpha testing*** — Before releasing the system, it is deployed to some in-house users for exploration of the functions and business tasks. Generally, there is no test plan to follow, but the individual tester determines what to do [38].
- ***Beta testing*** — The same as alpha testing but the system is deployed to external users. In this case, the amount of detail, the data, and approach taken are entirely up to the external *testers, who are responsible for creating their own environment, selecting their data, and* determining what functions, features, or tasks to explore. Each tester is also responsible for identifying their own criteria for whether to accept the system in its current state or not [38].
- ***Regression testing*** — According to [28], regression testing is the "selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements." In practice, *the objective is to show that a system which previously passed the tests still does* [67]. Notice that a trade-off must be made between the assurance given by regression testing every time a change is made and the resources required to do that.

- ***Usability testing***: — This important testing activity evaluates the ease of using and learning the system and the user documentation, as well as the effectiveness of system functioning in supporting user tasks, and, finally, the ability to recover from user errors [49].
- ***Conformance testing/Functional testing*** — The test cases are aimed at validating that the observed behavior conforms to the specifications. In particular, it checks whether the implemented functions are as intended and provide the required services and methods. This test can be implemented and executed against different test targets, including units, integrated units, and systems [53].
- ***Performance testing*** — This is specifically aimed at verifying that the system meets the specified performance requirements, for instance, capacity and response time [54].
- ***Reliability* achievement** — Indeed, whether few or many, some faults will inevitably escape testing and debugging. However, a fault can be more or less disturbing depending on whether, and how frequently, it will eventually show up to the final user (and depending of course on the seriousness of its consequences). So, in the end, one measure that is important in deciding whether a software product is ready for release is its reliability. Strictly speaking, software reliability is a probabilistic estimate, and measures the probability that the software will execute without failure in a given environment for a given period of time [44].

  Thus, the value of software reliability depends on how frequently those inputs that cause a failure will be exercised by the final users. Estimates of software reliability can be produced via testing. To this purpose, since the notion of reliability is specific to "a given environment," the tests must be drawn from an input distribution that approximates as closely as possible the future usage in operation, which is called the operational distribution. Testing can also be used as a means to improve reliability; in such a case, the test cases must be randomly generated according to the operational profile, i.e., they should sample more densely the most frequently used functionalities [44].

5. Test Techniques

As hinted in Section 2.5, the technique used for test case selection will greatly affect test effectiveness. There exists a full range of test techniques, quite different from one another, which embrace a variety of aims.

5.1 Selection Criteria Based on Code

Code-based testing, also called structural or white-box testing, was the dominant trend in software testing research during the late 70's and the 80's. One reason is certainly that in those years in which formal approaches to specification were much less mature and pursued than now, the only *RM* formalized enough to allow for the automation of test selection or for a quantitative measurement of thoroughness was the code itself.

Referring to the fault-error-failure chain described in Section 2.3, the motivation for code-based testing is that potential failures can only be detected if the parts of code related to the causing faults are executed. Hence, by monitoring code coverage, one tries to exercise thoroughly all "program elements:" depending on how the program elements to be covered are identified, several test criteria are defined.

In *structural testing*, the program is modeled as a graph, whose entry-exit paths represent the flow of control; hence it is called a *flowgraph*. Finding a set of flowgraph paths fulfilling a coverage criterion thus becomes a matter of properly visiting the graph (see for instance [12]). Code coverage criteria are also referred to as path-based test criteria, because they map each test input to a unique path *p* on the flowgraph.

The ideal and yet unreachable target of *code-based testing* would be the exhaustive coverage of all possible paths along the program control-flow. The underlying test hypothesis here is that by executing a path once, potential faults related to it will be revealed, i.e., it is assumed that every input executing a particular path will either fail or succeed (which is not necessarily true, of course).

Full path coverage is not applicable because every program with unbounded loops would yield an infinite number of paths. Even limiting the number of iterations within program loops, which is the usually practiced tactic in testing, the number of tests would remain unfeasibly high. Therefore, all the proposed code-based criteria attempt to realize cost-effective approximations to path coverage, by identifying specific (control-flow or data-flow) elements of a program that are deemed to be relevant for revealing possible failures, and by requiring that enough test cases to cover all such elements be executed.

The landmark paper in code-based testing is [55], in which a family of criteria was introduced, based on both control-flow and data-flow. A *subsumption* hierarchy between the criteria was derived, based on the inclusion relation such that a test suite satisfying a subsuming criterion is guaranteed to also satisfy the (transitively) subsumed criterion.

Statement coverage is the most elementary criterion, requiring that each statement in a program be exercised at least once. The *branch coverage* criterion instead requires that each branch in a program be exercised at least once. Note that complete statement coverage does not ensure that all branches are exercised (simply because empty branches would be left out).

Branch coverage is also called "decision coverage," because it considers the outcome of a decision predicate. When a predicate is composed by the logical combination of several conditions, a variation to branch coverage is given by "condition coverage," which requires instead testing the true and false outcome of the individual conditions of predicates. Further criteria consider together coverage of decisions and conditions under differing assumptions (see, e.g., [25]).

In *data flow-based t*esting, the flowgraph is annotated at each node with information about how the program variables are defined and used (a separate annotated flowgraph is derived for each variable), and the test cases are aimed at exercising how the values assigned to variables are used along different paths.

For example, *all-uses coverage* requires that for every variable, every possible use of a definition is covered by at least one test case. Note that if a variable V is assigned a value at node X of the flowgraph, a reference to the same variable at some other node Y is a proper "use" only if there exists at least one path from X to Y that contains no other definition of V. The triple (V, X, Y) then is called a *definition-use association.* Another more stringent data-flow criterion requires

covering *all-DU-paths*, i.e., for every variable, all loop-free or simple cycle paths from every definition to every use of that definition must be tested.

The application of code-based criteria poses some tough problems, which make complete coverage quite difficult, if not impossible, to reach. One reason can be the existence of unreachable code (e.g., procedures which are never invoked), due for instance to reuse of legacy systems. Another frequent problem hampering full coverage is *infeasible paths*, i.e., flowgraph paths that cannot be traversed because of contradicting predicate conditions. Intuitively, the more complex the requirements we impose on the elements to be covered, the higher the incidence of infeasibility. Last, even for feasible paths, finding an input that executes a selected flowgraph path is not only an undecidable problem in principle [61], but also a quite difficult one to solve in practice. As previously stated, traditionally, symbolic execution was attempted, with scarce practical success.

A promising approach to automating test data generation is Search-Based Testing [45], applying search-based metaheuristic techniques. The selected test criterion is encoded as a fitness function, which is used to guide the exploration of the space of solutions (the sought test cases) towards the potentially most promising areas of the input space. The approach is attractive in that it can be applied not only to automate structural testing, which has received the largest attention so far, but also to other test criteria, by appropriately defining the fitness function. Other test automation approaches include dynamic generation based on optimization [40], genetic algorithms [51], or the already mentioned constraint-solving techniques [6].

It must be kept in mind, however, that code-based test selection is a tautology: it looks for potential problems in a program by using the program itself as a reference model. In this way, for instance, faults of missing functionalities could never be found.

As a consequence, code-based techniques should be more properly used as adequacy criteria. In other words, testers should consider low measures of coverage as a warning. If uncovered elements remain, this implies that the set of test cases is not executing some parts of the functionalities or of the design.

A sensible approach is to use another artifact as the reference model from which the test cases are designed and to monitor a measure of coverage while tests are executed, so as to evaluate the thoroughness of the test suite. If some elements of the code remain uncovered, additional tests to exercise them should be found, as it can be an indication that the tests do not address some function that is coded.

A final warning is worth mentioning: "exercised" and "tested" are not synonymous. An element is really tested only when its execution produces an effect on the output. In view of this statement, under most existing code-based criteria even 100% coverage could leave some statements untested.

5.2 Selection Criteria Based on Specifications

In specification-based testing, the reference model *RM* is derived in general from the documentation relative to program specifications. Depending on how the latter are expressed, largely different techniques are possible [35]. Early approaches [47] looked at the Input/Output relation of the program seen as a "black-box" and manually derived:

- ***Equivalence classes*** — By partitioning the input domain into subdomains of "equivalent" inputs, in the sense explained in Section 2.4, that any input within a subdomain can be taken as a representative for the whole subset. Hence, each input condition must be separately considered to first identify the equivalence classes. The second step consists of choosing the test inputs representative of each subdomain; it is good practice to take both valid and invalid equivalence classes for each condition. The Category Partition method described below in this section belongs to this approach.
- ***Boundary conditions*** — For example, those combinations of values that are "close" (precisely on, above and beneath) the borders of the equivalence classes identified both in the input and the output domains. This test approach is based on the intuitive fact, also proved by experience that faults are more likely to be found at the boundaries of the input and output subdomains.
- ***Cause-effect graphs*** — These are combinatorial logic networks that can be used to explore in a systematic way the possible combinations of input conditions. By analyzing the specification, the relevant input conditions, or causes, and the consequent transformations and output conditions, the effects are identified and modeled into graphs linking the effects to their causes. A detailed description of this early technique can be found in [47].

A simple, intuitive, yet effective approach is the Category-Partition (CP) method [50] for the automated generation of functional tests from annotated semi-formal specifications. CP consists of a stepwise methodology to derive a suite of functional tests from the specifications written in structured, semiformal language. The first step of the CP method is to analyze the functional requirements to divide the analyzed system into functional units to be separately tested. A functional unit can be a high-level function or a procedure of the implemented system. For each defined functional unit, the *environment conditions* (system characteristic of a certain functional unit) and the *parameters* (explicit input of the same unit) relevant for testing must be identified.

The test cases are then derived by finding significant values of environment conditions and parameters. This can be done by dividing them into *categories* representing relevant system properties or particular characteristics of parameters or environment conditions. Then, for each category, different *choices* are identified that are significant values for these categories. To prevent the construction of redundant, not meaningful, or even contradictory combinations of choices, the choices can be annotated with constraints, which can be of two types: either (i) properties or (ii) special conditions.

In the first case, some properties are set for certain choices, and selector expressions related to them (in the form of simple *if* conditions) are associated with other choices: a choice marked with an *if* selector can then be combined only with those choices from other categories that fulfill the related property. The second type of constraint is useful to reduce the number of test cases: some markings, namely "error" and "single," are coupled to some choices, referring to erroneous or special conditions, respectively, that we intend to test, but that do not have to be combined with all possible choices. The list of all the choices identified for each category, with the possible addition of the constraints, is called the Test Specification. It is not yet a list of test cases, but contains all the necessary information for instantiating them by unfolding the constraints. A suite of test cases is finally obtained by taking all the possible combinations of choices for all the categories.

The CP method has encountered wide interest, and has inspired the development of a large number of test methodologies. Its basic principle has been applied to specifications in several languages, also using formal languages such as Z and UML [7].

Approaches such as the ones described above all require a degree of "creativity" [47]. To make testing more repeatable and to push automation, many approaches have been proposed that start from a formal specification. Early attempts included algebraic specifications, VDM, and Z [59], while a more recent collection of approaches to formal testing can be found in [26].

A prevalent approach nowadays is *model-based testing* (MBT), which relies on an abstract (formal) representation either of the system under test or of its requirements. This can be analysed for derivation of the test cases and the expected outcomes as well. Usually the abstract representation focuses on the functionality of the system under test so that the typical use of the derived test case is for checking if the software system complies with the functionality as described in the model.

The main advantage of MBT is that the model of the specification can be used to automatically derive the test cases for the system under test. Usually the test strategies adopted are based on maximising some notion of coverage, by trying to exercise the most useful sequences of inputs and thus deciding which test cases to include into a finite test suite.

The key components of MBT thus are: the notation used for representing the model of the system; the test strategy or algorithm for test case generation; and the supporting infrastructure for the test execution including the evaluation of the expected outputs. Generally, due to the complexity of adopted techniques, the MBT approaches are used in conjunction with test automation harnesses (see Section 10).

Concerning the model notation, the widespread ones are *Finite State Machine* (FSM) ([26], [59]) and *Labelled Transition Systems* (LTS) [15]. Both FSMs and LTSs represent a system as a set of states and transitions between them. For FSMs, each transition represents an input from the user and the corresponding response to be provided by the system. For LTSs, each transition is labelled by precisely one action, which can be either an input from the user or a response of the system. Usually FSMs are used for deterministic systems in which a synchronous communication of input and output actions is adopted, while LTSs are more suitable to support parallel computation. Many times the adoption of a LTS specification model is related to a "conformance relation."

Given the LTS for the specification S and one of its possible implementations I (the program to be tested), various test generation algorithms have been proposed to produce sound test suites, i.e., such that programs passing the test correspond to conformant implementations according to the selected relation. One of the widespread relations is the *ioco* conformance, which verifies whether an implemented system behaves as if it were an input-output transition system that is always able to perform any input action possible in the specification [59].

Many MBT approaches nowadays focus on testing from UML models. In this context, State Diagrams, Message Sequence Charts and Specification and Description Language (SDL) are alternatively used. A spectrum of approaches has been and is being developed, ranging from strictly formal testing approaches based on UML statecharts [43], to approaches trying to overcome UML limitations requiring OCL (Object Constraint Language) [60], to pragmatic approaches using the design documentation as-is and proposing automated support tools [7].

Frequently, extra information (such as the actions attached to the transitions or data processing behavior) could be added to the models [24].

A related problem to model-based testing is to check the correctness of the model itself. Depending on the notation used, the abstract representation of the system can be used for simulating the system behaviour or for formally checking its correctness. In particular, model checking might provide either a formal verification or counterexamples to violated properties. Normally, these counterexamples are useful to guide an analyst when searching for the root cause of a property violation. Even in those cases in which a correct model could be assumed, this does not completely solve the problem, as eventually the model and/or the derived test cases will change during the product development.

5.3 Other Criteria

Specification-based and *code-based test techniques* are often contrasted as functional vs. structural testing. These two approaches to test selection are not to be seen as alternative, but rather as complementary; in fact, they use different sources of information, and have proved to highlight different kinds of problems. They should be used in combination, depending on budgetary considerations [35]. Moreover, beyond code or specifications, the derivation of test cases can be done starting from other informative sources. Some other important strategies for test selection are briefly overviewed below.

- ***Based on tester's intuition and experience*** — The most practiced test selection criterion in industry probably is still ad-hoc testing [38], in which tests are derived relying on the tester's skill, intuition, and experience with similar programs. Indeed, expert testers may perform as very good selection "mechanisms" (with the necessary warnings against exclusively relying on such a subjective strategy). In particular, ad hoc testing might be useful for identifying special tests, those not easily captured by formalized techniques. Empirical investigations [8] showed in fact that tester's skill is the factor that mostly affects test effectiveness in finding failures.

- ***Exploratory testing*** [39] — A related technique is defined as "simultaneous learning, test design, and test execution"; that is, the tests are not defined in advance in an established test plan, but are dynamically designed, executed, and modified. The effectiveness of exploratory testing relies on the tester's knowledge, which can be derived from various sources: observed product behavior during testing, familiarity with the application, the platform, the failure process, the type of possible bugs, the risk associated with a particular product, and so on.

- ***Random*** — A basic criterion is random testing, according to which the test inputs are picked randomly from the whole input domain according to a specified distribution, i.e., after assigning different "weights" (more properly, probabilities) to the inputs. Under the uniform distribution, no distinction among the inputs is made, and any input has the same probability of being chosen. A growing body of research is, however, investigating ways to make random testing more efficient by exploiting the available information on "failure patterns," i.e., contiguous areas of the input domain where it is observed that failures tend to concentrate.

  Various approaches are proposed within the family of Adaptive Random Testing (ART) [18], which combines random test selection with a filtering process to favor an

even spread of test cases throughout the input domain: if some executed test cases have not revealed any failures, the approach assumes that the parts of the system exercised by these cases seem to be error-free or nearly so, and thus that new test cases located away from those are more likely to reveal failures. The latter approach differs from random testing under an operational distribution (described below).

- ***Based on operational usage*** — The test environment must reproduce the operational environment of the software as closely as possible (operational profile) [35], [44], [54]. The idea is to infer, from the observed test results, the future reliability of the software when in actual use. To do this, inputs are assigned a probability distribution, or profile, according to their occurrence in actual operation. In particular, Software Reliability Engineered Testing (SRET) [44] is a testing methodology encompassing the whole development process, whereby testing is "designed and guided by reliability objectives and expected relative usage and criticality of different functions in the field."

- ***Fault-based*** — With different degrees of formalization, fault-based testing techniques devise test cases specifically aimed at revealing categories of likely or pre-defined faults. In particular, it is possible that the RM is given by expected or hypothesized faults, such as in error guessing, or mutation testing. Specifically in error guessing [38] test cases are designed by testers trying to figure out the most plausible faults in a given program. A good source of information is the history of faults discovered in earlier projects, as well as the tester's expertise. In mutation testing [53], a mutant is a slightly modified version of the program under test, differing from it by a small, syntactic change. Every test case exercises both the original and all generated mutants.

  If a test case is successful in identifying the difference between the program and a mutant, the latter is said to be killed. The underlying assumption of mutation testing, the coupling effect is that, by looking for simple syntactic faults, more complex but real faults will be found. For the technique to be effective, a high number of mutants must be automatically derived in a systematic way.

- ***User observation*** — As above listed, one of the objectives for testing is usability, i.e., to test how the final users will approach the system and to detect potential problems early at the user interface. Usability principles can be used as guidelines to check and discover potential problems in the user interface design [49]. The evaluation of specific heuristics involves systematic observation under controlled conditions to determine how well people can use the systems and the interfaces.

6. Aspects of Usefulness and Usability

Ultimately, we are interested in the broader question of whether a system is good enough for its intended purposes, i.e., system *acceptability* [49], of which usability is just one aspect. Nielsen [49] proposes a model of system acceptability as a combination of social acceptability (i.e., whether populations will consider it desirable or offensive) and practical acceptability concerns. The latter includes among its various attributes, such as cost, reliability, compatibility, etc., the category of usefulness, which is further classified into utility and usability. Utility covers the functional aspects of a system, whether it does what is needed, whereas usability concerns all those aspects involving how the system interfaces with users.

Usability evaluation can be performed at different stages during the development process, and with varying degrees of users' involvement. Typically, prototypes provide means for early evaluation so that it is still feasible to influence the project. Two broad classes of usability evaluation methods can be distinguished as usability testing and usability inspection methods [27], depending on whether the system end users are involved or not. Usability inspection employs devoted evaluators, e.g., usability specialists or domain experts:

- ***Heuristic evaluation*** — The most common usability inspection method is heuristic evaluation, in which the user interface characteristics are compared against usability principles and established guidelines. Generally, more evaluations are carried out, as evidence has demonstrated that different evaluators tend to find different problems [49], so it is a good practice, compatible with the available budget, to aggregate more reports (reasonably five, and at least three). It is, however, important to ensure that the evaluations are independent; hence the evaluators should carry out their own assessment in isolation.

  The evaluation could be supervised by an observer or not, with pros and cons for either case. An observer would directly get the results from the evaluator assessment, thus alleviating the burden of this latter individual to write a detailed report and making communication more effective. On the other hand, with the presence of the observer, evaluators would not deliver any more formal records of the sessions and the cost of the inspection would increase.

- ***Cognitive walkthroughs*** — Another popular usability inspection method is the cognitive walkthrough [64], which is task-driven, i.e., the evaluator examines the sequences of user-required actions to accomplish a specified task. This is quite an intuitive approach to evaluating the learnability of a user's interface. The evaluator is required to answer, for each task, some precise questions about the user's expected action. Since its original proposal by Wharton et al. [63] in the early 90's, several variants of the approach have been proposed, mainly aimed at making it more effective and less formal.

- ***User-testing approaches (observation sessions)*** — Contrary to inspections carried out by evaluators, usability testing involves letting some users familiarize themselves with and use the system under test. It is clearly highly important that the session reproduce as closely as possible real usage; hence the testing session should be as representative as possible of field usage; besides, the testing should cover all system interface features. Such concerns are part of the planning stage.

  As we have said for functional testing, usability testing may also address different goals. Nielsen [49] distinguishes between "formative evaluation." aiming at highlighting potential usability issues and improving the interface, and "summative evaluation" aiming at assessing and comparing different interfaces. There are different techniques that can be applied.

  The most common one for formative evaluation is thinking-aloud. The technique simply consists of requiring the user to continuously think aloud while using the system. The strength of the approach is that the testing not only evidences potential problems, but also hints at their causes. However, verbalizing their thoughts aloud may seem unnatural to many people; to prevent such an issue, an experimenter could assist by prompting the users with enquiries about what they are thinking. but he/she should not interfere with the

test session by providing any additional information. An opposite approach is taken in coaching, where an expert guides the users to speedily learn how to use the system.

An alternative to thinking-aloud is *constructive interaction*, in which two users test the system interface together, so that they will be more naturally inclined to verbalise their thinking to communicate with each other.

Other approaches for usability testing include field observation, which just consists of observing the users working in their natural environment. The observation may be conducted through unobtrusive visits or through camera recording. Even less obtrusive is data logging, i.e., records of session logs are analysed, usually to get interesting performance measures of user's sessions.

Finally, *user's questionnaires* or *interviews* can also provide useful, although less direct, information.

7. Test Measurements

Measurements are nowadays applied in every scientific field for quantitatively evaluating parameters of interest, understanding the effectiveness of techniques or tools, the productivity of development activities (such as testing or configuration management), the quality of products, and more. In particular, in the software engineering context they are used for generating quantitative descriptions of key processes and products, and consequently controlling software behavior and results. But these are not the only reasons for using measurement; it can permit definition of a baseline for understanding the nature and impact of proposed changes. Moreover, measurement allows managers and developers to monitor the effects of activities and changes on all aspects of development. In this way, actions to check whether the final outcome differs significantly from plans can be taken as early as possible [37].

We have already hinted at useful test measures throughout the chapter. It can be useful to briefly summarize them altogether. Considering the testing phase, measurement can be applied to evaluate the program under test, or the selected test set, or even for monitoring the testing process itself [10].

7.1 Evaluation of the Program under Test

For evaluating the program under test, the following measurements can be applied:

- ***Program measurement can aid in test planning and design*** — Considering the program under test, three different categories of measurement can be applied as reported in [37]:
  - ***Linguistic measures*** — These are based on proprieties of the program or of the specification text. This category includes, for instance, the measurement of Source Lines of Code (LOC), statements, the number of unique operands or operators, and function points.
  - ***Structural measures*** — These are based on structural relations between objects in the program and comprise control flow or data flow complexity. These can include measurements relative to the structuring of program modules, e.g., in terms of the frequency with which modules call each other.
  - ***Hybrid measures*** — These may result from the combination of structural and linguistic properties.

- ***Fault density*** — This is a widely used measure in industrial contexts and foresees the counting of the discovered faults and their classification by their type. For each fault class, fault density is measured by the ratio between the number of faults found and the size of the program [53].
- ***Life testing, reliability evaluation*** — By applying the operational testing for a specific product, it is possible either to evaluate its reliability and decide if testing can be stopped, or to achieve an established level of reliability. In particular, *Reliability Growth* models can be used for predicting the product reliability [44].

7.2 Evaluation of the Tests Performed

For evaluating the set of test cases implemented, the following measures can be applied:

- ***Coverage/thoroughness measures*** — Some adequacy criteria require exercising a set of elements identified in the program or in the specification by testing. In this case, during testing the number of elements covered by test cases is monitored and the coverage (expressed as a percentage) is derived as the ratio between the covered elements and the total number. The coverage can be, for instance, relative to the paths, the statements or the branches, as well as the number of functionalities exercised during testing [54].
- ***Effectiveness*** — In general, a notion of effectiveness must be associated with a test case or an entire test suite, but test effectiveness does not yield a universal interpretation.

Some people misconceive the meaning of coverage measures and confuse coverage with effectiveness. More properly, coverage is relative to the tests themselves and measures their thoroughness in exercising the reference model *RM*. Being systematic and trying not to leave either element of code or of the specification untested is certainly a prudent norm. but should be properly understood for what it is.

A real measure of test effectiveness should be relative to the program and should allow testers to quantify the effect of testing on the program's attribute of interest, so that the test process can be kept under control.

7.3 Measures for Monitoring the Testing Process

We have already mentioned that one intuitive and widely used practice is to count the number of failures or faults detected. The test criterion that found the highest number could be deemed the most useful. Even this measure has drawbacks: as tests are gathered and more and more faults are removed, what can we infer about the resulting quality of the tested program? For instance, if we continue testing and no new faults are found for a while, what does this imply? That the program is "correct" or that the tests are ineffective?

It is possible that several different failures are caused by a single fault, as well as that a single failure is caused by different faults. What should be better estimated in a program: its number of contained "faults" or how many "failures" it exposed? Either estimate taken alone can be tricky: if failures are counted it is possible to end the testing with a pessimistic estimate of program "integrity," as one fault may produce multiple failures. On the other hand, if faults are considered, we could evaluate at the same level harmful faults that produce frequent failures, and inoffensive faults that would remain hidden for years of operation. It is hence clear that the two estimates are both important during development and are produced by different (complementary) types of analysis.

The most objective measure is a statistical one: if the executed tests can be taken as a representative sample of program behavior, then we can make a statistical prediction of what would happen for the next tests, should we continue to use the program in the same way. This reasoning is the basis of software reliability.

Documentation and analysis of test results require discipline and effort, but form an important resource of an organization for product maintenance and for improving future projects.

8. Test Process

We have seen that there exist various test objectives, many test selection strategies and differing stages of the lifecycle of a product during which testing can be applied. Before actually commencing any test derivation and execution, all these aspects must be organized into a coherent framework. Indeed, software testing itself consists of a compound process for which different models can be adopted.

A traditional test process includes sequential phases, namely test planning, test design, test execution, and test results evaluation.

8.1 Test Planning

Test planning is the very first phase and outlines the scope of testing activities, focusing in particular on the objectives, resources and schedule, i.e., it covers more the managerial aspects of testing, rather than the detail of techniques and the specific test cases. A test plan can already be prepared during the requirements specification phase.

8.2 Test Design

Test design is a crucial phase of software testing, in which the objectives and the features to be tested and the test suites associated with each of them are defined ([29], [30]). Also the levels of testing are planned. Then, it is decided what kind of approach will be adopted at each level and for each feature to be tested. This also includes deciding a stopping rule for testing. Due to time or budget constraints, at this point it can be decided that testing will concentrate on some of the more critical parts.

Specifically, the following test design sub-steps can be identified:

- ***Establishing test objectives*** — The test objectives, the features and combinations of features that will be the objects of the testing are identified and classified into a hierarchy. In particular, for each feature both the reference to the associated requirements in the requirements specification or design description and a specific test group must be fixed to ensure test traceability.
- ***Define the test case specification*** — The refinements to the approach identified in the previous sub-step are developed and the test cases are consequently individuated.
- ***Design test procedures*** — Using the available information, for instance, the requirements documentation or the test cases specification, for each test group the corresponding test procedures are established and defined. A test procedure provides a detailed description of the steps to be followed for test preparation and execution.
- ***Define pass/fail criteria*** — The expected result for each test procedure, or more generally the criteria to be used to determine whether a test procedure has passed or failed, are also decided.

An emerging and quite different practice for testing is test-driven development, also called *Test-First programming*, which focuses on the derivation of (unit and acceptance) tests before coding. This approach is a key practice of modern Agile development approaches such as *Extreme Programming* (XP) and *Rapid Application Development* (RAD) [9]. The leading principle of such approaches is to make development more lightweight by keeping design simple and reducing as much as possible the rules and the activities of traditional processes felt by developers to be overwhelming and unproductive, for instance documentation, formalized communication, or advance planning of rigid milestones. Therefore, a traditional test design phase as described above no longer exists, but new tests are continuously created, as opposed to a vision of designing test suites up front. In the XP paradigm, the leading principle is to "code a little, test a little ..." so that developers and customers can get immediate feedback.

8.3 Test Execution

Executing the test cases specified in test design may entail various difficulties. Below we discuss the various activities implied in launching the tests, and deciding the test outcome. We also hint at tools for automating testing activities.

Forcing the execution of the test cases (manually or automatically) derived according to one of the criteria presented in Section 5 might not be so obvious.

If a code-based criterion is followed, it provides us with entry-exit paths over the flowgraph that must be taken, and test inputs that execute the corresponding program paths need to be found. Actually, as already said, code-based is better used as an adequacy criterion; hence, in principle, we should not look for inputs ad hoc to execute the entities not covered, but rather use the coverage analysis results to understand the weaknesses in the executed test cases. However, in the cycle of testing, monitoring unexecuted elements, finding additional test cases, often conducted under pressure, finding those test cases that increase coverage can be very difficult.

If a specification-based criterion is adopted, the test cases correspond to sequences of events, which are specified at the abstraction level of the specifications; more precisely, they are labels within the signature of the adopted specification language. To derive concrete test cases, these labels must be translated into corresponding labels at code level (e.g., method invocations), and eventually into execution statements to be launched on the user interface of the test tool being used.

In addition to translating the specified test cases into executable runs, another requirement is the ability to put the system into a state from which the specified tests can be launched. This is sometimes referred to as the *test precondition*. In synchronous systems, before a specific command can be executed, several runs in sequence are required to put the system into the suitable test precondition. An effective way to deal with this is to arrange the selected test cases into suitable sequences, such that each test leaves the system in a state that is the precondition for the subsequent test in the sequence. This approach cannot easily scale up to the integration testing of large, complex systems, in which the specified tests involve actions specific to exercise a subsystem. It can be alleviated by always defining the tests at the external interfaces, i.e., as complete I/O sequences.

A new difficulty is added in concurrent systems allowing for nondeterminism. In this case, the system behavior not only depends on the internal status, but also on the interleaving of events with system tasks and other concurrently running systems. When testing reveals a failure, the

task of recreating the conditions that made it occur is termed test *replay*. In the deterministic approach, originally introduced in [17], exact replay is obtained by means of mechanisms that first capture the occurrence of synchronization events and memory access, and then force the same order of events when the test is replayed. Such an approach is clearly highly intrusive, as it requires heavy instrumentation of the system. Otherwise, a more pragmatic approach is to keep repeating a test until the desired sequence is observed (fixing a maximum number of iterations).

An orthogonal problem arises during integration testing, when testing only parts of a larger system. Indeed, the testing task itself requires a large programming effort: to be able to test a piece of a large system we need to simulate the surrounding environment of the piece under test (i.e., the caller and called methods). This is done by developing ad hoc drivers and stubs [54]; some commercial test tools exist than can facilitate these tasks (see Section 7.3).

8.4 Test Documentation

Documentation is an integral part of the formalization of the test process, which contributes to the coordination and control of the testing phase. Several types of documents may be associated with the testing activities ([54], [29]): Test Plan, Test Design Specification, Test Case Specification, Test Procedure Specification, Test Log, and Test Incident or Problem Report. We outline a brief description of each of them, referring to IEEE Standard for Software Test Documentation [29] for a complete description of test documents and of their relationship with one another and with the testing process.

- ***Test plan*** — Defines test items, features to be or not to be tested, approach to be followed (activities, techniques and tool[s] to be used), pass/fail criteria, the delivered documents, tasks to be performed during the testing phase, environmental needs (hardware, communication and software facilities), people and staff responsible for managing, designing, preparing, and executing the tasks, staffing needs, schedule (including milestones, estimation of time required to do each task, period of use of each testing resource).
- ***Test design specification*** — Describes the features to be tested and their associated test set.
- ***Test case specification*** — Defines a test case and the input/output required for executing it as well as any special constraints or inter-case dependencies. A skeleton is depicted in Table 1.
- ***Test procedure specification*** — Specifies the steps and the special requirements that are necessary for executing a set of test cases.
- ***Test log*** — Documents the result of a test execution, including: the observed failures (if any); the information needed for reproducing them, and locating and fixing the corresponding faults; the information necessary for establishing whether the project is complete; any anomalous events. See a summary in Table 2.
- ***Test incident or problem report*** — Provides a description of the incidents including inputs, expected and observed results, anomalies, date and time, procedure steps, environment, attempts to repeat the tests, observations and reference to the test case, and procedure specification and test log.

Table 1: Scheme of a possible test case

TEST CASE SPECIFICATIONS	
Test case ID	The unique identifier associated with the test case
Test item and purpose	The item and features exercised
Input data	The explicit list of the inputs required for executing the test case (value, file database, etc.)
Test case behavior	Description of the expected test case behavior
Output data	The list or the outputs admitted for each feature involved in the test case, possibly associated with tolerance values
Environmental setup	The hardware/software configuration required
Specific procedural requirements	The constraints and the special procedures required
Test case dependencies	The IDs of the test cases that must be executed prior to this test case

- These tables refer to a traditional plan-driven development process. On the other hand, the already mentioned Agile processes prioritize working software and face-to-face communication over comprehensive documentation. In test driven development in particular, the test cases themselves become a "working specification" document.

9. Test Management

The management processes for software development concern different activities mainly summarized in [48]: initiation and scope definition, planning, execution and control, review and evaluation, closure. These activities also concern the management of the test process.

In the *testing phase* a very important component of successful testing is a collaborative attitude towards testing and quality assurance activities. Managers have a key role in fostering a generally favorable reception towards failure discovery during development; for instance, by preventing a mindset of code ownership among programmers, so that they will not feel defensive about or responsible for failures revealed in their code. Moreover, the testing phases could be guided by various aims, for example, risk-based testing, which uses the product risks to prioritize and focus the test strategy; or scenario-based testing, in which test cases are defined based on specified system scenarios.

Test management can be conducted at different levels, so it must be organized, together with people, tools, policies, and measurements, into a well-defined process that is an integral part of the life cycle.

In the testing context, the manager's main activities can be summarized as [38], [53], [54]:

- ***Scheduling the timely completion of tasks.***
- ***Estimation of the effort and the resources needed to execute the tasks*** —An important task in test planning is the estimation of resources required, which means organizing not

only hardware and software tools but also people. Thus the formalization of the test process also requires putting together a test team, which can involve internal as well as external staff members. The decision will be determined by consideration of costs, schedule, maturity level of the involved organization, and the criticality of the application.

- ***Quantification of the risk associated with the tasks***.
- ***Effort/Cost estimation*** — The testing phase is a critical step in product development, often responsible for the high costs and effort required for product release. The effort can be evaluated for example in terms of person-days, months, or years necessary for the realization of each project. For cost estimation it is possible to use two kinds of models: static and dynamic multivariate models. The former use historical data to derive empirical relationships; the latter project future resource requirements as a function of time. In particular, these test measures can be related to the number of tests executed or the number of tests failed.
- ***Test asset reuse*** — To carry out testing or maintenance in an organized and cost/effective way, the assets/means used to test each part of the system should be reused systematically. This repository of test materials must be configuration-controlled, so that changes to system requirements or design can be reflected in changes to the scope of the tests conducted. The test solutions adopted for testing some application types under certain circumstances, with the motivations behind the decisions taken, form a test pattern which can itself be documented for later reuse in similar projects.

Table 2: Scheme of a possible test log

TEST LOG	
Test log ID	The unique identifier associated with the test log
Items tested	Details of the items tested including environmental attributes
Events	The list of the events occurred including: • The start and end date and time of each event • Identification of the test procedures executed • Personnel who executed the procedures • Description of test procedures results • Environmental details • Description of the anomalous events that occurred

- ***Quality control measures to be employed*** — Several measures relative to the resources spent on testing, as well as to the relative fault-finding effectiveness of the various test phases, are used by managers to control and improve the test process. These test measures may cover such aspects as: number of test cases specified, number of test cases executed, number of test cases passed, and number of test cases failed, among others. Evaluation of test problem reports can be combined with root-cause analysis to evaluate

test process effectiveness in finding faults as early as possible. Such an evaluation could be associated with the analysis of risks.

Moreover, the resources that are worth allocating to testing should be commensurate with the use/criticality of the application: specifically, a decision must be made as to how much testing is enough and when a test stage can be terminated. Thoroughness measures, such as achieved code coverage or functional completeness, as well as estimates of fault density or of operational reliability, provide useful support, but are not sufficient in themselves. The decision also involves considerations about the costs and risks incurred by potential remaining failures, as opposed to the costs implied by continuing to test.

10. Test Tools

Testing requires fulfilling many labor-intensive tasks, running numerous executions, and handling a great amount of information. Appropriate tools can alleviate the burden of clerical, tedious operations, and make them less error-prone. Sophisticated tools can support test design, making it more effective. Besides, many of the surveyed test techniques call for activities that require such massive effort that the application of the techniques is not feasible without the assistance of automated support tools.

Managers and testers are responsible to select those tools that will be the most useful to their organization and processes. This is a very important task, as tool selection greatly affects testing efficiency and effectiveness. Usually selection depends on requirements such as the necessity of standardized interfaces and messages for testing, the ability to record, replay and manage test scripts, the ability to simulate the interacting systems or deploy and test the SUT, the ability to run regression or stress tests, the verification of the components, suitability for the adopted test process, and the support of security requirements. The ISO Standard "Information Technology - Guideline for the evaluation and selection of CASE tools" [32] covers the topic in depth, and specifically also lists suitable characteristics for testing tools used for verification and validation, while a selection of commercial tools can be found in [1].

The field is so active and in such continuous evolution, that it would be impossible to compile here a comprehensive list of existing commercial and academic tools. In the rest of this section, we provide testers with a taxonomy of most commonly used tools:

1. ***Test harness (drivers, stubs)*** [48] — Provides a controlled environment in which tests can be launched and the test outputs can be logged. In order to execute parts of a system, drivers and stubs are provided to simulate caller and called modules, respectively.

2. ***Test generators*** [48] — Provide assistance in the generation of tests. The generation can be random, path wise (based on the flowgraph) or functional (based on the formal specifications). Others exploit the formal specification of a system to derive a set of test cases or generating tests on the fly [57].

3. ***Capture/Replay*** — This type of tool automatically re-executes or replays, previously run tests of which it recorded inputs and outputs (e.g., screens) [1].

4. ***Oracle/file comparators/assertion checking*** [53] — These kinds of tools assist in deciding whether a test outcome is successful or faulty.

5. ***Coverage analyzer/instrumenter*** [48] — A *coverage analyzer* assesses which and how many entities of the program flowgraph have been exercised amongst all those required

by the selected coverage testing criterion. The analysis can be done thanks to program instrumenters, which insert probes into the code. A survey of existing coverage-based testing tools can be found in [66].

6. ***Tracers*** — Trace the history of execution of a program [49].
7. ***Reliability evaluation tools*** [44] —Support test results analysis and graphical visualization in order to assess reliability related measures according to selected models.
8. ***Model checkers*** — Provide counterexamples to violated properties specified in the model. Some of them can also generate a set of test cases. A survey of available tools is provided in [23].
9. **Mutation testing tools** — Implement the basic steps and structure of mutation analysis. Evaluation of test case effectiveness and trend analyses is also provided. A survey of existing approaches and facilities is provided in [33].

11. Conclusions

We have overviewed the fundamentals of software testing, highlighting the most important techniques and approaches applicable during the process lifecycle. Our intent has been to provide the readers with a comprehensive reference guide that could be useful for planning, managing, and executing testing activities. Thus, the approaches overviewed include more traditional techniques, e.g., code-based criteria, as well as more recent ones such as usability, adaptive random testing, model-based testing, and model checking.

As is apparent from the reading, software testing is a complex and effort-intensive activity. It involves many topics and tasks and deserves a first-class role in software development, in terms of both resources and intellectual requirements. For this, the attention of academia is monitoring the feedback provided from industrial context so as to realize better and more useful solutions. Indeed, research activity is evolving every day, and tools and automatic facilities to aid in the solution of specific problems are continuously being provided.

In this document, we highlighted the relevant issues and open questions, so to attract further interest from academia and industry in contributing to the evolution of the state of the art on the many remaining open issues.

Over the years, software testing has evolved from an "art" [47] to an engineering discipline, as the standards, techniques and tools cited throughout the chapter demonstrate. However, test practice inherently still remains a trial-and-error methodology. We will never find a test approach that is guaranteed to deliver a "perfect" product, regardless of the effort we employ. However, what we can and must pursue is to continue to transform testing from "trial-and-error" to a systematic, cost-effective, and predictable engineering discipline.

It is our wish that industry and academia work together to close the large gap still existing between the research and the practice of software testing, towards the ultimate dream of an "efficiency-maximized test engineering" as foreseen in [10].

References

[1] Aptest (Applied Testing and Technology, Inc.), "Software QA Testing and Test Tool Resources," [http://www.aptest.com/resources.html].

[2] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using Static Analysis to Find Bugs," *IEEE Software*, vol. 25, no. 5, 2008, pp. 22-29.

[3] R. Bache and M. Mullerburg, "Masures of Testability as a Basis for Quality Assurance," *Software Engineering Journal*, vol. 5, no. 2, Mar 1990, pp. 86-92.

[4] T. Ball, "The Concept of Dynamic Analysis," *Proc. of Joint 7th ESEC/7th ACM FSE,* Toulouse, France, vol. 24, no. 6, October 1999, pp. 216–234.

[5] L.Baresi and M. Young, "Test Oracles," Tech Report CIS-TR-01-02,.cs.uoregon.edu/~michal/pubs/oracles.html.

[6] R. Barták, "On-line Guide to Constraint Programming," Prague, [http://ktiml.mff.cuni.cz /~bartak/constraint], 1998.

[7] F. Basanieri, A. Bertolino, and E. Marchetti, "The Cow_Suite Approach to Planning and Deriving Test Suites in UML Projects," *Proc. 5th Int. Conf. UML 2002*, Dresden, Germany, LNCS 2460, 2002, pp. 383-397.

[8] V. R. Basili and R.W. Selby, "Comparing the Effectiveness of Software Testing Strategies," *IEEE Trans. Software Eng.* vol. 13, no. 12, 1987, pp. 1278-1296.

[9] K. Beck, *Test-Driven Development by Example,* Addison-Wesley, Reading, MA, November, 2002.

[10] A. Bertolino, "Software Testing Research: Achievements, Challenges, Dreams." In 2007 Future of Software Engineering (FOSE '07). IEEE Computer Society, Washington, DC, USA, 85-103.

[11] A. Bertolino, "A Guided Tour of Four Decades of a Software Testing Discipline" (Abstract), in *SEAA 2008 - EUROMICRO Conf. Software Engineering and Advanced Applications* (Parma, 3-5 September 2008).

[12] Bertolino, and M. Marré, "A General Path Generation Algorithm for Coverage Testing," *Proc. 10th Int. Soft. Quality Week,* San Francisco, Ca. paper. 2T1, 1997.

[13] Bertolino, and L. Strigini, "On the Use of Testability Measures for Dependability Assessment," *IEEE Trans. Software Eng.*, vol. 22, no. 2, 1996, pp. 97-108.

[14] P. Bourque and R. Dupuis, "Guide to the Software Engineering Body of Knowledge, 2004 Version," *SWEBOK,* IEEE CS, 2004, [http://www.computer.org/portal /web /swebok].

[15] E. Brinksma, and J. Tretmans, "Testing Transition Systems: An Annotated Bibliography," *Proc. of MOVEP'2k,* Nantes, 2000, pp. 44-50.

[16] C. Cadar, P. Godefroid, S. Khurshid, C. S. Pasareanu, K. Sen, N.Tillmann, and W. Visser, "Symbolic Execution for Software Testing in Practice: Preliminary Assessment." *Proc. of ICSE 2011*, Waikiki, Honolulu, HI, USA, May, 201, pp. 1066-1071.

[17] R. H. Carver, and K.C Tai, "Use of Sequencing Constraints for Specification-Based Testing of Concurrent Programs," *IEEE Trans. on Soft. Eng*, vol. 24, no. 6, 1998, pp. 471-490.

[18] T. Y. Chen, F. C. Kuo, R. G. Merkel, and T. H. Tse. Adaptive Random Testing: The ART of Test Case Diversity, *J. Syst. Software.* 83, vol. 1. Jan. 2010, pp. 60-66.

[19] T. Y. Chen, Y. T. Yu, "On the Relationship between Partition and Random Testing," *IEEE Trans. on Soft. Eng*, vol. 20, no. 12, pp. 977–980, 1994.

[20] I. Ciupa, A. Pretschner, M. Oriol, A. Leitner, and B. Meyer, "On the Number and Nature of Faults Found by Random Testing," *Software Testing, Verification and Reliability Journal*, vol. 22, no. 1, 2011, pp. 3-28.

[21] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*, MIT Press Cambridge, MA, USA, 2000.

[22] E. W. Dijkstra,"Notes on Structured Programming," *T. H. Rep. 70-WSK03, 1970*, [http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF].

[23] G. Fraser, F. Wotawa, and P. E. Ammann, "Testing with Model Checkers: a Aurvey," *Software Testing, Verification and Reliability*, vol. 19, no. 3. John Wiley & Sons, Hoboken, NJ, 2009, pp. 215- 261.

[24] A. Hartman, M. Katara, and S. Olvovsky, "Choosing a Test Modeling Language: A Survey," Proc. 2nd Int. Haifa Verification Conference on Hardware and Software, Verification and Testing, LNCS 4383, Springer, 2006, pp. 204–218.

[25] K. J. Hayhurst, D. S. Veerhusen, J. J. Chikenski, and L. K. Rierson, "A Practical Tutorial on Modified Condition/Decision Coverage," NASA/TM-2001-210876, May 2001.

[26] R. Hierons and J. Derrick, (Eds) "Special Issue on Specification-based Testing," *Soft. Testing, Verification and Reliability*, vol. 10, 2000.

[27] A. Holzinger, "Usability Engineering Methods for Software Developers," Commun. ACM, vol. 48, no.1, 2005, pp. 71-74.

[28] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std 610.12-1990.

[29] *IEEE Standard for Software Test Documentation*, IEEE Std 829-1998.

[30] *IEEE Standard for Software Unit Testing* IEEE Std. 1008-1987 (R1993).

[31] *IEEE Standard: Guide for Developing Software life-cycle Processes,* IEEE Std 1074-1995.

[32] *Information Technology - Guideline for the Evaluation and Selection of CASE Tools,* ISO/IEC 1462-1998.

[33] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Trans. on Software Engineering*, vol. 99, 2010.

[34] R. Hala, and R. Majumdar, "Software Model Checking," *ACM Comput. Surv.* 41, 4, Article 21 (October 2009), 54 pages.

[35] P.C. Jorgensen, *Software Testing – a Craftsman's Approach*, CRC Press, New York, 1995.

[36] N. Juristo, A.M. Moreno, and S. Vegas, "Reviewing 25 Years of Testing Technique Experiments," *Empirical Software. Engineering Journal*, vol. 9, no. 1/2, March 2004, pp. 7-44.

[37] S.H. Kan, "Metrics and Models in Software Quality Engineering," Addison-Wesley, Reading, MA, 2002.

[38] C. Kaner, J. Falk, and H.Q. Nguyen, *Testing Computer Software*, 2nd Edition, John Wiley & Sons, Hoboken, NJ, April, 1999.

[39] C. Kaner, J. Bach, and B. Pettichord, *Lessons Learned in Software Testing*. Wiley Computer Publishing, Hoboken, NJ, 2001.

[40] B. Korel, "Automated Software Test Data Generation," *IEEE Trans. Software Eng.*, vol. 16, no. 8, 1990, pp. 870-879.

[41] Y. Labiche, P. Thévenod-Fosse, H. Waeselynck, and M. H. Durand, "Testing Level for Object-Oriented Software," *Proceedings of ICSE*, Limerick, Ireland, June 2000, pp. 136-145.

[42] J.C. Laprie, "Dependability - Its Attributes, Impairments and Means," *Predictably Dependable Computing Systems*, B. Randell, J.C. Laprie, H. Kopetz, B. Littlewood, eds., Springer, Berlin, 1995.

[43] D. Latella, and M. Massink, "On Testing and Conformance Relations for UML Statechart Diagrams Behaviours" *Symposium on Soft.Testing and Analysis ISSTA 2002*, Roma, Italy July 2002.

[44] M. R Lyu, ed., *Handbook of Software Reliability Engineering*, McGraw-Hill, New York, 1996.

[45] P. McMinn, "Search-Based Software Test Data Generation: A Survey," *Software Testing, Verification and Reliability,* vol. 14, no. 2, pp. 105-156, 2004.

[46] H. Muccini, A. Bertolino, P. Inverardi, "Using Software Architecture for Code Testing," *IEEE Transactions on Software Engineering*, vol. 30, no. 3, pp. 160 - 170, March 2004.

[47] G.J. Myers, *The Art of Software Testing*, Wiley & Sons, Hoboken, NJ, 1979.

[48] K. Naik and P. Tripathy, eds., "Software Testing and Quality Assurance: Theory and Practice," Wiley & Sons, Hoboken, NJ, 2008.

[49] J. Nielsen, "Usability Engineering," Morgan Kaufmann, San Francisco, 1993.

[50] T. J. Ostrand and M. J. Balcer, "The Category-Partition Method for Specifying and Generating Functional Tests," ACM Comm., vol. 31, no. 6, 1988, pp. 676-686.

[51] R. Pargas, M. J. Harrold, and R. Peck, "Test-Data Generation Using Genetic Algorithms," *J. of Soft. Testing, Verification and Reliability*, vol. 9, 1999, pp. 263-282.

[52] W. W. Peng, and D. R. Wallace, "Software Error Analysis," *NIST SP 500-209*, National Institute of Standards and Technology, Gaithersburg, MD, 20899, [http://hissa.nist.gov /SW ERROR/], December, 1993.

[53] W. Perry, *Effective Methods for Software Testing*, Wiley & Sons. Hoboken. NJ, 3rd ed.. 2006.

[54] S. Rapps and E.J. Weyuker, "Selecting Software Test Data Using Data Flow Information", *IEEE Trans. Software Eng.*, vol.11, 1985, pp. 367-375.

[55] K. Sen, D. Marinov, and G. Agha, "CUTE: a Concolic Unit Testing Engine for C", *ACM Proc. ESEC-FSE 2005*, New York, NY, ACM, 2005, pp. 263-272.

[56] M. Shafique and Y. Labiche, "A Systematic Review of Model Based Testing Tool Support" Carleton University, *Tech. Rep. SCE-10-04*, 2010 [http://squall.sce.carleton.ca /pubs /tech _report/TR_SCE-10-04.pdf].

[57] S. R. Shahamiri, W. M. N. W. Kadir, and S. Z. Mohd-Hashim, "A Comparative Study on Automated Software Test Oracle Methods", Proc of 4th. ICSEA 2009, Porto, Portugal, 2009, pp. 140-145.

[58] J. Tretmans, "Model-Based Testing and Some Steps towards Test-Based Modelling," *Journal of Formal Methods for Eternal Networked Software Systems*, 2011, pp.297-326.

[59] J. Warmer, and A. Kleppe, *Object Constraint Language, The, Getting Your Models Ready for MDA*, Second Edition, Addison Wesley, 2003.

[60] E. J. Weyuker, "Translatability and Decidability Questions for Restricted Classes of Program Schemas" *SIAM J. on Computers*, vol. 8, no. 4, 1979. pp. 587-598.

[61] E. J. Weyuker, "On Testing Non-testable Programs," *The Computer Journal*, vol. 25, no.4, 1982, pp. 465-470.

[62] E. J. Weyuker, T. J. Ostrand, and R.M. Bell, "Comparing the Effectiveness of Several Modeling Methods for Fault Prediction", *Empirical Software Engineering*, vol. 15, no.3. 2010, pp. 277-295.

[63] C. W. Wharton, J. Reiman, C. Lewis, and P. Polson, "The Cognitive Walkthrough Method: A Practitioner's Guide," in J. K. Nielsen, & R. L. Mack (eds.), Usability Inspection Methods, Wiley, New York, 1994.

[64] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal Methods: Practice and Experience," *ACM Computing Surveys (CSUR)*, vol. 41, no. 4, 2009. pp. 1-36.

[65] Q. Yang, J. J. Li, and D. J. Weiss, "A Survey of Coverage Based Testing Tools", *Proc. of ASE 2006*, 2006. pp. 99-103.

[66] S. Yoo, and M. Harman, "Regression Testing Minimization, Selection and Prioritization : a Survey," *Journal of Software Testing, Verification and Reliability*, ed., Wiley, 2010.

Chapter 4.2

Essentials of Software Engineering Testing

Richard Hall Thayer and Merlin Dorfman

This is the fourth chapter in a reference guide to aid individual software engineers in a greater understanding the IEEE SWEBOK [2013] and in passing the IEEE CSDP/CSDA certification exams. The chapter introduces software engineering *testing.*

This module provides an introduction to software testing. Topics covered include basic definitions of testing, validation and verification; the levels of testing from unit testing through to acceptance testing; the relationship with requirements and design specifications; and test documentation.

This list of exam specifications is reported to be the same list that the exam writers used to write the exam questions. Therefore it is the best source of help for the exam takers.

Chapter 4 covers the CSDP exam specifications for the software test module [Software Exam Specification, Version 2, 18 March 2009]:

1. Software testing fundamentals (testing-related terminology; key issues [test selection criteria, test adequacy criteria, testing effectiveness, testing for defect identification, oracle problem, limitations, infeasible paths, testability]; relationship)
2. Test levels (the target of the tests [unit testing, integration testing, system testing]; objectives of testing)
3. Test techniques (based on the tester's intuition and experience; specification-based; code-based; fault-based; usage-based; based on nature of application; selecting and combining techniques)
4. Human-computer user interface testing and evaluation (the variety of aspects of usefulness and usability); Heuristic evaluation; cognitive walkthroughs; user testing approaches [observation sessions, and so on]; web usability, testing techniques for web)
5. Test-related measures (evaluation of the program under test; evaluation of the tests performed)
6. Test process (practical considerations [attitudes, egoless, test guides, test process management, test documentation, independence, cost/effort estimation, termination, test reuse and patterns]; test activities; software testing tools)

Software testing is the process of executing a program or system with the intent of finding errors. Software is not unlike other physical processes where inputs are received and outputs are produced. Where software differs is in the manner in which it fails. Most physical systems fail in a fixed (and reasonably small) set of ways. By contrast, software can fail in many bizarre ways. Detecting all of the different failure modes for software is generally infeasible.

Unlike most physical systems, most of the defects in software are design errors, not manufacturing defects. Software does not suffer from corrosion or wear-and-tear; generally it will not change until upgrades, or until obsolescence. So once the software is shipped, the design defects, or bugs, will be buried and remain latent until activation.

Software bugs will almost always exist in any software module of moderate size: not because programmers are careless or irresponsible, but because the complexity of software is generally intractable, and humans have only limited ability to manage complexity. It is also true that for any complex system, design defects can never be completely ruled out.

Discovering the design defects in software is equally difficult, for the same reason as complexity. Because software and any digital systems are not continuous, testing boundary values is not sufficient to guarantee correctness. All the possible values need to be tested and verified, but complete testing is infeasible. Exhaustively testing a simple program to add only two integer inputs of 32-bits (yielding 2^{64} distinct test cases) would take hundreds of years. even if tests were performed at a rate of thousands per second. Obviously, for a realistic software module, the complexity can be far beyond the example mentioned here. If inputs from the real world are involved, the problem will get worse, because timing and unpredictable environmental effects and human interactions are all possible input parameters under consideration.

A further complication has to do with the dynamic nature of programs. If a failure occurs during preliminary testing and the code is changed, the software may now work for a test case that it didn't work for previously. But its behavior on test cases that it passed before can no longer be guaranteed. To account for this possibility, testing should be restarted. The expense of doing this is often prohibitive.

Regardless of the limitations, testing is an integral part of software development. It is broadly deployed in every phase of the software development cycle. Typically, more than 50 percent of the development time is spent in testing [SWEBOK 2004].

4.1 Software Testing Fundamentals

Software testing is the process of executing a program or system with the intent of finding errors. Software testing cannot "prove" that the software is correct.

4.1.1. Testing-related terminology [http://www.computer.org/portal/web /swebok/html /ch5# References].

- *Testing is* defined as:
  - The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software item [IEEE 829-2007].
  - The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component [IEEE 829-2007].
- ***Faults vs. Failures*** — Many terms are used in the software engineering literature to describe a malfunction, notably fault, failure, error, and several others. It is essential to clearly distinguish between the cause of a malfunction for which the term fault or defect will be used here, and an undesired effect observed in the system's delivered service, which will be called a failure. Testing can reveal failures, but it is the faults that can and must be removed.

  However, it should be recognized that the cause of a failure cannot always be unequivocally identified. No theoretical criteria exist to definitively determine what fault caused the observed failure. It might be said that it was the fault that had to be modified

to remove the problem, but other modifications could have worked just as well. To avoid ambiguity, some authors prefer to speak of failure-causing inputs instead of faults—that is, those sets of inputs that cause a failure to appear [SWEBOK 2004].

4.1.2. Key issues. The following are some of the key issues in software testing [SWEBOK 2004].

- ***Test selection criteria/test adequacy criteria (or stopping rules)*** — A test selection criterion is a means of deciding what a suitable set of test cases should be. A selection criterion can be used for selecting the test cases or for checking whether a selected test suite is adequate—that is, to decide whether the testing can be stopped.
- ***Testing effectiveness/objectives for testing*** — *Testing* is the observation of a sample of program executions. Sample selection can be guided by different objectives: it is only in light of the objectives pursued that the effectiveness of the test set can be evaluated.
- ***Testing for defect identification*** — In testing for defect identification, a successful test is one that causes the system to fail. This is quite different from testing to demonstrate that the software meets its specifications or other desired properties, in which case testing is successful if no (significant) failures are observed. *The authors believe this is incorrect. Testing is for the purpose of finding errors. A demonstration (not a test) can be used to show the software meets its requirements.*
- ***The oracle problem*** — An oracle is any (human or mechanical) agent that decides whether a program behaved correctly in a given test, and accordingly produces a verdict of "pass" or "fail." There exist many different kinds of oracles, and oracle automation can be very difficult and expensive.
- ***Theoretical and practical limitations of testing*** — Testing theory warns against ascribing an unjustified level of confidence to a series of passed tests. Unfortunately, most established results of testing theory are negative ones, in that they state what testing can never achieve as opposed to what it actually achieved. The most famous quotation in this regard is the Dijkstra aphorism that "program testing can be used to show the presence of bugs, but never to show their absence." The obvious reason is that complete testing is not feasible in real software. Because of this, testing must be driven based on risk and can be seen as a risk management strategy.
- ***The problem of infeasible paths*** — Infeasible paths, the control flow paths that cannot be exercised by any input data, are a significant problem in path-oriented testing, and particularly in the automated derivation of test inputs for code-based testing techniques.
- ***Testability*** — The term "software testability" has two related but different meanings: on the one hand, it refers to the degree to which it is easy for software to fulfill a given test criterion, as in [Bache & Müllerberg 1990]; on the other hand, it is defined as the likelihood, possibly measured statistically, that the software will expose a failure under testing if it is faulty, as in [Voas & Miller 1995]. Both meanings are important.

4.1.3. Relationships of testing to other activities. Software testing is related to but different from static software quality management techniques, proofs of correctness, debugging, and programming. However, it is informative to consider testing from the points of view of software quality analysts and of certifiers [SWEBOK 2004].

4.2 Test Levels

4.2.1 The target of the test. Software testing is usually performed at different levels along the development and maintenance processes. That is to say, the target of the test can vary: a single module, a group of such modules (related by purpose, use, behavior, or structure), or a whole system. Four big test stages can be conceptually distinguished, namely *unit, integration, system, and acceptance.* No process model is implied, nor are any of those stages assumed to have greater importance than the other three [SWEBOK 2004].

- ***Unit testing*** — *Unit testing* verifies the functioning in isolation of software pieces that are separately testable. Depending on the context, these could be the individual subprograms or a larger component made of tightly related units. Typically, unit testing occurs with access to the code being tested and with the support of debugging tools, and might involve the programmers who wrote the code.

- ***Integration testing*** — *Integration testing* is the process of verifying the interaction (interfaces) between software components. Classical integration testing strategies, such as top-down or bottom-up, are used with traditional, hierarchically structured software.

  Modern systematic integration strategies are rather architecture-driven, which implies integrating the software components or subsystems based on identified functional threads. Integration testing is a continuous activity, at each stage of which software engineers must abstract away lower-level perspectives and concentrate on the perspectives of the level they are integrating. Thus, failure to do unit testing properly dooms integration testing, since testing will reveal problems that should have been found in unit testing. Time and effort are then wasted fixing defects internal to the units, retesting the units, and rerunning integration tests.

  Except for small, simple software, systematic, incremental integration testing strategies are usually preferred to putting all the components together at once, which is pictorially called "big bang" testing.

- ***System testing*** — *System testing* is concerned with the behavior of a whole system. The majority of functional failures should already have been identified and corrected during unit and integration testing. System testing is usually considered appropriate for comparing the system to the non-functional system requirements, such as security, speed, accuracy, and reliability. (We suggest the reader refer to Chapter 1 to understand the differences between functional and non-functional requirements.) External interfaces to other applications, utilities, hardware devices, or the operating environment are also evaluated at this level.

- ***Acceptance testing*** — *Acceptance testing* checks the system's behavior against the customer's requirements, however these may have been expressed; the customers undertake, or specify, typical tasks to check that their requirements have been met or that the organization has identified them for the software's target market. This testing activity may or may not involve the developers of the system.

4.2.2 Objectives of testing. Testing is conducted to accomplish a specific objective, which is stated more or less explicitly and with varying degrees of precision. Stating the objective in precise, quantitative terms allows control to be established over the test process. Testing can be aimed at verifying different properties. Test cases can be designed to check that the functional

specifications are correctly implemented, which is variously referred to in the literature as conformance testing, correctness testing, or functional testing. However, other nonfunctional properties may be tested as well, including performance, reliability, and usability, among many others.

Other important objectives for testing include (but are not limited to) reliability measurement, usability evaluation, and acceptance, for which different approaches would be taken. Note that the test objective varies with the system under test. In general, different purposes are addressed at different levels of testing.

The sub-topics listed below are those most often cited in the literature as *objectives of testing.* Note that some kinds of testing are more appropriate for custom-made software packages, for example installation testing, while others are more appropriate for generic products, such as beta testing [SWEBOK 2004].

- ***Installation testing*** — Usually after completion of software and acceptance testing, the software can be verified upon installation in the target environment. Installation testing can be viewed as system testing conducted once again according to hardware configuration requirements. Installation procedures may also be verified.
- ***Alpha and beta testing*** — Before the software is released, it is sometimes given to a small, representative set of potential users for trial use, either in-house (alpha testing) or external (beta testing). These users report problems with the product. Alpha and beta use is often uncontrolled, and is not always referred to in a test plan.
- ***Reliability achievement and evaluation*** — In helping to identify faults, testing is a means to improve reliability. By contrast, by randomly generating test cases according to the operational profile, statistical measures of reliability can be derived. An operational profile (OP) is a quantitative characterization of how the software will be used [Musa 1993], where a profile is a set of independent possibilities, called elements, and their associated probability of occurrence. Using reliability growth models, both objectives can be pursued together.

  Software reliability refers to the probability of failure-free operation of a system. It is related to many aspects of software, including the testing process. Directly estimating software reliability by quantifying its related factors can be difficult. Testing is an effective sampling method to measure software reliability. Guided by the operational profile, software testing (usually black-box testing; see definition below) can be used to obtain failure data, and an estimation model can be further used to analyze the data to estimate the present reliability and predict future reliability. Therefore, based on the estimation, the developers can decide whether to release the software, and the users can decide whether to adopt and use the software. Risk of using software can also be assessed based on reliability information. The primary goal of testing should be to measure the dependability of tested software.

  There is agreement on the intuitive meaning of dependable software: it does not fail in unexpected or catastrophic ways. Robustness testing and stress testing are variances of reliability testing based on this simple criterion.

  The robustness of a software component is the degree to which it can function correctly in the presence of exceptional inputs or stressful environmental conditions. Ro-

bustness testing differs from correctness testing in the sense that the functional correctness of the software is not of concern. It only watches for robustness problems such as machine crashes, process hangs, or abnormal termination. The oracle is relatively simple; therefore robustness testing can be made more portable and scalable than correctness testing.

- ***Stress testing*** — Stress testing or load testing is often used to test the whole system rather than the software alone. In such tests, the software or system is exercised with or beyond the specified limits. Typical stress includes resource exhaustion, bursts of activities, and sustained high loads.

- ***Regression testing*** — According to [IEEE 610.12-90] regression testing is the "selective retesting of a system or component to verify that modifications have not caused unintended effects." In practice, the idea is to show that software that previously passed the tests still does. Obviously, a trade-off must be made between the assurance given by regression testing every time a change is made and the resources required to run the test(s).

  Regression testing can be conducted at each of the test levels and may apply to functional and nonfunctional testing.

- ***Correctness testing*** — *Correctness* is the minimum requirement of software, the essential purpose of testing. Correctness testing will need some type of oracle to tell the right behavior from the wrong one. The tester may or may not know the internal details of the software module under test, e.g., control flow, data flow, etc. (See Section 4.3.4.3.) Therefore, either a white-box point of view or black-box point of view can be taken in testing software. We must note that the black-box and white-box ideas are not limited to correctness testing only.

- ***Black-box testing*** — The *black-box approach* is a testing method in which test data are derived from the specified functional requirements without regard to the program structure. It is also termed data-driven, input/output driven or requirements-based testing. Because only the functionality of the software module is of concern, black-box testing also mainly refers to functional testing--a testing method emphasizing executing the functions and examining their input and output data. The tester treats the software under test as a black box--only the inputs, outputs, and specification are visible, and the functionality is determined by observing the outputs to corresponding inputs. In testing, various inputs are exercised and the outputs are compared against specification to validate the correctness. All test cases are derived from the specification. No implementation details of the code are considered.

- ***White-box testing*** — Contrary to black-box testing, in *white-box testing* the structure and flow of the software under test are visible to the tester. Testing plans are made according to the details of the software implementation, such as programming language, logic, and styles. Test cases are derived from the program structure. White-box testing is also called glass-box testing, logic-driven testing or design-based testing.

  There are many techniques available in white-box testing, because the problem of intractability is eased by specific knowledge of and attention to the structure of the software under test. The intention of exhausting some aspect of the software is still strong in white-box testing, and some degree of exhaustion can be achieved, such as executing

each line of code at least once (statement coverage), traversing every branch statement (branch coverage), or covering all the possible combinations of true and false condition predicates (multiple condition coverage).

- ***Boundary between black-box and white-box testing*** —The boundary between the black-box and white-box approaches is not clear-cut. Many testing strategies mentioned above may not be safely classified into black-box testing or white-box testing. This is also true for transaction-flow testing, syntax testing, finite-state testing, and many other testing strategies not discussed in this text. One reason is that all the above techniques will need some knowledge of the specification of the software under test. Another reason is that the idea of specification itself is broad--it may contain any requirement including the structure, programming language, and programming style as part of the specification content.
- ***Random testing*** — We may be reluctant to consider random testing as a testing technique. The test case selection is simple and straightforward: they are randomly chosen. Random testing is more cost effective for many programs. Some very subtle errors can be discovered at low cost, and it is not inferior in coverage to other carefully designed testing techniques. One can also obtain reliability estimates using random testing results based on operational profiles. Effectively combining random testing with other testing techniques may yield more powerful and cost-effective testing strategies.
- ***Performance testing*** — Not all software systems have explicit performance specifications, but every system will have implicit performance requirements. The software should not take infinite time or infinite resources to execute. "Performance bugs" sometimes are used to refer to those design problems in software that cause the system performance to degrade.

  Performance has always been a great concern and a driving force of computer evolution. Performance evaluation of a software system usually includes: resource usage, throughput, stimulus-response time, and queue lengths detailing the average or maximum number of tasks waiting to be serviced by selected resources. Typical resources that need to be considered include network bandwidth requirements, CPU cycles, disk space, disk access operations, and memory usage. The goal of performance testing can be performance bottleneck identification, performance comparison and evaluation, etc. The typical method of doing performance testing is using a benchmark—a program, workload, or trace designed to be representative of the typical system usage.
- ***Security testing*** — Software quality, reliability and security are tightly coupled. Flaws in software can be exploited by intruders to open security holes. With the development of the Internet, software security problems are becoming even more severe.

  Many critical software applications and services have integrated security measures against malicious attacks. The purposes of security testing of these systems include identifying and removing software flaws that may potentially lead to security violations, and validating the effectiveness of security measures. Simulated security attacks can be performed to find vulnerabilities.

4.3 Test Techniques

One of the aims of testing is to reveal as much potential for failure as possible, and many techniques have been developed to do this, which attempt to "break" the program, by running one or

more tests drawn from identified classes of executions deemed equivalent. The leading principle underlying such techniques is to be as systematic as possible in identifying a representative set of program behaviors; for instance, considering subclasses of the input domain, scenarios, states, and dataflow.

It is difficult to find a homogeneous basis for classifying all techniques, and the one used here must be seen as a compromise. The classification is based on how tests are generated from the software engineer's intuition and experience, the specifications, the code structure, the (real or artificial) faults to be discovered, the field usage, or, finally, the nature of the application. Sometimes these techniques are classified as white-box, also called glass box, if the tests rely on information about how the software has been designed or coded, or as black-box if the test cases rely only on the input/output behavior. One last category deals with combined use of two or more techniques. Obviously, these techniques are not used equally often by all practitioners. Included in the list are those that a software engineer should know [SWEBOK 2004].

4.3.1. Based on the software engineer's intuition and experience. Some testing techniques are based on the software engineer's experience [SWEBOK 2004]:

- ***Ad hoc testing*** — Perhaps the most widely practiced technique remains ad hoc testing: tests are derived relying on the software engineer's skill, intuition, and experience with similar programs. Ad hoc testing might be useful for identifying special tests, those not easily captured by formalized techniques.
- ***Exploratory testing*** — Exploratory testing is defined as simultaneous learning, test design, and test execution; that is, the tests are not defined in advance in an established test plan, but are dynamically designed, executed, and modified. The effectiveness of exploratory testing relies on the software engineer's knowledge, which can be derived from various sources: observed product behavior during testing, familiarity with the application, the platform, the failure process, the type of possible faults and failures, the risk associated with a particular product, and so on.

4.3.2 Specification-based techniques. Test techniques are [SWEBOK 2004]:

- ***Equivalence partitioning*** — The input domain is subdivided into a collection of subsets, or equivalent classes, which are deemed equivalent according to a specified relation, and a representative set of tests (sometimes only one) is taken from each class.
- ***Boundary-value analysis*** — Test cases are chosen on and near the boundaries of the input domain of variables, with the underlying rationale that many faults tend to concentrate near the extreme values of inputs. An extension of this technique is robustness testing, wherein test cases are also chosen outside the input domain of variables, to test program robustness to unexpected or erroneous inputs.
- ***Decision table*** — Decision tables represent logical relationships between conditions (roughly, inputs) and actions (roughly, outputs). Test cases are systematically derived by considering every possible combination of conditions and actions. A related technique is cause-effect graphing.
- ***Finite-state machine-based*** — By modeling a program as a finite state machine, tests can be selected in order to cover states and transitions on it.

- ***Testing from formal specifications*** — Stating the specifications in a formal language allows for automatic derivation of functional test cases and, at the same time, provides a reference output, an oracle, for checking test results. Methods exist for deriving test cases from model-based or algebraic specifications.
- ***Random testing*** — Tests are generated purely at random, not to be confused with statistical testing from the operational profile as described earlier.
- ***Operational profile*** — This form of testing falls under the heading of the specification-based entry, since at least the input domain must be known, to be able to pick random points within it.

4.3.3 Code-based techniques. A number of code-based techniques are [SWEBOK 2004]:

- ***Control-flow-based criteria*** — A control-flow-based coverage criterion is aimed at covering all the statements or blocks of statements in a program, or specified combinations of them. Several coverage criteria have been proposed, like condition/decision coverage. The strongest of the control-flow-based criteria is path testing, which aims to execute all entry-to-exit control flow paths in the flow graph. Since path testing is generally not feasible because of loops, other less stringent criteria tend to be used in practice, such as statement testing, branch testing, and condition/decision testing. The adequacy of such tests is measured in percentages; for example, when all branches have been executed at least once by the tests, 100% branch coverage is said to have been achieved.
- ***Data flow-based criteria*** — In data-flow-based testing, the control flow graph is annotated with information about how the program variables are defined, used, and killed (undefined). The strongest criterion, all definition-use paths, requires that, for each variable, every control flow path segment from a definition of that variable to a use of that definition is executed. In order to reduce the number of paths required, weaker strategies such as all-definitions and all-uses are employed.
- ***Reference models for code-based testing (flow graph, call graph)*** — Not a technique in itself; the control structure of a program is graphically represented using a flow graph in code-based testing techniques. A flow graph is a directed graph the nodes and arcs of which correspond to program elements. For instance, nodes may represent statements or uninterrupted sequences of statements, and arcs may represent the transfer of control between nodes.

4.3.4 Fault-based techniques. With different degrees of formalization, fault-based testing techniques devise test cases specifically aimed at revealing categories of likely or predefined faults [SWEBOK 2004].

- ***Error guessing*** — In error guessing, test cases are specifically designed by software engineers trying to figure out the most plausible faults in a given program. A good source of information is the history of faults discovered in earlier projects, as well as the software engineer's expertise.
- ***Mutation testing*** — Also known as *error seeding*. A *mutant* is a slightly modified version of the program under test, differing from it by a small, syntactic change. Every test case exercises both the original and all generated mutants: if a test case is successful in identifying the difference between the program and a mutant, the latter is said to be

"killed." Originally conceived as a technique to evaluate a test set (see 4.2), mutation testing is also a testing criterion in itself: either tests are randomly generated until enough mutants have been killed, or tests are specifically designed to kill surviving mutants. In the latter case, mutation testing can also be categorized as a code-based technique. The underlying assumption of mutation testing, the coupling effect, is that by looking for simple syntactic faults, more complex but real faults will be found. For the technique to be effective, a large number of mutants must be automatically derived in a systematic way.

4.3.5 Usage-based techniques. Some usage based techniques are [SWEBOK 2004]:

- ***Operational profile*** — In testing for reliability evaluation, the test environment must reproduce the operational environment of the software as closely as possible. The idea is to infer, from the observed test results, the future reliability of the software when in actual use. To do this, inputs are assigned a probability distribution, or profile, according to their occurrence in actual operation.
- ***Software Reliability Engineered Testing*** — Software Reliability Engineered Testing (SRET) [Musa 1993] is a testing method encompassing the whole development process, whereby testing is "designed and guided by reliability objectives and expected relative usage and criticality of different functions in the field."

4.3.6 Techniques based on the nature of the application. These techniques apply to all types of software. However, for some kinds of applications, some additional know-how is required for test derivation. A list of a few specialized testing fields is provided here, based on the nature of the application under test [SWEBOK 2004]:

- Object-oriented testing
- Component-based testing
- Web-based testing
- GUI testing
- Testing of concurrent programs
- Protocol conformance testing
- Testing of real-time systems
- Testing of safety-critical systems

4.3.7 Selecting and combining techniques. Some techniques that select and combine other techniques are [SWEBOK 2004]:

- ***Functional and structural*** — Specification-based and code-based test techniques are essentially black-box and white-box techniques and are often contrasted as functional vs. structural testing. These two approaches to test selection are not to be seen as alternative but rather as complementary; in fact, they use different sources of information and have been proven to highlight different kinds of problems. They could be used in combination, depending on budgetary considerations.

- ***Deterministic vs. random*** — Test cases can be selected in a deterministic way, according to one of the various techniques listed, or randomly drawn from some distribution of inputs, such as is usually done in reliability testing. Several analytical and empirical comparisons have been conducted to analyze the conditions that make one approach more effective than the other.

4.4 Human Computer User Interface Testing and Evaluation

4.4.1 Human computer user interface (HCI) testing and evaluation (the variety of aspects of usefulness and usability). HCI interface testing includes methods of measuring usability and the study of the principles behind an object's perceived efficiency or elegance. In human-computer interaction and computer science, usability studies the elegance and clarity with which the interaction with a computer program or a web site (web usability) is designed. However, an even more basic requirement is that the user interface be *useful*, i.e., that it allow the user to complete relevant tasks.

4.4.2 Heuristic evaluation. A *heuristic evaluation is* a usability inspection method for computer software that helps to identify usability problems in the user interface (UI) design. It specifically involves evaluators examining the interface and judging its compliance with recognized usability principles (the "heuristics"). *Usability inspection* is the name for a set of methods where an evaluator inspects a user interface. This is in contrast to usability testing where the usability of the interface is evaluated by testing it on real users. Usability inspections can generally be used early in the development process by evaluating prototypes or specifications for the system that can't be tested on users. Usability inspection methods are generally considered to be cheaper to implement than testing on users [http://en.wikipedia.org /wiki/Heuristic_evaluation].

4.4.3 Cognitive walkthroughs. The *cognitive walkthrough* method is a usability inspection method used to identify usability issues in a piece of software or on a web site, focusing on how easy it is for new users to accomplish tasks with the system. Whereas cognitive walkthrough is task-specific, heuristic evaluation takes a holistic view to catch problems not caught by this and other usability inspection methods. The method is rooted in the notion that users typically prefer to learn a system by using it to accomplish tasks rather than, for example, studying a manual. The method is prized for its ability to generate results quickly with low cost, especially when compared to usability testing, as well as the ability to apply the method early in the design phases, before coding has even begun [http://en.wikipedia.org/wiki/Cognitive_walkthrough].

4.4.4 User testing approaches. Usability testing is the process by which the human-computer interaction characteristics of a system are measured, and weaknesses are identified for correction. Such testing can range from rigorously structured to highly informal, from quite expensive to virtually free, and from time-consuming to quick. While the amount of improvement is related to the effort invested in usability testing, all of these approaches lead to better systems [Levi & Conrad 2008].

Observational methods involve an investigator viewing users as they work in a field study, and taking notes on the activity that takes place. Observation may be either direct, where the investigator is actually present during the task, or indirect, where the task is viewed by some other means such as through use of a video recorder. The method is useful early in the user requirements specification phase for obtaining qualitative data. It is also useful for studying currently executed tasks and processes.

The benefit is that the observer can view what users actually do in context. *Direct observation* allows the investigator to focus attention on specific areas of interest. Indirect observation captures activity that would otherwise have gone unrecorded or unnoticed [http://www.Usability net.org/tools/userobservation.htm].

A *scenario-based usability test* involves presenting representative end-users with scenarios, or specific tasks, designed to cover the major functionality of the software system and to simulate expected real-life usage patterns. Such scenarios should be formulated by knowledgeable task experts in consultation with the system designers. Results are then tabulated using such measures as whether the participants correctly accomplished the tasks, the time taken for each task, and the number of pages accessed for each task [Levi & Conrad 2008].

Mining the Logs usability evaluation evaluates web server logs as a valuable source of information about usage patterns. Usability evaluation need not end with a system's release. Standard Web servers, or httpd logs, are an invaluable source of information about usage patterns once a Web site has gone live. At this point, the testers need not find usability experts or representative users; real users' sessions are captured in great detail and are available for analysis [Levi & Conrad 2008].

Usability testing can be performed with developers, HCI experts, or representative end users. Some authors distinguish between "testing," which they limit to empirical end-user oriented methods, and "evaluation," which utilizes HCI professionals' expertise [Levi & Conrad 2008].

4.4.5 Web usability. Usability can be defined as the degree to which a given piece of software assists the person sitting at the keyboard to accomplish a task, as opposed to becoming an additional impediment to the accomplishment. The broad goal of usable systems is often assessed using several criteria [Levi & Conrad 2008]:

- Ease of learning
- Retention of learning over time
- Speed of task completion
- Error rate
- Subjective user satisfaction

4.4.6 Testing techniques for the web. There are three main styles of testing for the web. *Exploratory testing* examines a system and looks for areas of user confusion, slow-down, or mistakes. Such testing is performed with no particular preconceived notions about where the problems lie or what form they may take. The deliverable for an exploratory test is a list of problem areas for further examination: "users were visibly confused when faced with page *x*; only half the users were able to complete task *y*; task *z* takes longer than it should." Exploratory testing can be used at any point in the development life cycle, but is most effective when implemented early and often.

Threshold testing measures the performance characteristics of a system against predetermined goals. This is a pass/fail effort: "With this system users were able to complete task *x* in *y* seconds, making an average of *z* mistakes. This does (does not) meet the release criteria." Threshold testing typically accompanies a beta release.

Finally, *comparison testing* measures the usability characteristics of two approaches or designs to determine which better suits users' needs. This is usually done at the early prototyping stage [Levi & Conrad 2008].

4.5 Test-related Measures

Sometimes test techniques are confused with test objectives. Test techniques are to be viewed as aids that help to ensure the achievement of test objectives. For instance, branch coverage is a popular test technique. Achieving a specified branch coverage measure should not be considered the objective of testing per se: it is a means to improve the chances of finding failures by systematically exercising every program branch out of a decision point. To avoid such misunderstandings, a clear distinction should be made between test-related measures, which provide an evaluation of the program under test based on the observed test outputs, and those that evaluate the thoroughness of the test set.

Measurement is usually considered instrumental to quality analysis. Measurement may also be used to optimize the planning and execution of the tests. Test management can use several process measures to monitor progress.

4.5.1. Evaluation of the program under test. The following are methods of evaluating programs under test [SWEBOK 2004]:

- ***Program measurement to aid in planning and designing testing*** — Measures based on program size (for example, source lines of code or function points) or on program structure (like complexity) are used to guide testing. Structural measures can also include measurements among program modules in terms of the frequency with which modules call each other [SWEBOK 2004].
- ***Fault types and classification, and statistics***. The testing literature is rich in classifications and taxonomies of faults. To make testing more effective, it is important to know which types of faults could be found in the software under test, and the relative frequency with which these faults have occurred in the past. This information can be very useful in making quality predictions, as well as for process improvement. An IEEE standard exists on classification of software "anomalies" [SWEBOK 2004; IEEE1044-1993].
- ***Fault density*** — A program under test can be assessed by counting and classifying the discovered faults by their types. For each fault class, fault density is measured as the ratio of the number of faults found to the size of the program [SWEBOK 2004].
- ***Life test, reliability evaluation*** — A statistical estimate of software reliability, which can be obtained by reliability achievement and evaluation, can be used to evaluate a product and decide whether or not testing can be stopped [SWEBOK 2004].
- ***Reliability growth models*** — Reliability growth models provide a prediction of reliability based on the failures observed under reliability achievement and evaluation They assume, in general, that the faults that caused the observed failures have been fixed (although some models also accept imperfect fixes), and thus, on average, the product's reliability exhibits an increasing trend. There now exist dozens of published models. Many are laid down on some common assumptions, while others differ. Notably, these models are divided into failure-count and time-between-failure models [Musa 2005].

4.5.2. Evaluation of the tests performed. The following are methods of evaluating tests performed [SWEBOK 2004]:

- ***Coverage/thoroughness measures*** — Several test adequacy criteria require that the test cases systematically exercise a set of elements identified in the program or in the specifications. To evaluate the thoroughness of the executed tests, testers can monitor the elements covered, so that they can dynamically measure the ratio between covered elements and their total number. For example, it is possible to measure the percentage of covered branches in the program flow graph, or that of the functional requirements exercised among those listed in the specifications document. Code-based adequacy criteria require appropriate instrumentation of the program under test [IEEE 982.1-1988].
- ***Fault seeding*** — Some faults are artificially introduced into the program before testing. When the tests are executed, some of these seeded faults will be revealed, and possibly some faults that were already there will be as well. In theory, depending on which of the artificial faults are discovered, and how many, testing effectiveness can be evaluated, and the remaining number of genuine faults can be estimated. In practice, statisticians question the distribution and representativeness of seeded faults relative to genuine faults and the small sample size on which any extrapolations are based. Some also argue that this technique should be used with great care, since inserting faults into software involves the obvious risk of leaving them there.
- ***Mutation score*** — In mutation testing (see earlier discussion) the ratio of killed mutants to the total number of generated mutants can be a measure of the effectiveness of the executed test set.
- ***Comparison and relative effectiveness of different techniques*** — Several studies have been conducted to compare the relative effectiveness of different test techniques. It is important to be precise as to the property against which the techniques are being assessed; what, for instance, is the exact meaning given to the term "effectiveness"? Possible interpretations are: the number of tests needed to find the first failure, the ratio of the number of faults found through testing to all the faults found during and after testing, or how much reliability was improved. Analytical and empirical comparisons between different techniques have been conducted according to each of the notions of effectiveness specified above.

4.6 Test Process

Testing concepts, strategies, techniques, and measures need to be integrated into a defined and controlled process that is run by people. The test process supports testing activities and provides guidance to testing teams, from test planning to test output evaluation, in such a way as to provide justified assurance that the test objectives will be met cost-effectively [SWEBOK 2004].

4.6.1. Practical considerations. Some practical test processes are [SWEBOK 2004:

- ***Attitudes/Egoless programming*** — A very important component of successful testing is a collaborative attitude towards testing and quality assurance activities. Managers have a key role in fostering a generally favorable reception towards failure discovery during development and maintenance; for instance, by preventing a mindset of code ownership among programmers, so that they will not feel responsible for failures revealed by their code.

- ***Test guides*** — The testing phases could be guided by various aims, for example: in risk-based testing, which uses the product risks to prioritize and focus the test strategy; or in scenario-based testing, in which test cases are defined based on specified software scenarios.
- ***Test process management*** — Test activities conducted at different test levels must be organized, together with people, tools, policies, and measurements, into a well-defined process that is an integral part of the life cycle. In IEEE/EIA Standard 12207.0 [1996], testing is not described as a stand-alone process, but principles for testing activities are included along with both the five primary life-cycle processes and the supporting processes. In [IEEE Std 1074], testing is grouped with other evaluation activities as integral to the entire life cycle.
- ***Test documentation and work products*** — Documentation is an integral part of the formalization of the test process. The IEEE Standard for Software Test Documentation [IEEE 829-2007] provides a good description of test documents and of their relationship with one another and with the testing process. Test documents may include, among others, Test Plan, Test Design Specification, Test Procedure Specification, Test Case Specification, Test Log, and Test Incident or Problem Report. The software under test is documented as the Test Item. Test documentation should be produced and continually updated, to the same level of quality as other types of documentation in software engineering.
- ***Internal vs. independent test team*** — Formalization of the test process may involve formalizing the test team organization as well. The test team can be composed of internal members (that is, on the project team, involved or not in software construction), of external members, in the hope of bringing in an unbiased, independent perspective, or, finally, of both internal and external members. Considerations of costs, schedule, maturity levels of the involved organizations, and criticality of the application may determine the decision.
- ***Cost/effort estimation and other process measures*** — Several measures related to the resources spent on testing, as well as to the relative fault-finding effectiveness of the various test phases, are used by managers to control and improve the test process. These test measures may cover such aspects as number of test cases specified, number of test cases executed, number of test cases passed, and number of test cases failed, among others.

  Evaluation of test phase reports can be combined with root-cause analysis to evaluate test process effectiveness in finding faults as early as possible. Evaluation of test reports could provide feedback about typical errors found to improve coding or pre-test activities such as peer reviews (so the code is better when it enters testing).

  Such an evaluation could be associated with the analysis of risks. Moreover, the resources that are worth spending on testing should be commensurate with the use/criticality of the application: different techniques have different costs and yield different levels of confidence in product reliability.
- ***Termination*** — A decision must be made as to how much testing is enough and when a test stage can be terminated. Thoroughness measures, such as achieved code coverage or functional completeness, as well as estimates of fault density or of operational reliability,

provide useful support, but are not sufficient in themselves. The decision also involves considerations about the costs and risks incurred by the potential for remaining failures, as opposed to the costs implied by continuing to test.

- ***Test reuse and test patterns*** — To carry out testing or maintenance in an organized and cost-effective way, the means used to test each part of the software should be reused systematically. This repository of test materials must be under the control of software configuration management, so that changes to software requirements or design can be reflected in changes to the scope of the tests conducted.

  The test solutions adopted for testing some application types under certain circumstances, with the motivations behind the decisions taken, form a test pattern that can itself be documented for later reuse in similar projects.

4.6.2. Test activities. Under this topic, a brief overview of test activities is given. As often implied by the following description, successful management of test activities strongly depends on the software configuration management process [SWEBOK 2004].

- ***Planning*** — Like any other aspect of project management, testing activities must be planned. Key aspects of test planning include coordination of personnel, management of available test facilities and equipment (which may include magnetic media, test plans and procedures), and planning for possible undesirable outcomes. If more than one baseline of the software is being maintained, then a major planning consideration is the time and effort needed to ensure that the test environment is set to the proper configuration.
- ***Test-case generation*** — Generation of test cases is based on the level of testing to be performed and the particular testing techniques. Test cases should be under the control of software configuration management and include the expected results for each test.
- ***Test environment development*** — The environment used for testing should be compatible with the software engineering tools. It should facilitate development and control of test cases, as well as logging and recovery of expected results, scripts, and other testing materials.
- ***Execution*** — Execution of tests should embody a basic principle of scientific experimentation: everything done during testing should be performed and documented clearly enough that another person could replicate the results. Hence, testing should be performed in accordance with documented procedures using a clearly defined version of the software under test.
- ***Test results evaluation*** — The results of testing must be evaluated to determine whether or not the test has been successful. In most cases, "successful" means that the software performed as expected and did not have any major unexpected outcomes. Not all unexpected outcomes are necessarily faults, however, but could be judged to be simply noise. Before a failure can be removed, an analysis and debugging effort is needed to isolate, identify, and describe it. When test results are particularly important, a formal review board may be convened to evaluate them.
- ***Problem reporting/test log*** — Testing activities can be entered into a test log to identify when a test was conducted, who performed the test, what software configuration was the basis for testing, and other relevant identification information. Unexpected or incorrect

test results can be recorded in a problem-reporting system, the data of which forms the basis for later debugging and for fixing the problems that were observed as failures during testing. Also, anomalies not classified as faults could be documented in case they later turn out to be more serious than first thought. Test reports are also an input to the change management request process.

- ***Defect tracking*** — Failures observed during testing are most often due to faults or defects in the software. Such defects can be analyzed to determine when they were introduced into the software, what kind of error caused them to be created (poorly defined requirements, incorrect variable declaration, memory leak, programming syntax error, for example), and when they could have been first observed in the software. Defect-tracking information is used to determine what aspects of software engineering need improvement and how effective previous analyses and testing have been.

References

Additional information on the *software testing KA* can be found in the following documents:

- **[Bache & Müllerberg 1990]** R. Bache and M. Müllerberg, "Measures of Testability as a Basis for Quality Assurance," *Software Engineering Journal*, vol. 5, March 1990, pp. 86-92.
- **[IEEE 610.12-90]** IEEE Std 610.12-1990 (R2002), *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Inc., New York 1990.
- **[IEEE Std 1074]** IEEE Standard 1074-2006 *IEEE Standard for Developing a Software Project life-cycle Process*. IEEE Inc., New York 2006.
- **[IEEE 829-2007]** IEEE Standard 829-2007, *IEEE Standard for Software Test Documentation.* IEEE, Inc., New York, 2007.
- **[IEEE/EIA 12207.0-1996]** IEEE/EIA12207.0-1996, *Industry Implementation of International Standard ISO/IEC 12207:1995 Standard for Information Technology Software lifecycle Processes*. IEEE Inc., New York 1996.
- **[IEEE 1044-1993]** IEEE Standard 1044-1993, *IEEE Standards for the classification of Software Anomalies*. IEEE, Inc., New York. 1993.
- **[IEEE 982.1-1988]** *IEEE Standard Dictionary of Measurements to Produce Reliable Software*. IEEE, Inc., New York, 1988.
- **[Levi & Conrad 2008]** Michael D. Levi and Frederick G. Conrad, "Usability Testing of World Wide Web Sites." Office of Survey Methods Research, Bureau of Labor Statistics, United States Department of Labor, Washington D.C., 2008.
- **[Musa 1993]** John Musa, "Operational Profiles in Software Reliability Engineering," *IEEE Software Magazine,* March, 1993.
- **[Musa 2005]** John Musa, *Software Reliability Engineering: More Reliable Software Faster and Cheaper,* 2nd. Edition, Author House, Bloomington, IN, 2005
- **[SWEBOK 2004]** *Software Engineering Body of Knowledge (SWEBOK)*. IEEE Computer Society Press, Los Alamitos, CA, 2004.

- **[Usability Net 2006]** Usability Net was a project funded by the European Union to provide resources and networking for usability practitioners, managers and EU projects [http://www.usabilitynet.org].
- **[Voas & Miller 1995]** J.M. Voas and K.W. Miller, "Software Testability: The New Verification," *IEEE Software,* May, 1995, pp. 17-28.
- **[Wikipedia]** Wikipedia is a free web based encyclopedia enabling multiple users to freely add and edit online content. Definitions cited on Wikipedia and their related sources have been verified by the authors and other peer reviewers. Readers who would like to verify a source or a reference should search the subject on Wikipedia.

Chapter 5.1

Software Engineering Maintenance: An introduction[11]

Keith H. Bennett
School of Engineering & Computing Sciences
University of Durham
Durham City, England

The purpose of this tutorial is:

1. *To explain what is meant by software maintenance*
2. *To show how software maintenance fits into other software engineering activities*
3. *To explain the relationship between software maintenance and the organization*
4. *To explain best practice in software maintenance in terms of a process model*
5. *To describe important maintenance technologies such as impact analysis*
6. *To explain what is meant by a legacy system, and describe how reverse engineering and other techniques may be used to support legacy systems*

1. OVERVIEW OF THE TUTORIAL

The tutorial starts with a short introduction to the field of software engineering, thereby providing the context for the constituent field of software maintenance. The aim of the tutorial is to focus on solutions, not problems, but an appreciation of the problems in software maintenance is important. The solutions are categorized in a three-layer model: organizational issues, process issues, and technical issues.

Our presentation of organizational solutions to maintenance concentrates on software as an asset whose value needs to be sustained. We explain the process of software maintenance by describing the IEEE standard for the maintenance process. It provides a very sensible approach that is applicable to many organizations.

Technical issues are explained by concentrating on techniques of particular importance to maintenance. For example, *configuration management* and *version control* are as important for initial development as for maintenance, so these are not addressed. In contrast, coping with the ripple (domino) effect is only found during maintenance, and it is one of the crucial technical problems to be solved. We describe solutions to this.

By this stage, the tutorial will have presented the typical iterative maintenance process that is used, at various levels of sophistication, in many organizations. However, the software may become so difficult and expensive to maintain that special, often drastic, action is needed. The software is then called a "legacy system," and the particular problems of and solutions to coping with legacy code are described.

The tutorial is completed by considering some fruitful research directions for the field.

---

11. Based on "Software Maintenance: A Tutorial," by Keith H. Bennett, which appeared in R.H. Thayer and M. Dorfman (editors), *Software Engineering*, Volume I, 3rd edition, IEEE Computer Society Press, Los Alamitos, CA,

2. THE SOFTWARE ENGINEERING FIELD

Software maintenance is concerned with modifying software once it is delivered to a customer. By that definition, it forms a subarea of the wider field of software engineering, which is defined as:

> *The application of the systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software* [IEEE91].

It is helpful to understand trends and objectives of the wider field in order to explain the detailed problems and solutions concerned with maintenance. McDermid's definition in the *Software Engineering Reference Book* embodies the spirit of the engineering approach. He states that:

> *Software engineering is the science and art of specifying, designing, implementing and evolving—with economy, time limits and elegance—programs, documentation, and operating procedures whereby computers can be made useful to man* [MCDER91].

Software engineering is still a very young discipline and the term itself was only invented in 1967 [BAUE93]. Modern computing is only some 50+ years old, yet within that time we have gained the ability to solve very difficult and large problems. Often, these huge projects consume thousands of person-years or more of analysis and design. The rapid increase in the size of the systems that we tackle, from programs as small as 100 lines of code to multimillion-line systems now, presents very many problems of dealing with scale, so it is not surprising that evolving such systems to meet continually changing user needs is difficult.

Much progress has been made over the past decade in improving our ability to construct high-quality software that meets users' needs. Is it feasible to extrapolate these trends? Baber [BABE91] has identified three possible futures for software engineering:

1. Failures of software systems are common, due to limited technical competence of developers. This is largely an extrapolation of the present situation.
2. The use of computer systems is limited to that application in which there is a minimal risk to the public. There is widespread skepticism about the safety of software-based systems. There may be legislation covering the use of software in safety-critical and safety-related systems.
3. The professional competence and qualifications of software designers are developed to such a high level that even very challenging demands can be met reliably and safely. In this vision of the future, software systems would be delivered on time, fully meeting their requirements, and be applicable in safety-critical systems.

In case (1), software development is seen primarily as a craft activity. Option (2) is unrealistic; software is too important to be restricted in this way. Hence, there is considerable interest within the software engineering field in addressing the issues raised by (3). In this tutorial, we see (3), with obvious extensions to address evolving systems, as defining the goal of software maintenance.

A *root problem* for many software systems, which causes some of the most difficult problems for software maintenance, is complexity. Sometimes, this arises because a system is

migrated from hardware to software in order to gain the additional functionality that is easy to achieve in software. Complexity should be a result of implementing an inherently complex application (for example, in a tax calculation package, which is deterministic but nonlinear; or automation of the U.K. Immigration Act, which is complex and ambiguous). The main tools to control complexity are modular design and building systems as separated layers of abstraction in order to separate concerns. Nevertheless, the combination of scale and application complexity means that it is not feasible for one person alone to understand the complete software system.

3. SOFTWARE MAINTENANCE

Once software has been initially produced, it then passes into the maintenance phase. The IEEE definition of software maintenance is as follows:

> Software maintenance is the process of modifying the software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a change in environment [IEEE91].

Some organizations use the term software maintenance to refer only to the implementation of very small changes (e.g., less than one day), and software development is used to refer to all other modifications and enhancements. However, to avoid confusion, we shall continue to use the IEEE standard definition.

Software maintenance, although part of software engineering, is by itself of major economic importance. A number of surveys over the last 15 years have shown that for most software, software maintenance represents anything between 40% and 90% of total life-cycle costs (see [FOST93] for a review of such surveys). A number of surveys have also tried to compute the total software maintenance costs in the United Kingdom and in the United States. Although these figures need to be treated with a certain amount of caution, it seems clear that a huge amount of money is being spent on software maintenance.

The inability to undertake maintenance quickly, safely, and cheaply means that for many organizations, a substantial applications backlog builds up. The Management Information Services Department is unable to make changes at the rate required by marketing or business needs. End users become frustrated, and often adopt PC solutions in order to short circuit the problems. They may then find that a process of rapid prototyping and end-user computing provides them (at least in the short term) with quicker and easier solutions than those supplied by the Management Information Systems Department.

In the early decades of computing, software maintenance comprised a relatively small part of the software life cycle; the major activity was writing new programs for new applications. In the late 1960s and 1970s, management began to realize that old software does not simply die, and at that point software maintenance started to be recognized as a significant activity. An anecdote about the early days of electronic data processing in banks illustrates this point. In the 1950s, a large U.S. bank was about to take the major step of employing its very first full-time programmer. Management raised the issue of what would happen to this person once the programs had been written. The same bank now has several buildings full of data processing staff.

In the 1980s, it was becoming evident that old architectures were severely constraining new design. In another example from the U.S. banking system, existing banks had difficulty modifying their software in order to introduce automatic teller machines. In contrast, new banks writing software from scratch found this relatively straightforward. It has also been reported in the

United Kingdom that at least two mergers of financial organizations were unable to go ahead due to the problems of bringing together software from two different organizations.

In the 1990s, a large part of the business needs of many organizations was implemented, so that business change is now represented by evolutionary change to the software, not revolutionary change, and most so-called development is actually enhancement and evolution.

4. TYPES OF SOFTWARE MAINTENANCE

Leintz and Swanson [LEIN78, LEIN80] undertook a survey, as a result of which, maintenance was categorized into four different categories:

1. ***Perfective maintenance.*** Changes required as a result of user requests (also known as evolutive maintenance)
2. ***Adaptive maintenance.*** Changes needed as a consequence of operating system, hardware, DBMS, and so forth, changes
3. ***Corrective maintenance.*** The identification and removal of faults in the software
4. ***Preventative maintenance.*** Changes made to software to make it more maintainable

The above categorization is very useful to bring home to management some of the basic costs of maintenance. However, as will be seen in Section 9, the processes for the four types are very similar, and there is little advantage in distinguishing them when designing best practice maintenance processes.

It seems clear from a number of surveys that the majority of software maintenance is concerned with evolution deriving from user-requested changes.

The important requirement of software maintenance for the client is that changes are accomplished quickly and cost effectively. The reliability of the software should, at worst, not be degraded by the changes. Additionally, the maintainability of the system should not degrade otherwise future changes will be progressively more expensive to carry out. This phenomenon was recognized by Lehman, and expressed in terms of his well-known laws of evolution [LEHE80, LEHE84]. The first law of continuing change states that "a program that is used in a real-world environment necessarily must change or become progressively less useful in that environment."

This argues that software evolution is not an undesirable attribute; essentially, it is only useful software that evolves. Lehman's second law of increasing complexity states that "as an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving the semantics and simplifying the structure." This law argues that things will become much worse unless we do something about it. The problem for most software is that nothing has been done about it so that changes are increasingly more expensive and difficult. Ultimately, maintenance may become too expensive and almost infeasible; the software then becomes known as a "legacy system" (see Section 11). Nevertheless, it may be of essential importance to the organization.

5. PROBLEMS OF SOFTWARE MAINTENANCE

There are many technical and managerial problems in striving to accomplish the objective of changing software quickly, reliably, and cheaply. For example, user changes are often described in terms of the *behavior* of the software system; these must be interpreted as changes to

the source code. When a change is made to the code, there may be substantial consequential changes, not only in the code itself, but within documentation, design, test suites, and so on (this is termed the *domino* or *ripple effect).* Many systems under maintenance are very large, and solutions that work for laboratory-scale pilots will not scale up to industrial-sized software. Indeed, it may be said that any program that is sufficiently small to fit into a textbook or to be understood by one person does not have maintenance problems.

There is much in common between best practice in software engineering in general and software maintenance in particular. Software maintenance problems essentially break into three categories:

1. ***The alignment with organizational objectives.*** Initial software development is usually project based, with defined timescale and budget. The main emphasis is to deliver on time and within budget to meet user needs. In contrast, software maintenance often has the objective of extending the life of a software system for as long as possible. In addition, it may be driven by the need to meet user demand for software updates and enhancements. In both cases, return on investment is much less clear, so that the view at senior management level is often of a major activity that is consuming large resources and providing no clear quantifiable benefit for the organization.
2. ***Process issues.*** At the process level, there are many activities in common with software development. For example, configuration management is a crucial activity in both. However, software maintenance requires a number of additional activities not found in initial development. Initial requests for changes are usually made to a "help desk" (often part of a larger end-user support unit), which must assess the change (as many change requests derive from misunderstanding of documentation), and if it is viable, pass it to a technical group that can assess the cost of making the change. Impact analysis on both the software and the organization, and the associated need for system comprehension, are crucial issues. Further down the life cycle, it is important to be able to perform regression tests on the software so that the new changes do not introduce errors into the parts of the software that were not altered.
3. ***Technical issues***. There are a number of technical challenges to software maintenance. As noted above, the ability to construct software such that it is easy to comprehend is a major issue [ROBS91]. A number of studies have shown that the majority of time spent in maintaining software is actually consumed in this activity. Similarly, testing in a cost-effective way provides major challenges. Despite the emergence of methods based on discrete mathematics (e.g., to prove that an implementation meets its specification), most current software is tested rather than verified, and the cost of repeating a full test suite on a major piece of software can be very large in terms of money and time. It will be better to select a subset of tests that only stress those parts of the system that have been changed, together with the regression tests.

   The technology to do this is still not available, despite much useful progress. As an example, it is useful to consider a major billing package for an industrial organization. The change of the taxation rate in such a system should be a simple matter; after all, generations of students are taught to place such constants at the head of the program so only a one-line edit is needed. However, for a major multinational company, dealing with taxation rates in several countries with complex and different rules for tax calcula-

tions (i.e., complex business rules), the change of the taxation rate may involve a huge expense.

Other problems relate to the lower status of software maintenance compared with software development. In the manufacture of a consumer durable, the majority of the cost lies in production, and it is well understood that design faults can be hugely expensive. In contrast, the construction of software is automatic, and development represents almost all the initial cost. Hence, in conditions of financial stringency, it is tempting to cut costs by cutting back on design. This can have a very serious effect on the costs of subsequent maintenance.

One of the problems for management is that it is very difficult to assess a software product to determine how easy it is to change. This means that there is little incentive for initial development projects to construct software that is easy to evolve. Indeed, lucrative maintenance contracts may follow a software system in which shortcuts have been taken during development [WALT94].

We have stressed the problems of software maintenance in order to differentiate it from software engineering in general. However, much is known about best practice in software maintenance and there are excellent case studies such as the U.S. Space Shuttle on-board flight control software system, which demonstrates that software can be evolved carefully and with improving reliability. The remainder of this paper is focused on solutions rather than problems. The great majority of software in use today is neither geriatric nor state of the art, and this tutorial addresses this type of software. It describes a top-down approach to successful maintenance, addressing:

1. Software maintenance and the organization
2. Process models
3. Technical Issues

In particular, we shall focus on the IEEE standard for software maintenance process, which illustrates the improving maturity of the field.

6. ORGANIZATIONAL ASPECTS OF MAINTENANCE

In 1987, Colter [COLT87] stated that the major problem of software maintenance as not technical, but managerial: software maintenance organizations were failing to relate their work to the needs of the business, and, therefore, it should not be a surprise that the field suffered from low investment and poor status in comparison to initial development, which was seen as a revenue and profit generator

Initial software development is product oriented; the aim is to deliver an artifact within budget and on time. In contrast, software maintenance is much closer to a service. In many Japanese organizations, for example [BENN94], software maintenance is seen at the senior management level primarily as a means of ensuring continued satisfaction with the software; it is closely related to quality. The customer expects the software to continue to evolve to meet his or her changing needs, and the vendor must respond quickly and effectively or lose business. In Japan, it is also possible in certain circumstances to include software as an asset on the balance sheet. These combine to ensure that software maintenance has a high profile with senior management in Japan.

Like any other activity, software maintenance requires financial investment. We have already seen that maintenance may be regarded by senior management in a company simply as a drain on resources, distant from core activities, and it becomes a prime candidate for funding reduction and even for closing down. Software maintenance thus needs to be expressed in terms of return on investment. In many organizations undertaking maintenance for other internal divisions, the service is rarely charged out as a revenue-generating activity from a profit center. In the U.K. defense sector, there has been a major change in practice in charging for maintenance. Until recently, work would be charged to Government based on the time taken to do the work plus a profit margin. Currently, competitive tendering (procurement) is used for specific work packages.

Recently there has been a trend for software maintenance to be *outsourced;* in other words, a company will contract out its software maintenance to another that specializes in this field. Companies in India, China, and other countries are becoming increasingly competitive in this market. This is sometimes done for peripheral software, as the company is unwilling to release the software used in its core business. An outsourcing company will typically spend a number of months assessing the software before it will accept a contract. Increasingly, *service-level agreements* between the maintenance organization (whether internal or external) and the customer are being used as a contractual mechanism for defining the maintenance service that will be provided. The U.K. Central Computer and Telecommunications Agency has produced a series of guidelines on good practice in this area, in the form of the Information Technology Infrastructure Library [ITIL93].

When new software is passed over to the customer, payment for subsequent maintenance must be determined. At this stage, primary concerns are typically:

- Repair of errors on delivery
- Changes to reflect an ambiguous specification

Increasingly, the former is being met by some form of warranty, to bring software in line with other goods (although much commodity software is still ringed with disclaimers). Hence, the vendor pays. The latter is much more difficult to resolve, and addresses much more than the functional specification. For example, if the software is not delivered in a highly maintainable form, there will be major cost implications for the purchaser.

Recently, Foster [FOST93] proposed an interesting investment cost model that regards software as a corporate asset that can justify financial support in order to sustain its value. Foster uses his model to determine the optimum release strategy for a major software system. This is, hence, a business model, allowing an organization the ability to calculate return on investment in software by methods comparable with investment in other kinds of assets. Foster remarks that many papers on software maintenance recognize that it is a little understood area but it consumes vast amounts of money. With such large expenditure, even small technical advances must be worth many times their cost. The software maintenance manager, however, has to justify investment in an area that does not directly generate income. Foster's approach allows a manager to derive a model for assessing the financial implications of the proposed change of activity, thereby providing the means to calculate both cost and benefit. By expressing the result in terms of return on investment, the change can be ranked against competing demands for funding.

Some work has been undertaken in applying predictive cost modeling to software maintenance, based on the COCOMO techniques. The results of such work remain to be seen.

The AMES project [HATH94, BOLD94, and BOLD95] is addressing the development of methods and tools to aid application management, where application management is defined as "the contracted responsibility for the management and execution of all activities related to the maintenance of existing applications." Its focus is on the formalization of many of the issues raised in this section, and, in particular, customer-supplier relations. It is developing a maturity model to support the assessment of this relationship in a quantitative and systematic way.

7. PROCESS MODELS

Process management is defined as "the direction, control, and co-ordination of work performed to develop a product or perform a service" [IEEE91]. This definition, therefore, encompasses software maintenance and includes quality, line management, technical, and executive processes. A mature engineering discipline is characterized by mature, well-understood processes, so it is understandable that modeling software maintenance, and integrating it with software development, is an area of active concern [MCDER91]. A software process model may be defined as "a purely descriptive representation of the software process, representing the attributes of a range of particular software processes and being sufficiently specific to allow reasoning about them" [DOWS85].

The foundation of good practice is a mature process, and the Software Engineering Institute at Carnegie-Mellon University has pioneered the development of a scale by which process maturity may be measured. More recently, the BOOTSTRAP project has provided an alternative maturity model from a European perspective.

In order to promote the establishment of better understood processes, the IEEE has published a standard for software maintenance [IEEE98] and the next section describes this in detail. This reflects the difference between maintenance and initial development processes. It represents well many of the elements of good practice in software maintenance. The model is based on an iterative approach of accepting a stream of change requests (and error reports), implementing the changes, and, after testing, forming new software releases. This model is widely used in industry, in small-to-medium-sized projects, and for in-house support. It comprises four keys stages:

1. **Help desk.** The problem is received, a preliminary analysis undertaken, and, if the problem is sensible, it is accepted.
2. **Analysis.** A managerial and technical analysis of the problem is undertaken, to investigate and cost alternative solutions.
3. **Implementation.** The chosen solution is implemented and tested.
4. **Release.** The change (along with others) is released to the customer.

Most best-practice models (e.g., that of Hinley [HINL92]) incorporate this approach, though it is often refined into much more detailed stages (as in the IEEE model described in the next section). Wider aspects of the software maintenance process, in the form of applications management, are addressed in [HATH94].

8. IEEE STANDARD FOR SOFTWARE MAINTENANCE [IEEE98]

8.1. Overview of the Standard

This standard describes the process for managing and executing software maintenance activities. Almost the entire standard is relevant for software maintenance. The focus of the standard is in a seven-stage activity model of software maintenance, which incorporates the following stages:

1. Problem identification
2. Analysis
3. Design
4. Implementation
5. System test
6. Acceptance test
7. Delivery

Each of the seven activities has five associated attributes; these are:

1. Input life-cycle products
2. Output life-cycle products
3. Activity definition
4. Control
5. Metrics

A number of these, particularly in the early stages of the maintenance process, were addressed by existing IEEE standards.

As an example, we consider the second activity in the process model, the analysis phase. This phase accepts as its input a validated problem report, together with any initial resource estimates and other repository information, plus project and system documentation if available. The process is seen as having two substantial components. First of all, feasibility analysis is undertaken, in which the impact of the modification is assessed, alternative solutions investigated, short- and long-term costs assessed, and the value of the benefit of making the change computed. Once a particular approach has been selected, then the second stage of detailed analysis is undertaken. This determines firm requirements of the modification, identifies the software involved, and requires a test strategy and an implementation plan to be produced.

In practice, this is one of the most difficult stages of software maintenance. The change may affect many aspects of the software, including not only documentation, test suites, and so on, but also the environment and even the hardware. The standard insists that all affected components shall be identified and brought into the scope of the change.

The standard also requires that at this stage a test strategy be derived comprising at least three levels of test including unit testing, integration testing, and user-orientated functional acceptance tests. It is also required to supply regression test requirements associated with each of these levels of test.

8.2. Structure of the Standard

The standard also establishes quality control for each of the seven phases. For example, for the analysis phase, the following controls are required as a minimum:

1. Retrieval of the current version of project and systems documentation from the configuration control functions of the organization.
2. A review of the proposed changes and an engineering analysis to assess the technical and economic feasibility and to assess correctness.
3. Consideration of the integration of the proposed change within the existing software.
4. *Verification* that all appropriate analysis and project documentation is updated and properly controlled.
5. *Verification* that the testing organization is providing a strategy for testing the changes and that the change schedule can support the proposed test strategy.
6. *Review* of the resource estimates and schedules and verification of their accuracy.
7. The undertaking of a technical review to select the problem reports and proposed enhancements to be implemented and released. The list of changes shall be documented.

Finally, at the end of the analysis phase a risk analysis is required to be performed. Any initial resource estimate will be revised, and a decision that includes the customer is made on whether to proceed on to the next phase.

The phase deliverables are also specified, again as a minimum, as follows:

1. Feasibility report for problem reports
2. Detailed analysis report
3. Updated requirements
4. Preliminary modification list
5. Development, integration, and acceptance test strategy
6. Implementation plan

The contents of the analysis report are further specified in greater detail by the proposed standard. The standard suggests that the following metrics be taken during the analysis phase:

1. Requirement changes
2. Documentation error rates
3. Effort per function area
4. Elapsed time
5. Error rates generated, by priority and type

The standard also includes appendices that provide guidelines on maintenance practice. These are not part of the standard itself but are included as useful information. For example, in terms of our analysis stage, the appendix provides a short commentary on the provision of

change on impact analysis. A further appendix addresses supporting maintenance technology, particularly reengineering and reverse engineering. A brief description of these processes is also given.

8.3. Assessment of the IEEE Standard

The standard represents a welcome step forward in establishing a process standard for software maintenance. Strength of this approach is that it is based on existing IEEE standards from other areas in software engineering. It accommodates practical necessities, such as the need to undertake emergency repairs.

On the other hand, it is clearly oriented toward classic concepts of software development and maintenance. It does not cover issues such as rapid application development, Agile methods, and end-user computing. Nor does it address executive-level issues in the process model nor establish boundaries for the scope of the model.

The process model corresponds approximately to level two in the SEI five-level model. The SEI model forms the basis of the SPICE process assessment standards initiative.

Organizations may well be interested in increasing the maturity of their software engineering processes. Neither the IEEE standard nor the SEI model give direct help in process improvement. Further details of this may be found in [HINL92]. Additionally, there is still little evidence in practice that improving software process maturity actually benefits organizations, and the whole edifice is based on the assumption that the quality of the product, not its successes, depends on the process by which it is developed.

It is useful to note that the International Standards Organization [ISO01] has published a standard for a process model to assess the quality (including maintainability) of software. However, many technical problems in measurement remain unsolved.

9. TECHNICAL ASPECTS OF SOFTWARE MAINTENANCE

9.1. Technical Issues

Much of the technology required for software maintenance is similar to that needed for initial development, with minor changes. For example, configuration management and version control are indispensable for both. Information relating to development and maintenance will be kept in a repository. For maintenance, the repository will be used to hold frequently occurring queries handled by the help desk. Metrics data for product and process will be similar. CASE tools, supporting graphical representation of software, are widely used in development and maintenance. These topics are described in other chapters of this book; here we concentrate on issues of specific importance to maintenance.

In our description of the IEEE standard process model, the need for impact analysis was identified. This is a characteristic of software maintenance that is not needed in initial software development. We shall present further details of this technique as an example of the technology needed to support software maintenance.

In the above process model, it was necessary to determine the cost of making a change to meet a software change request. In this section, we therefore examine how impact analysis can help this activity. To amplify the analysis needed, the user-expressed problem must first of all be translated into software terms to allow the maintenance team to decide if the problem is viable for further work or if it should be rejected. It then must be localized; this step determines

the origin of the anomaly by identifying the primary components to the system that must be altered to meet the new requirement.

Next, the above step may suggest several solutions, all of which are viable. Each of these must be investigated, primarily by using impact analysis. The aim is to determine all changes that are consequent to the primary change. It must be applied to all software components, not just to code. At the end of impact analysis, we are in the position to make a decision on the best implementation route or to make no change. Weiss [WEIS 89] has shown, for three NASA projects, the primary source of maintenance changes deriving from user problem reports:

- Requirements phase 19%
- Design phase 52%
- Coding phase 7%

He noted that 34% of changes affected only one component and 26% affected two components.

9.2. The Problem

One of the major difficulties of software maintenance that encourages maintainers to be very cautious by nature is that a change made at one place in the system may have a ripple effect elsewhere, so consequent changes must be made. In order to carry out a consistent change, all such ripple effects must be investigated; the impact of the change must be assessed and changes possibly made in all affected contexts. You define this as:

> *Ripple effect propagation is a phenomenon by which changes made to a software component along the software life cycle (specification, design, code, or test phase) have a tendency to be felt in other components* [YAU87].

As a very simple example, a maintainer may wish to remove a redundant variable *X*. It is obviously necessary to remove all applied occurrences of *X* too, but for most high-level languages the compiler can detect and report undeclared variables. This is, hence, a very simple example of an impact that can be determined by *static analysis*. In many cases, ripple effects cannot be determined statically, and dynamic analysis must be used. For example, an assignment to an element of an array, followed by the use of a subscripted variable, may or may not represent a ripple effect depending on the particular elements accessed. In really large programs containing pointers, aliases, and so on, the problem is much harder. We shall define the problem of impact analysis as the task for assessing the effects for making the set of changes to a software system [WILD93].

The starting point for impact analysis is an explicit set of primary software objects that the maintainer intends to modify. He or she has determined the set by relating the change request to objects such as variables, assignments, and goals. The purpose of impact analysis is, hence, to ensure that the change has been correctly and consistently bounded. The impact analysis stage identifies a set of further objects impacted by changes in the primary sector. This process is repeated until no further objects can be identified.

9.3. Traceability

In general, we require traceability of information between various software artifacts in order to help us assess impact in software components. Traceability is defined as:

Traceability is the degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another [IEEE91].

Informally, traceability provides us with semantic links that we can then use to perform impact analysis. The links may relate similar components such as design documents or they may link between different types, for example, from specification to code.

Some types of traceability links are very hard to determine. For example, altering the source code in even a minor way may have performance implications that cause a real-time system to fail to meet a specification. It is not surprising that the majority of work in impact analysis has been undertaken at the code level as this is the most tractable. Wilde [WILD89] provides a good review of code-level impact analysis techniques.

Many modern programming languages are based on using static analysis to detect or stop the ripple effect. The use of modules with opaque types, for example, can prevent at compile time several unpleasant types of ripple effect. Many existing software systems are unfortunately written in older languages, using programming styles (such as global aliased variables) that make the potential for ripple effects much greater and their detection much harder.

More recently, Turver and Munro [TURV94] have described an approach that has placed impact analysis within the overall software maintenance process. The major advance is that documentation is included within the objects analyzed; documentation is modeled using a ripple propagation graph and it is this representation that is used for analysis. The approach has the advantage that it may be set in the early stages of analysis to assess costs without reference to the source code.

Work has also been undertaken recently to establish traceability links between HOOD design documents [FILL94] in order to support impact analysis of the design level.

In a major research project at Durham, formal semantic-preserving transformations are being used to derive executable code from formal specifications, and in reverse engineering to derive specifications from existing code. The ultimate objective is to undertake maintenance at the specification level rather than the code level, and generate executable code automatically or semi automatically. The transformation technique supports the derivation of the formal traceability link between the two representations, and research is underway to explore this as a means of enhancing the ripple effect across wider sections of the life cycle (see, e.g., WARD93, WARD94, WARD94a, YOUN94, BENN95, and BENN95b for more details).

10. LEGACY SYSTEMS

10.1. Legacy Problems

There is no standard definition of a legacy system, but many in industry will recognize the problem. A legacy system is typically very old and has been heavily modified over the years to meet continually evolving needs. It is very large, so that a team is needed to support it; none of the team were involved when the software was first developed. It will be based on old technology and be written in out-of-date languages such as Assembler. Documentation will not be available. Testing new releases is a major difficulty. Often, the system is supporting very large quantities of live data.

Such systems are surely a candidate for immediate replacement. The problem is that the software is often at the core of the business; replacing it would be a huge expense, and while less than ideal, the software works and continues to do useful things.

An example of a legacy system is the billing software for a telecommunications company. The software was developed 30 years ago, when the company was owned by the government, and the basic service sold was restricted to a telephone connection to each premise. The system is the main mechanism for generating revenue; it supports a huge on-line database of paying customers.

Over the years, the software has been maintained to reflect the changing telecommunications business: from government to private ownership; from simple call charging to wide-ranging and complex (and competitive) services; from single country to international organization, with highly complex VAT (value added tax) systems. The system now comprises several million lines of source code.

Although the process of maintenance to meet continually evolving customer needs is becoming better understood, and more closely linked with software engineering in general, dealing with legacy software is still very hard. It has been estimated that there are 70 billion lines of COBOL in existence, and still doing useful work. Much of the useful software being written today will end up as legacy software in 20 years' time. Software that is 40 years old is being used in mission-critical applications.

It is easy to argue that the industry should never have ended up in the position of relying on such software. It is not clear that steps are being taken to avoid the problem for modern software. There seems to be a hope that technology such as object-oriented design will solve the problems for future generations, though there is as yet little positive evidence for this.

In this section, we shall analyze why it might be useful not just to discard the legacy system and start over again. In the subsequent section, we shall present solutions to dealing with legacy systems.

10.2. Analysis of Legacy Systems

In some cases, discarding the software and starting again may be the courageous, if expensive, solution, following analysis of the business need and direction, and the state of the software. Often, the starting point has to be taking an inventory of the software, as this may not be known. As a result of analysis, the following solutions for the legacy system may be considered:

- Carry on as now, possibly subcontracting the maintenance
- Replace software with a package
- Reimplement from scratch
- Discard software and discontinue
- Freeze maintenance and phase in a new system
- Encapsulate the old system and use as a server for the new system
- Reverse engineer the legacy system and develop a new software suite.

In the literature, case studies addressing these types of approaches are becoming available. The interest of this tutorial is focused on reverse engineering, as it appears to be the most fruitful

approach. Increasing interest is being shown in encapsulation as a way of drawing a boundary round the legacy system. The new system is then evolved so that it progressively takes over functionality from the old one, until the latter becomes redundant. Currently, few successful studies have been published, but these support the move to distributed open systems based on client-server architectures.

10.3. Reverse Engineering

Chikofsky and Cross have defined several terms in these fields that are now generally accepted. Reverse engineering is:

> *. . . The process of analyzing a subject system to identify the system's components and their inter-relationships, and to create representations of the system in another form or at higher levels of abstraction [CHIK90].*

It can be seen that reverse engineering is passive; it does not change the system or result in a new one, though it may add new representations to it. For example, a simple reverse engineering tool may produce call graphs and control flow graphs from source code. These are both higher-level abstractions, though in neither case is the original source code changed. Two important types of reverse engineering are redocumentation, which is the creation or revision of a semantically equivalent representation within the same relative abstraction layer, and design recovery, which involves identifying meaningful higher-level abstractions beyond those obtained directly by examining the system itself.

The main motivation is to provide help in program comprehension; most maintainers have little choice but to work with source codes, in the absence of any documentation. Concepts such as procedure structures and control flow are important mechanisms by which the maintainer understands the system, so tools have been constructed to provide representations to help the process.

If good documentation existed (including architectural, design, test suite documentation, etc.), reverse engineering would be unnecessary. However, the types of documentation needed for maintenance are probably different from those produced during typical initial development. As an example, most large systems are too big for one person to maintain, yet the maintainer rarely needs to see a functional decomposition or object structure; he or she is trying to correlate external behavior with internal descriptions. In these circumstances, slicing offers help. Slicing is a static analysis technique, in which only those source code statements that can affect a nominated variable are displayed.

Pragmatically, many maintainers cover source-code listings with notes and stick-on pieces of paper. In an attempt to simulate this, Foster and Munro [FOST87] and Younger [YOUN93] have built tools to implement a hypertext form of documentation that is managed incrementally by the maintainer, who is able to attach "notes" to the source code. An advantage of this approach is that it does not attempt to redocument the whole system; documentation is provided, by the maintainer, in the form preferred, only for the "hot spots." Those parts of the code that are stable, and are never studied by the maintainer (often large parts), do not have to be redocumented, thereby saving money.

For a description of a reverse engineering method, see [EDWA95].

10.4. Program Comprehension

Program comprehension is a topic in its own right, and has stimulated an annual IEEE workshop. Documentation is also an active area; see, for example, Knuth's WEB [KNUT84] and also Gilmore [GILM90] for details of issues concerned with psychology. In [YOUN93], there is a useful list of criteria for software maintenance documentation:

- Integrated source code, via traceability links
- Integrated call graphs, control graphs, and so on
- Integration of existing documentation (if any)
- Incremental documentation
- Informal update by maintainer
- Quality assurance on the documentation
- Configuration management and version control of all representations
- Information hiding to allow abstraction
- Team use

It may be decided that active change of the legacy system is needed. Restructuring is the transformation from one representation to another at the same relative level of abstraction, while preserving the system's external behavior.

Lehman's second law argues that such remedial action is essential in a system that is undergoing maintenance. Otherwise, the maintainability will degrade and the cost of maintenance correspondingly increases. Examples include:

- Control flow restructuring to remove "spaghetti" code
- Converting monolithic code to use parameterized procedures
- Identifying modules and abstract data types
- Removing dead code and redundant variables
- Simplifying aliased/common and global variables

Finally, reengineering is the examination and alteration of the subject system to reconstitute it in a new form, and the subsequent implementation of the new form.

Reengineering is the most radical (and expensive) form. It is not likely to be motivated simply by wanting more maintainable software. For example, owners of on-line systems produced in the 1960s and 1970s would like to replace the existing character-based input/output with a modern graphical user interface. This is usually very difficult to achieve easily, so it may be necessary to undertake substantial redesign.

10.5. Reverse Engineering and Reengineering

In [BENN93], a list of 26 decision criteria for considering reverse engineering is presented. In abbreviated form, these are:

- Management criteria

- Enforcing product and process standards (such as the IEEE draft standard introduced above)
- Permit better maintenance management
- Legal contesting of reverse engineering legislation
- Better audit trails

- Quality criteria
  - Simplification of complex software
  - Facilitating detection of errors
  - Removing side effects
  - Improving code quality
  - Undertaking major design repair correction
  - Production of up-to-date documentation
  - Preparing full test suites
  - Improving performance
  - Aligning with practices elsewhere in the company
  - Financial auditing
  - Facilitating quality audits (e.g., IS0 9000)
- Technical criteria
  - To allow major changes to be made
  - To discover the underlying business model
  - To discover the design and requirements specification
  - To port the system
  - To establish a reuse library
  - To introduce technical innovation such as fault tolerance, graphic interfaces, etc.
  - To reflect evolving maintenance processes
  - To record many different types of high-level representations
  - To update tool support
  - To facilitate disaster recovery

It is useful to amplify two of the above points. First, many legacy systems represent years of accumulated experience, and this experience may now no longer be represented anywhere else. Systems analysis cannot start with humans and hope to introduce automation; the initial point is the software that contains the business rules.

Second, it is not obvious that a legacy system, which has been modified over many years, does actually have a high-level, coherent representation. Is it simply the original system plus the

aggregation of many accumulated changes? The evidence is not so pessimistic. The current system reflects a model of current reality, and it is that model we are trying to discover.

10.6. Techniques

Work on the simplification of control-flow and data-flow graphs has been undertaken for many years. A very early result showed that any control graph (using, e.g., unstructured GOTOs) can be restructured into a semantically equivalent form using sequences, "if-then-else" conditionals, and loops, although this may cause flag variables to be introduced. A good review of this type of approach can be found in the *Redo Compendium* [ZUYL93]. This work is generally mature, and commercial tools exist for extracting, displaying, and manipulating graphical representations of source code. In [WARD93], an approach using formal transformations is described that is intended to support the human maintainer, rather than act as a fully automated tool. This work shows that much better simplification is achievable, such as the conversion of monolithic code with aliased variables to well-structured code using parameterized procedures.

Much research in reverse engineering, especially in the United States, has been based on the program plan or cliché approach, pioneered by Rich and Waters [RICH90]. This is based on the recognition that many programs use a relatively small number of generic design ideas, which tend to be used over and over again. Reverse engineering should then attempt to find such plans in existing source code, by matching from a set of patterns in a library. This would appear to have had some modest success, but there are many open issues. For example, how should patterns be represented? How generic are they? And how good is the matching process? This approach shares many of the problems of libraries of reusable components.

Most researchers aim to make their approach source language independent, so that different languages may be handled by adding front ends. Thus, design of intermediate languages is an important issue. In [ZUYL93], an approach called UNIFORM is described.

Ward [WARD93] uses a formally defined, wide-spectrum language, WSL, as the heart of his system. A wide-spectrum language is used as the representational format because only one language is then needed, for both low and high levels of abstractions and intermediary points. The approach has been shown to work for large (80K line) assembler programs, and also for very challenging benchmark cases such as the Schorr-Waite graph-marking algorithm. Further details are given in [BULL92] and [BULL94] ([BULL94] also contains a useful review of other transformation systems.)

Cimitile and his colleagues have undertaken much research on producing tools and methods for discovering abstract data types in existing code [CANF94]. Sneed [SNEE91, also NYAR95, and SNEE93] has presented his experience in reverse engineering of large commercial COBOL systems using partial tool support.

It is encouraging to observe that most new, promising approaches to reverse engineering address two basic properties of legacy systems:

- They are very large, and "toy" solutions are not applicable.
- They must be taken as they are and not how the engineer would like them to be. Often this means "one-off" solutions.

11. RESEARCH QUESTIONS

Although software maintenance tends to be regarded in academic circles as being of minor importance, it is of major commercial and industrial significance. It is useful to end this tutorial with a brief review of promising trends.

There are many interesting research problems to be solved that can lead to important commercial benefits. There are also some grand challenges that lie at the heart of software engineering.

How do we change software quickly, reliably, and safely? In safety-critical systems, for example, enormous effort is expended in producing and validating software. If we wish to make a minor change to the software, do we have to completely repeat the validation or can we make the cost of the change proportional in some way to its size? There are several well-publicized cases in which very minor changes to important software have caused major crashes and failures in service. A connected problem lies in the measurement of how easily new software can be changed. Without this, it is difficult to purchase software in the knowledge that a reduced purchase price is not to be balanced by enormous maintenance costs later on. Almost certainly, the solution to this problem will involve addressing process issues as well as attributes of the product itself. This is a major problem for computer science. A new approach is described in [SMIT95].

In practice, much existing software has been evolved in ad-hoc ways, and has suffered the fate predicted by Lehman's laws. Despite their often central role in many organizations, such legacy systems provide a major headache. Management and technical solutions are needed to address the problems of legacy systems; otherwise, we shall be unable to move forward and introduce new technology because of our commitments and dependence on the old.

It is often thought that the move to end-user computing, open systems and client-server systems has removed this problem. In practice, it may well make it considerably worse. A system that is comprised of many components, from many different sources, by horizontal and vertical integration, and possibly across a widely distributed network, poses major problems when any of those components change. For further details of this issue, see [BENN94b].

12. PROFESSIONAL SUPPORT

Over the last 20 years, professional activity in software maintenance has increased considerably. The annual International Conference on Software Maintenance, sponsored by the IEEE, represents the major venue that brings academics and practitioners together to discuss and present the latest results and experiences. Also relevant is the IEEE workshop on program comprehension. The proceedings of both conferences are published by the IEEE.

In Europe, the main annual event is the annual European Workshop on Software Maintenance, organized in Durham, England. This is mainly aimed at practitioners, and, again, the proceedings are published.

There is a journal—the *Journal of Software Maintenance*: *Research and Practice*—which appears bimonthly and acts as a journal of record for significant research and practice advances in the field.

Finally, aspects of software maintenance are increasingly being taught in university courses, and Ph.D. graduates are starting to appear who have undertaken research in the field.

13. CONCLUSIONS

We have described a three-level approach to considering software maintenance in terms of the impact on the organization, on the process, and on technology supporting that process. This approach has provided a framework with which to consider maintenance. Much progress has been made in all three areas and we have briefly described recent work on the establishment of a standard maintenance process model. The adoption of such models, along with formal process assessment and improvement, will do much to improve the best practice and average practice in the field of software maintenance.

We have also described a major problem that distinguishes software maintenance: coping with legacy systems. We have presented several practical techniques for addressing such systems.

Thus, we have presented software maintenance not as a problem but as a solution. However, there are still major research issues of strategic industrial importance to be solved. We have defined these as, first, to learn how to evolve software quickly and cheaply, and, second, how to deal with large legacy systems. Whereas modern technologies such as object-oriented systems claim to improve the situation, this is largely a hope, and there is yet little evidence that these technologies do indeed do so. Such technology may introduce new maintenance problems; see, for example, [SMIT92, TURN93, and TURN95] for testing methods associated with object-oriented programs. As usual, there are no magic bullets, and the Japanese principle of *Kaizen,* the progressive and incremental improvement of practices, is likely to be more successful.

ACKNOWLEDGEMENTS

Much of the work at Durham in software maintenance has been supported by SERC (now EPSRC) and DTI funding, together with major grants from IBM and British Telecom. I am grateful to colleagues at Durham for discussions that led to ideas presented in this paper, in particular to Martin Ward and Malcolm Munro. A number of key ideas have arisen from discussions with Pierrick Fillon. Thanks are due to Cornelia Boldyreff for reading drafts of this paper.

REFERENCES

The references in this chapter are:

- **[BABE91]** Baber, R. L., "Epilogue: Future Developments," in *Software Engineer's Reference Book,* Ed. McDermid, Butterworth-Heinemann, 1991.
- **[BAUE93]** Introduction to a 1993 IEEE tutorial, R.H. Thayer, and A.D. McGettrick (eds.), *Software Engineering: A European Perspective*, IEEE Computer Society Press, Los Alamitos, CA, 1993.
- **[BENN93]** Bennett, K. H., "An Overview of Maintenance and Reverse Engineering," in *The REDO Compendium,* Ed. van Zuylen, Wiley, 1993.
- **[BENN94]** Bennett, K. H., "Software Maintenance in Japan." Report published under the auspices of the U.K. Department of Trade and Industry, September, 1994. Available from the Computer Science Department, University of Durham, South Road, Durham UK.
- **[BENN94b]** Bennett, K. H., "Theory and Practice of Middle-out Programming to Support Program Understanding," in *Proceedings of IEEE Conference on Program Comprehension,* Washington, 1994, pp. 168-175.

- **[BENN95]** Bennett, K. H., and Ward, M. P., "Formal Methods for Legacy Systems," *Journal of Software Maintenance: Research and Practice,* 7(3): 203-219, May–June 1995.
- **[BENN95b]** Bennett, K. H., and Yang, H., "Acquisition of ERA Models from Data Intensive Code," in *Proceedings of IEEE International Conference on Software Maintenance,* Nice, France, October 1995, pp. 116-123.
- **[BOLD94]** Boldyreff, C., Burd, E., and Hather, R., "An Evaluation of the State of the Art for Application Management," in *Proceedings of the International Conference on Software Maintenance,* IEEE Computer Society Press, 1994, pp. 161-169.
- **[BOLD95]** Boldyreff, C., Burd, E., Hather, R. M., Mortimer, R. E., Munro, M., and Younger, E. J., "The AMES Approach to Application Understanding: A Case Study," in *Proceedings of the International Conference on Software Maintenance,* IEEE Computer Society Press, 1995, pp. 182-191.
- **[BULL94]** Bull, T., *Software Maintenance by Program Transformation in a Wide Spectrum Language*, Ph.D. Thesis, Department of Computer Science, University of Durham, 1994.
- **[BULL92]** Bull, T. M., Bennett, K. H., and Yang, H., "A Transformation System for Maintenance—Turning Theory into Practice," in *Proceedings of IEEE Conference on Software Maintenance,* Orlando, Florida, USA, 1992.
- **[CANF94]** Canfora, G., Cimitile, A., and Munro, M., "RE2: Reverse Engineering and Reuse Re-Engineering," *Journal of Software Maintenance: Research and Practice,* 6(2): 53-72, March–April 1994.
- **[CHIK90]** Chikofsky, E. J., and Cross, J. H., "Reverse engineering and design recovery: A taxonomy," *IEEE Software*, 7(1):13-17, January 1990.
- **[COLT87]** Colter, M., "The Business of Software Maintenance," in *Proceedings of First Workshop on Software Maintenance,* University of Durham, Durham, 1987. Available from the Computer Science Department, University of Durham (see [BENN94]).
- **[DOWS85]** Dowson, M., and Wilden, J. C., "A Brief Report on the International Workshop on the Software Process and Software Environment," *ACM Software Engineering Notes,* 10: 19-23, 1985.
- **[EDWA95]** Edwards, H. M., Munro, M., and West, R., *The RECAST Method for Reverse Engineering,* Information Systems Engineering Library, CCTA, HMSO, ISBN: 1 85 554705 8, 1995.
- **[FILL94]** Fillon, P., *An Approach to Impact Analysis in Software Maintenance,* M.Sc. Thesis, University of Durham, 1994.
- **[FOST87]** Foster, J., and Munro, M., "A Documentation Method Based on Cross-Referencing," in *Proceedings of IEEE Conference on Software Maintenance,* Austin, Texas, 1990.
- **[FOST93]** Foster, J., *Cost Factors in Software Maintenance,* Ph.D. Thesis, Computer Science Department, University of Durham, 1993.

- **[GLLM90]** Gilmore, D., "Expert Programming Knowledge: A Strategic Approach," in *Psychology of Programming,* Ed. Hoc, J. M., Green, T. R. G., Samurcay, R., and Gilmore, D. J., Academic Press, 1990.
- **[HATH94]** Hather, R., Burd, L., and Boldyreff, C., "A Method for Application Management Maturity Assessment," in *Proceedings of Centre for Software Reliability Conference,* Dublin, 1994.
- **[HINL92]** Hinley, D. S., and Bennett, K. H., "Developing a Model to Manage the Software Maintenance Process," in *Proceedings of Conference on Software Maintenance,* Orlando, Florida, IEEE Computer Society Press, 1992.
- **[IEEE91]** IEEE Std. 610.12-1990, *IEEE Standard Glossary of Software Engineering Terminology,* IEEE, 1991.
- **[IEEE98]** *IEEE Std. 1219-1998, Standard for Software Maintenance.* IEEE, 1998.
- **[IS001]** *International Standards Organisation Information Technology—Software Product Evaluation—Quality Characteristics and Guidelines for Their Use.* ISO/IEC Standard 9126, 2001.
- **[ITIL93]** *The IT Infrastructure Library,* Central Computer and Telecommunications Agency, Gildengate House, Upper Green Lane, Norwich, NR3 1DW.
- **[KNUT84]** Knuth, D. E., "Literate programming," *Computer Journal,* 27 (2): 97-111, 1984.
- **[LEHE80]** Lehman, M. M., "Programs, Lifecycles, and the Laws of Software Evolution," in *Proceedings of IEEE, 19*:1060-1076, 1980.
- **[LEHE84]** Lehman, M. M., "Program Evolution," *Information Processing Management, 20*: 19-36, 1984.
- **[LEIN78]** Lientz, B., Swanson, E. B., and Tompkins, G. E., "Characteristics of Applications Software Maintenance," *Communications of the ACM, 21*: 466-471, 1978.
- **[LEIN80]** Leintz, B., and Swanson, E. B., *Software Maintenance Management,* Addison-Wesley, Reading, MA, 1980.
- **[LEVE93]** Leveson, N. G., and Turner, C. S., "An Investigation of the Therac-25 Accidents," *IEEE Computer, 26* (7): 18-41, July 1993.
- **[McDER91]** McDermid, J. (Ed.), *SERB: Software Engineering Reference Book,* Butterworth-Heinemann, 1991.
- **[NYAR95]** Nyary, E., and Sneed, H., "Software Maintenance Offloading at the Union Bank of Switzerland," in *Proceedings of IEEE International Conference on Software Maintenance,* Nice, France, pp. 98-108 October 1995.
- **[RICH90]** Rich, C., and Waters, R. C., *The Programmer's Apprentice,* Addison-Wesley, Reading, MA, 1990.
- **[ROBS91]** Robson, D. J., Bennett, K. H., Cornelius, B. J., and Munro, M., "Approaches to Program Comprehension," *Journal of Systems Software, 14* (1): 1991.

- **[SMIT92]** Smith, M. D., and Robson, D. J., "A Framework for Testing Object-Oriented Programs," *Journal of Object-Oriented Programming, 5* (3): 45-53, June 1992.
- **[SMIT95]** Smith, S. R., Bennett, K. H., and Boldyreff, C., "Is Maintenance Ready for Evolution?" in *Proceedings of IEEE International Conference on Software Maintenance,* Nice, pp. 367-372, October 1995, IEEE Computer Society Press, (1995).
- **[SNEE91]** Sneed, H., "Economics of Software Re-Engineering," *Journal of Software Maintenance: Research and Practice,* 3(3): 163-182, Sept. 1991.
- **[SNEE93]** Sneed, H., and Nyary, E., "Downsizing Large Application Programs," in *Proceedings of IEEE International Conference on Software Maintenance,* Montreal, 1993, pp. 110-119, IEEE Computer Society Press, 1995.
- **[TURN93]** Turner, C. D., and Robson, D. J., "The State-Based Testing of Object-Oriented Programs," in *Proceedings of IEEE Conference on Software Maintenance*, Montreal, September 1993, pp. 302-310.
- **[TURN95]** Turner, C. D., and Robson, D. J., "A State-Based Approach to the Testing of Class-Based Programs," *Software—Concepts and Tools*, 16 (3): 106-112, 1995.
- **[TURV94]** Turver, R. J., and Munro, M., "An Early Impact Analysis Technique for Software Maintenance," *Journal of Software Maintenance: Research and Practice, 6* (1): 35-52, Jan. 1994.
- **[WALT94]** Walton, D. S., "Maintainability Metrics," in *Proceedings of the Centre for Software Reliability Conference,* Dublin, 1994. Available from the Centre for Software Reliability, City University, London, U.K.
- **[WARD93]** Ward, M. P., "Abstracting a Specification from Code," *Journal of Software Maintenance: Practice and Experience, 5* (2): 101-122, June 1993.
- **[WARD94]** Ward, M. P, "Reverse Engineering through Formal Transformation," *Computer Journal, 37* (9) 1994.
- **[WARD94a]** Ward, M. P., "Language Oriented Programming," *Software—Concepts and Tools,* 15: 147-161, 1994.
- **[WEIS89]** Weiss, D. M., *Evaluating Software Development by Analysis of Change,* Ph. D. Dissertation, University of Maryland, USA, 1989.
- **[WILD93]** Wilde, N., "Software Impact Analysis: Processes and Issues," Durham University Technical Report 7/93, 1993.
- **[YAU87]** Yau, S. S., and Liu, S., "Some Approaches to Logical Ripple Effect Analysis," Technical Report, SERC, USA, 1987.
- **[YOUN93]** Younger, E., "Documentation," in *The REDO Compendium,* Ed. van Zuylen, Wiley, 1993.
- **[YOUN94]** Younger, E., and Ward, M. P., "Inverse Engineering – A Simple Real Time Program," *Journal of Software Maintenance: Research and Practice, 6*: 197-234, 1994.
- **[ZUYL93]** van Zuylen, H. (Ed.), *The REDO Compendium,* Wiley, 1993.

Chapter 5.2

Essentials of Software Maintenance

Richard Hall Thayer and Merlin Dorfman

This is the fifth knowledge area (KA) in a reference guide to aid individual software engineers in a greater understanding the IEEE SWEBOK [2013] and in passing the IEEE CSDP/CSDA certification exams.

In this KA, we introduce the concepts and terminology that form an underlying basis for understanding the role and scope of software maintenance and activities required to provide cost-effective support for software. Activities are performed during the pre-delivery stage, as well as post-delivery.

The chapter starts with the CSDP Exam Specification for the KA of software engineering maintenance. This list of exam specifications is reported to be the same list that the exam writers used to write the exam questions. Therefore it is the best source of help for the exam takers.

Chapter 5 covers the CSDP exam specifications for the software maintenance engineering KA [Software Exam Specification, Version 2, 18 March 2009]:

1. Software maintenance fundamentals (definitions and terminology; nature of maintenance; need for maintenance; majority of maintenance costs; evolution of software; categories of maintenance)
2. Key issues in software maintenance (technical issues [limited understanding, testing, impact analysis, maintainability]; management issues [organizational objectives, staffing, process, organizational aspects, outsourcing]; maintenance cost estimation; measures
3. Maintenance process (maintenance processes; maintenance activities)
4. Techniques for maintenance (program comprehension; re-engineering; reverse engineering; re-factoring; migration; retirement; disaster recovery techniques; software maintenance tools)

In software engineering, software maintenance is the process of enhancing and optimizing deployed software (software release), as well as remedying defects. Software maintenance is one of the phases in the software development process, and follows deployment of the software into the field. The software maintenance phase involves changes to the software in order to correct defects and deficiencies found during field usage as well as the addition of new functionality to improve the software's usability and applicability.

The software maintenance phase is an explicit part of the waterfall model of the software development process [Royce 1970], which was developed during the structured programming movement of computer programming. Another major model, the spiral model [Boehm 1998], makes no explicit mention of a maintenance phase. Nevertheless, this activity is notable, considering that, according to conventional wisdom, two-thirds of a software system's lifetime cost involves maintenance.

In a formal software development environment, the developing organization or team will have some mechanisms to document and track defects and deficiencies. Software, just like most other products, is typically released with a known set of defects and deficiencies. The software is

released with the issues because the development organization decides the utility and value of the software at a particular level of quality outweighs the impact of the known defects and deficiencies; i.e., it is not cost effective to delay release while the known problems are being resolved.

The known issues are normally documented in a letter of operational considerations or release notes so that the users of the software will be able to work around the known issues and will know when the use of the software would be inappropriate for particular tasks.

With the release of the software, other, undocumented defects and deficiencies will be discovered by the users of the software. As these issues are reported to the development organization, they will be entered into the defect tracking system.

The people involved in the software maintenance phase are expected to work on these known issues, address them, and prepare for a new release of the software, known as a maintenance release, which will address the documented issues [SWEBOK 2004].

5.1 Software Maintenance Fundamentals

This section introduces the concepts and terminology that form an underlying basis to understanding the role and scope of software maintenance. The topics provide definitions and emphasize why there is a need for maintenance. Categories of software maintenance are critical to understanding its underlying meaning.

5.1.1 Definitions and terminology. *Software maintenance* is defined as the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.

The IEEE/EIA 12207.0-1996 Standard for Software Life-cycle Processes essentially depicts maintenance as one of the primary processes, and describes maintenance as the process of a software product undergoing "modification to code and associated documentation due to a problem or the need for improvement. The objective is to modify the existing software product while preserving its integrity." The standard also addresses maintenance activities prior to delivery of the software product, but only in an *information appendix* of the standard.

5.1.2 Nature of maintenance. Software maintenance sustains the software product throughout its operational life cycle. Modification requests are logged and tracked, the impact of proposed changes is determined, code and other software artifacts are modified, testing is conducted, and a new version of the software product is released. Also, training and daily support are provided to users. Pfleeger [2001] states that "maintenance has a broader scope, with more to track and control" than development."

A maintainer is defined by IEEE/EIA 12207 as an organization that performs maintenance activities; this sometimes refers to individuals who perform those activities, contrasting them with the developers.

IEEE/EIA 12207 identifies the primary activities of software maintenance as: process implementation; problem and modification analysis; modification implementation; maintenance review/acceptance; migration; and retirement.

Maintainers can learn from the developer's knowledge of the software. Contact with the developers and early involvement by the maintainer helps reduce the maintenance effort. In some instances, the software engineer cannot be reached or has moved on to other tasks, which creates an additional challenge for the maintainers. Maintenance must take the products of the develop-

ment, for example code or documentation, and support them immediately and evolve/maintain them progressively over the software life cycle.

5.1.3 Need for maintenance. Maintenance is needed to ensure that the software continues to satisfy user requirements (or to be modified to satisfy requirements if it is found deficient in that regard). Maintenance is applicable to software developed using any software life-cycle model (for example, spiral). The system changes due to corrective and non-corrective software actions. Maintenance must be performed in order to:

- Correct faults
- Improve the design
- Implement enhancements
- Interface with other systems
- Adapt programs so that different hardware, software, system features, and telecommunications facilities can be used
- Migrate legacy software
- Retire software

The maintainer's activities comprise four key characteristics, according to Pfleeger [2001].

- Maintaining control over the software's day-to-day functions
- Maintaining control over software modification
- Perfecting existing functions
- Preventing software performance from degrading to unacceptable levels

5.1.4 Majority of maintenance costs. Maintenance consumes a major share of software life cycle financial resources. A common perception of software maintenance is that it merely fixes faults. However, studies and surveys over the years have indicated that the majority, over 80%, of the software maintenance effort is used for non-corrective actions. Jones [1998] describes the way in which software maintenance managers often group enhancements and corrections together in their management reports. This inclusion of enhancement requests with problem reports contributes to some of the misconceptions regarding the high cost of corrections. Understanding the categories of software maintenance helps to understand the structure of software maintenance costs. Also, understanding the factors that influence the maintainability of a system can help to contain costs. Pfleeger presents some of the technical and non-technical factors affecting software maintenance costs, as follows:

- Application type
- Software novelty
- Software maintenance staff availability
- Software life span
- Hardware characteristics
- Quality of software design, construction, documentation and testing

5.1.5 Evolution of software. Lehman [1997] first addressed software maintenance and evolution of systems in 1969. Over a period of twenty years, his research led to the formulation of eight "Laws of Evolution." Key findings include that maintenance is evolutionary development, and that maintenance decisions are aided by understanding what happens to systems (and software) over time. Others state that maintenance is continued development, except that there is an extra input (or constraint)–existing large software is never complete and continues to evolve. As it evolves, it grows more complex unless some action is taken to reduce this complexity.

Since software demonstrates regular behavior and trends, these can be measured. Attempts to develop predictive models to estimate maintenance effort have been made, and. as a result, useful management tools have been developed.

5.1.6 Categories of maintenance. Maintenance consists of four parts:

- ***Corrective maintenance*** — Reactive modification of a software product performed after delivery to correct discovered problems. It deals with fixing bugs (faults) in the code.
- ***Adaptive maintenance*** — Modification of a software product performed after delivery to keep a software product usable in a changed or changing environment. It deals with adapting the software to new environments.
- ***Perfective maintenance*** — Modification of a software product after delivery to improve performance or maintainability. It deals with updating the software according to changes in user requirements.
- ***Preventive maintenance*** — Modification of a software product after delivery to detect and correct latent faults in the software product before they become effective faults. It deals with updating documentation and making the software more maintainable.

All changes to the system can be characterized by these four types of maintenance. Corrective maintenance is "traditional maintenance" while the other types are considered as "software evolution."

5.2 Key Issues in Software Maintenance

A number of key issues must be dealt with to ensure the effective maintenance of software. It is important to understand that software maintenance provides unique technical and management challenges for software engineers. Trying to find a fault in software containing 500K lines of code that the software engineer did not develop is a good example. Similarly, competing with software developers for resources is a constant battle. Planning for a future release, while coding the next release and sending out emergency patches for the current release, also creates a challenge. The following section presents some of the technical and management issues related to software maintenance. They have been grouped under the following topic headings [SWEBOK 2004]:

- Technical issues
- Maintainability
- Management issues
- Maintenance cost estimation
- Software maintenance measurement

5.2.1 Technical issues. The technical issues in maintaining software are [SWEBOK 2004]:

- ***Limited understanding*** — Limited understanding refers to how quickly a software engineer can understand where to make a change or a correction in software which this individual did not develop. Research indicates that some 40% to 60% of the maintenance effort is devoted to understanding the software to be modified. Thus, the topic of software comprehension is of great interest to software engineers.

  Comprehension is more difficult in text-oriented representation, for example in source code, where it is often difficult to trace the evolution of software through its releases/versions if changes are not documented and when the developers are not available to explain it, which is often the case. Thus, software engineers may initially have a limited understanding of the software, and much has to be done to remedy this problem.

- ***Testing*** — The cost of repeating full testing on a major piece of software can be significant in terms of time and money. *Regression testing*, the selective retesting of system or software components to verify that the modifications have not caused unintended effects, is important to maintenance. As well, finding time to test is often difficult. There is also the challenge of coordinating tests when different members of the maintenance team are working on different problems at the same time. When software performs critical functions, it may be impossible to bring it offline to test.

- ***Impact analysis*** — *Impact analysis* describes how to conduct, cost effectively, a complete analysis of the impact of a change in existing software. Maintainers must possess an intimate knowledge of the software's structure and content. They use that knowledge to perform impact analysis, which identifies all systems and software products affected by a software change request and develops an estimate of the resources needed to accomplish the change. Additionally, the risk of making the change is determined. The change request, sometimes called a modification request (MR) and often called a problem report (PR), must first be analyzed and translated into software terms. It is performed after a change request enters the software configuration management process. The objectives of impact analysis are [Arthur 1988]:

  - Determination of the scope of a change in order to plan and implement work
  - Development of accurate estimates of resources needed to perform the work
  - Analysis of the cost/benefits of the requested change
  - Communication to others of the complexity of a given change

  The severity of a problem is often used to decide how and when a problem will be fixed. The software engineer then identifies the affected components. Several potential solutions are provided and then a recommendation is made as to the best course of action. Software designed with maintainability in mind greatly facilitates impact analysis.

- ***Maintainability*** — How does one promote and follow up on maintainability issues during development? The IEEE defines maintainability as the ease with which software can be maintained, enhanced, adapted, or corrected to satisfy specified requirements [IEEE610.12-90]. Maintainability can be further subdivided into the sub-characteristics of analyzability, changeability, stability, and testability [ISO/IEC 9126:2001].

Maintainability sub-characteristics must be specified, reviewed, and controlled during the software development activities in order to reduce maintenance costs. If this is done successfully, the maintainability of the software will improve. This is often difficult to achieve because the maintainability sub-characteristics are not an important focus during the software development process. The developers are preoccupied with many other things and often disregard the maintainer's requirements. This in turn can, and often does, result in a lack of current, accurate system documentation, which is a leading cause of difficulties in program comprehension and impact analysis. It has also been observed that the presence of systematic and mature processes, techniques, and tools helps to enhance the maintainability of a system.

5.2.2 Management issues. The management issues in maintaining software are:

- ***Alignment with organizational objectives*** — Organizational objectives describe how to demonstrate the return on investment of software maintenance activities. Bennett [2001] states that "initial software development is usually project-based, with a defined time scale and budget. The main emphasis is to deliver on time and within budget to meet user needs. In contrast, software maintenance often has the objective of extending the life of software for as long as possible. In addition, it may be driven by the need to meet user demand for software updates and enhancements. In both cases, the return on investment is much less clear, so that the view at senior management level is often of a major activity consuming significant resources with no clear quantifiable benefit for the organization."

- ***Staffing*** — Staffing refers to how to attract and keep software maintenance staff. Maintenance is often not viewed as glamorous work. Deklava [1992] provides a list of staffing-related problems based on survey data. As a result, software maintenance personnel are frequently viewed as "second-class citizens" and morale therefore suffers.

- ***Process*** — Software process is a set of activities, methods, practices, and transformations that people use to develop and maintain software and the associated products. At the process level, software maintenance activities share much in common with software development (for example, software configuration management is a crucial activity in both). Maintenance also requires several activities that are not found in software development. These activities present challenges to management.

- ***Organizational aspects of maintenance*** — Organizational aspects describe how to identify which organization and/or function will be responsible for the maintenance of software. The team that develops the software is not necessarily assigned to maintain the software once it is operational.

  In deciding where the software maintenance function will be located, software engineering organizations may, for example, stay with the original developer or go to a separate team (or maintainer). Often, the maintainer option is chosen to ensure that the software runs properly and evolves to satisfy changing user needs. Since there are many pros and cons to each of these options, the decision should be made on a case-by-case basis. What is important is the delegation or assignment of the maintenance responsibility to a single group or person, regardless of the organization's structure.

- ***Outsourcing*** — Outsourcing of maintenance is becoming a major industry. Large corporations are outsourcing entire portfolios of software systems, including software mainte-

nance. More often, the outsourcing option is selected for less mission-critical software, as companies are unwilling to lose control of the software used in their core business. Carey [1994] reports that some will outsource only if they can find ways of maintaining strategic control. However, control measures are hard to find. One of the major challenges for the outsourcers is to determine the scope of the maintenance services required and the contractual details. It has been stated that 50% of outsourcers provide services without any clear service-level agreement [McCracken 2002]. Outsourcing companies typically spend a number of months assessing the software before they will enter into a contractual relationship. Another challenge identified is the transition of the software to the outsourcer.

5.2.3 Maintenance cost estimation. Software engineers must understand the different categories of software maintenance, discussed above, in order to address the question of estimating the cost of software maintenance. For planning purposes, estimating costs is an important aspect of software maintenance.

- ***Cost estimation*** — It was mentioned in Section 5.2.1 that impact analysis identifies all systems and software products affected by a software change request and develops an estimate of the resources needed to accomplish that change.
- ***Parametric models*** — Some work has been undertaken in applying parametric cost modeling to software maintenance. The significance is that data from past projects are needed in order to use the models. Jones [1988] discusses all aspects of estimating costs, including function points, and provides a detailed chapter on maintenance estimation.
- ***Experience*** — Experience, in the form of expert judgment (using the Delphi technique, for example), analogies, and a work breakdown structure, are several approaches which should be used to augment data from parametric models. Clearly, the best approach to maintenance estimation is to combine empirical data and experience. These data should be provided as a result of a measurement program.

5.2.4 Software maintenance measurement. Grady and Caswell [1987] discuss establishing a corporate-wide software measurement program, in which software maintenance measurement forms and data collection are described. The Practical Software and Systems Measurement (PSM) Project [http://www.psmsc.com/] describes an issue-driven measurement process that is used by many organizations and is quite practical [McCracken 2002].

There are software measures that are common to all endeavors; the following are categories that the Software Engineering Institute (SEI) has identified: size/effort/productivity, requirements volatility, progress, quality, and resource utilization. These measures constitute a good starting point for the maintainer [Pigoski 1997].

- ***Specific measures*** — Abran and Nguyenkim [1993] presents internal benchmarking techniques to compare different internal maintenance organizations. The maintainer must determine which measures are appropriate for the organization in question [IEEE1219-1998] suggests measures that are more specific to software maintenance measurement programs. That list includes a number of measures for each of the four sub-characteristics of maintainability:
    - **Analyzability** — Measures of the maintainer's effort or resources expended in trying to diagnose deficiencies or causes of failure, or in identifying parts to be modified

- **Changeability** — Measures of the maintainer's effort associated with implementing a specified modification
- **Stability** — Measures of the unexpected behavior of software, including that encountered during testing
- **Testability** — Measures of the maintainer's and users' effort in trying to test the modified software

Certain measures of the maintainability of software can be obtained using available commercial tools.

5.3 Maintenance Process

The maintenance process KA provides references and standards used to implement the software maintenance process. The maintenance activities topic differentiates maintenance from development and shows its relationship to other software engineering activities.

5.3.1 Maintenance processes. The maintenance processes provide needed activities and detailed inputs/outputs to those activities, and are described in software maintenance standard [IEEE 1219-1998].

The maintenance process model described in this standard starts with the software maintenance effort during the post-delivery stage and discusses items such as planning for maintenance. The maintenance process is depicted in Figure 1.

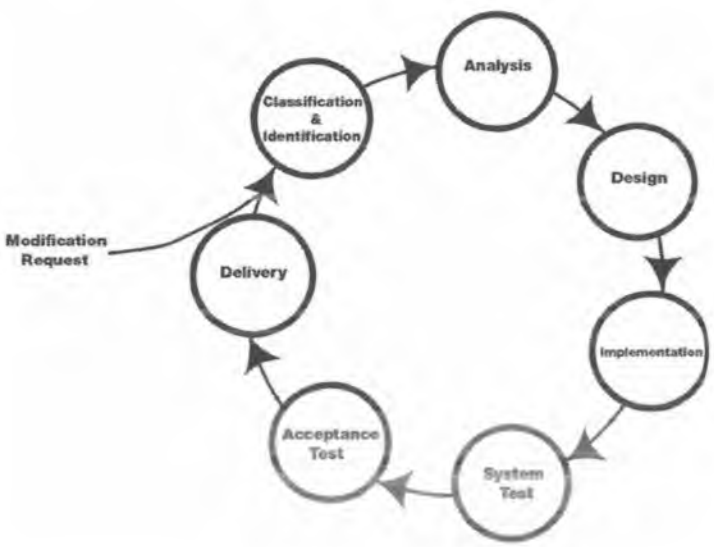


Figure 1: Software maintenance process [IEEE Std 1219-1998]

[ISO/IEC 14764-1999] is an elaboration of the IEEE/EIA 12207.0-1996 maintenance process. The activities of the ISO/IEC maintenance process are similar to those of the IEEE, except that they are aggregated a little differently.

Each of the ISO/IEC 14764 primary software maintenance activities is broken down into tasks, as follows:

- Process implementation

- Problem and modification analysis
- Modification implementation
- Maintenance review/acceptance
- Migration
- Software retirement

5.3.2 Maintenance activities. As already noted, many maintenance activities are similar to those of software development. Maintainers perform analysis, design, coding, testing, and documentation. They must track requirements in their activities just as is done in development, and update documentation as baselines change. ISO/IEC14764-1999 recommends that, when a maintainer refers to a similar development process, he must adapt it to meet his specific need. However, for software maintenance, some activities involve processes unique to software maintenance.

- ***Unique activities*** — There are a number of processes, activities, and practices that are unique to software maintenance, for example:
  - A *transition* is a controlled and coordinated sequence of activities during which software is transferred progressively from the developer to the maintainer.
  - Modification request work over a certain size/effort/complexity may be rejected by maintainers and rerouted to a developer.
  - Modification request and problem report help desk: an end-user support function that triggers the assessment, prioritization, and costing of modification requests.
  - *Impact analysis* describes how to conduct, cost effectively, a complete analysis of the impact of a change in existing software.
  - Software support: help and advice to users concerning a request for information (for example, business rules, validation, data meaning and ad-hoc requests/reports).
  - Service level agreements (SLAs) and specialized (domain-specific) maintenance contracts that are the responsibilities of the maintainers.
- ***Supporting activities*** — Maintainers may also perform supporting activities, such as software maintenance planning, software configuration management, verification and validation, software quality assurance, reviews, audits, and user training.
- ***Maintenance planning activity*** — An important activity for software maintenance is planning, and maintainers must address the issues associated with a number of planning perspectives:
  - Business planning (organizational level)
  - Maintenance planning (transition level)
  - Release/version planning (software level)
  - Collect the dates of availability of individual requests
  - Agree with users on the content of subsequent releases/versions
  - Identify potential conflicts and develop alternatives

- Assess the risk of a given release and develop a back-out plan in case problems should arise
- Inform all the stakeholders

- Whereas software development projects can typically last from some months to a few years, the maintenance phase usually lasts for many years. Making estimates of resources is a key element of maintenance planning. Those resources should be included in the developers' project planning budgets. Software maintenance planning should begin with the decision to develop a new system and should consider quality objectives [IEEE1061-1998]. A concept document should be developed, followed by a maintenance plan.

  The concept document for maintenance should address:

  - The scope of the software maintenance
  - Adaptation of the software maintenance process
  - Identification of the software maintenance organization
  - An estimate of software maintenance costs

The next step is to develop a corresponding software maintenance plan. This plan should be prepared during software development, and should specify how users will request software modifications or report problems. Software maintenance planning is addressed in IEEE 1219-1998, and ISO/IEC14764-1999 provides guidelines for a maintenance plan.

Finally, at the highest level, the maintenance organization will have to conduct business planning activities (budgetary, financial, and human resources) just like all the other divisions of the organization.

- ***Software configuration management*** — The IEEE Standard for Software Maintenance, [IEEE 1219-1998], describes software configuration management as a critical element of the maintenance process. Software configuration management procedures should provide for the verification, validation, and audit of each step required in identifying, authorizing, implementing, and releasing the software product.

  It is not sufficient to simply track modification requests or problem reports. The software product and any changes made to it must be controlled. This control is established by implementing and enforcing an approved software configuration management (SCM) process. SCM for software maintenance is different from SCM for software development in the number of small changes that must be controlled on operational software. The SCM process is implemented by developing and following a configuration management plan and operating procedures. Maintainers participate in Configuration Control Boards to determine the content of the next release/version.

- ***Software quality*** — It is not sufficient, either, to simply hope that increased quality will result from the maintenance of software. It must be planned and processes implemented to support the maintenance process. The activities and techniques for software quality assurance (SQA), V&V, reviews, and audits must be selected in concert with all the other processes to achieve the desired level of quality. It is also recommended that the maintainer adapt the software development processes, techniques, and deliverables, for instance, testing documentation and test results.

5.4 Techniques for Maintenance

This section introduces some of the generally accepted techniques used in software maintenance documents.

5.4.1 Program comprehension. Programmers spend considerable time reading and understanding programs in order to implement changes. Code browsers are key tools for program comprehension. Clear and concise documentation can aid in program comprehension.

5.4.2 Reengineering. Reengineering is defined as the examination and alteration of software to reconstitute it in a new form, and includes the subsequent implementation of the new form. Dorfman and Thayer [2001] state that reengineering is the most radical (and expensive) form of alteration. Others believe that reengineering can be used for minor changes. It is often not undertaken to improve maintainability, but to replace aging legacy software. Arnold [1993] provides a comprehensive compendium of topics, for example: concepts, tools and techniques, case studies, and risks and benefits associated with reengineering.

5.4.3 Reverse engineering. Reverse engineering is the process of analyzing software to identify the software's components and their interrelationships and to create representations of the software in another form or at higher levels of abstraction. Reverse engineering is passive; it does not change the software, or result in new software. Reverse engineering efforts produce call graphs and control flow graphs from source code. One type of reverse engineering is *redocumentation*. Another type is *design recovery*. *Refactoring* is program transformation that reorganizes a program without changing its behavior, and is a form of reverse engineering that seeks to improve program structure.

Finally, *data reverse engineering* has gained in importance over the last few years where logical schemas are recovered from physical databases.

5.4.4 Re-factoring. [*Code*] *refactoring* is the process of changing a computer program's source code without modifying its external *functional* behavior in order to improve some of the *non-functional* attributes of the software. Advantages include improved code readability and reduced complexity to improve the maintainability of the source code, as well as a more expressive internal architecture or object model to improve extensibility [http://en.wikipedia.org/wiki/Code_refactoring].

5.4.5 Migration. [*Software*] *migration* is the process of moving from the use of one operating environment to another operating environment that is, in most cases, thought to be a better one. For example, moving from a Windows NT Server to a Windows 2000 Server would usually be considered a migration because it involves making sure that new features are exploited, old settings do not require changing, and taking steps to ensure that current applications continue to work in the new environment. Migration could also mean moving from Windows NT to a UNIX-based operating system (or the reverse). Migration can involve moving to new hardware, new software, or both. Migration can be small-scale, such as migrating a single system, or large-scale, involving many systems, new applications, or a redesigned network [http://en.wikipedia.org/wiki/Software_modernization].

5.4.6 Retirement. The act of retirement of application programs usually involves relocating data from the legacy application database to another data repository or archive store that can be accessed independently using industry-standard reporting or business intelligence tools. Application retirement allows IT departments within companies to reduce the software, hardware, and

resources required to manage legacy data. Application retirement is also referred to as application decommissioning and application "sunsetting." [http://en.wikipedia.org/wiki/Application_retirement].

5.4.7 Disaster recovery techniques. *Disaster recovery* is the process, policies and procedures related to preparing for recovery or continuation of technology infrastructure critical to an organization after a natural or human-induced disaster. Disaster recovery is a subset of business continuity. While business continuity involves planning for keeping all aspects of a business functioning in the midst of disruptive events, disaster recovery focuses on the IT or technology systems that support business functions [http://en.wikipedia.org/wiki/Disaster_recovery].

Disasters can be classified in two broad categories. The *first* is natural disasters such as floods, hurricanes, tornadoes, or earthquakes. While preventing a natural disaster is very difficult, measures such as good planning, which includes mitigation measures, can help reduce or avoid losses. The *second* category is man-made disasters. These include hazardous material spills, infrastructure failure, or terrorism. In these instances surveillance and mitigation planning are invaluable towards avoiding or lessening losses from these events.

It is estimated that most large companies spend between 2% and 4% of their IT budget on disaster recovery planning, with the aim of avoiding larger losses in the event that the business cannot continue to function due to loss of IT infrastructure and data. Of companies that had a major loss of business data, 43% never reopen, 51% close within two years, and only 6% will survive long-term [http://en.wikipedia.org/wiki/Disaster_recovery].

The following is a list of the most common strategies for disaster recovery techniques:

- Backups recorded on tape and sent off-site at regular intervals
- Backups recorded to disk on-site and automatically copied to off-site disk, or recorded directly to off-site disk
- Replication of data to an off-site location, which overcomes the need to restore the data (only the systems then need to be restored or synchronized).
- High availability systems that keep both the data and system replicated off-site, enabling continuous access to systems and data

In addition to preparing for the need to recover systems, organizations must also implement precautionary measures with an objective of preventing a disaster in the first place. These may include some of the following:

- Local mirrors of systems and/or data and use of disk protection technology such as RAID. *RAID,* an acronym for *Redundant Array of Independent Disks*, is a technology that provides increased storage functions and reliability through redundancy. This is achieved by combining multiple disk drive components into a logical unit, where data is distributed across the drives in one of several ways called "RAID levels."
- Surge protectors minimize the effect of power surges on delicate electronic equipment
- Uninterruptible power supply (UPS) and/or a backup generator to keep systems running in the event of a power failure
- Fire prevention techniques such as alarms and fire extinguishers, both manual and automatic

- Anti-virus software and other security measures

5.5 Software Maintenance Tools

5.5.1 Introduction. A software maintenance tool is an artifact that supports a software maintainer in performing a task. The use of tools for software maintenance simplifies the tasks and increases efficiency and productivity.

There are several criteria for selecting the right tool for the task. These criteria are capability, features, cost/benefit, platform, programming language, ease of use, openness of architecture, stability of vendor, and organizational culture.

Capability decides whether the tool is capable of fulfilling the task. Once it has been decided that a method can benefit from being automated, then the features of the tool need to be considered for the job.

The tool must be analyzed for the benefits it brings compared to its cost. The benefit indicators of a tool are quality, productivity, responsiveness, and cost reduction. The environment that the tool runs on is called the platform. The language of the source code is called the programming language. It's important to select a tool that supports a language that is an industry standard.

The tool should have a similar feel to the ones that the users are already familiar with. The tool should have the ability to be integrated with different vendors' tools. This will help when a tool will need to run with other tools. The openness of the architecture plays an important role when the maintenance problem is complex. Therefore, it is not always sufficient to use only one tool. There may need to be multiple tools running together.

It is also important to consider the vendor's credibility. The vendor should be capable of supporting the tool in the future. If the vendor is not stable, the vendor could run out of business and not be able to support the tool.

Another important factor is the culture of the organization. Every culture has its own work pattern. Therefore, it is important to take into consideration whether the tool is going to be accepted by the target users.

- The chosen tools must support program understanding and reverse engineering, testing, configuration management, and documentation.
- Selecting a tool that promotes understanding is very important in the implementation of change since a large amount of time is used to study and understand programs.
- Tools for reverse engineering also accomplish the same goal. The tools mainly consist of visualization tools, which assist the programmer in drawing a model of the system.
- Examples of program understanding and reverse engineering tools include the program slicer, static analyzer, dynamic analyzer, cross-referencer, and dependency analyzer.

Slicing is the mechanical process of marking all the sections of a program text that may influence the value of a variable at a given point in the program. Program slicing helps the programmers select and view only the parts of the program that are affected by the changes. A static analyzer is used in analyzing the different parts of the program such as modules, procedures, variables, data elements, objects, and classes. A static analyzer allows general viewing of the

program text and generates summaries of contents and usage of selected elements in the program text, such as variables or objects.

A *dynamic analyzer* could be used to analyze the program while it is executing. A data flow analyzer is a static analysis tool that allows the maintainer to track all possible data flow and control flow paths in the program. It allows analysis of the program to better outline the underlying logic of the program. It also helps display the relationship between components of the system. A cross-referencer produces information on the usage of a program. This tool helps the user focus on the parts that are affected by the change.

A *dependency analyzer* assists the maintainer in analyzing and understanding the interrelationships between entities in a program. Such a tool provides capabilities to set up and query the database of the dependencies in a program. It also provides graphical representations of the dependencies.

Testing is the most time-consuming and demanding task in software maintenance. Therefore, it could benefit the most from tools. A test simulator tool helps the maintainer to test the effects of the change in a controlled environment before implementing the change on the actual system. A test case generator produces test data that is used to test the functionality of the modified system, while a test path generator helps the maintainer to find all the data flow and control flow paths affected by the changes.

Configuration management benefits from automated tools. Configuration management and version control tools help store the objects that form the software system. A source control system is used to keep a history of the files so that versions can be tracked and the programmer can keep track of the file changes [SWEBOK 2004].

5.5.2 Commercially available products. There are numerous products on the market available for software maintenance. Some maintenance tools are: [SWEBOK 2004].

- ***Bug tracking tools*** — A *bug [fault] tracking system* is a software application that is designed to help quality assurance and programmers keep track of reported software bugs in their work. It may be regarded as a type of issue tracking system. Many bug-tracking systems, such as those used by most open source software projects, allow users to enter bug reports directly. Other systems are used only internally in a company or organization doing software development. Typically bug tracking systems are integrated with other software project management applications.

  Having a bug tracking system is extremely valuable in software development, and they are used extensively by companies developing software products. Consistent use of a bug or issue tracking system is considered one of the attributes of a good software development organization [http://en.wikipedia.org/wiki/Bug_tracking_system].

- ***Debugger*** — A *debugger* or *debugging tool* is a computer program that is used to test and debug other programs (the "target" program). The code to be examined might alternatively be running on an *instruction set simulator* (ISS), a technique of great power due to its ability to halt when specific conditions are encountered, but will typically be somewhat slower than executing the code directly on the appropriate (or the same) processor. Some debuggers offer two modes of operation--full or partial simulation, to limit this impact [http://en.wikipedia.org/wiki/Debugger].

- ***Maintenance software package*** — A maintenance package lets users schedule preventative maintenance, generate automatic work orders, document equipment maintenance history, track assets, and inventory, track personnel, create purchase orders, and generate reports.
- ***Profiling*** — *Software profiling* or simply profiling, a form of dynamic program analysis (as opposed to static code analysis), is the investigation of a program's behavior using information gathered as the program executes. The usual purpose of this analysis is to determine which sections of a program to optimize--to increase its overall speed, decrease its memory requirements or sometimes both [http://en.wikipedia.org/wiki/Profiling_(computer_programming)].
- An ***instruction set simulator*** — A software system which can measure the totality of a program's behavior from invocation to termination.

5.5.3 Summary of tools. The task of software maintenance has become so vital and complex that automated support is required to perform it effectively. The use of tools simplifies tasks and increases efficiency and productivity. There are numerous tools available on the market for maintenance.

References

Additional information on the *software maintenance KA* can be found in the following documents:

- **[Abran & Nguyenkim 1993]** A. Abran and H. Nguyenkim, "Measurement of the Maintenance Process from a Demand-Based Perspective," *Journal of Software Maintenance: Research and Practice,* vol. 5, issue. 2, 1993, pp. 63-90.
- **[Arnold 1993]** R.S. Arnold, *Software Reengineering,* IEEE Computer Society Press, Los Alamitos, CA, 1993.
- **[Arthur 1988]** L.J. Arthur, *Software Evolution: The Software Maintenance Challenge*, Wiley, New York, 1988.
- **[Bennett 2001]** K.H. Bennett, "Software Maintenance: A Tutorial," in *Software Engineering*, M. Dorfman and R. Thayer, eds., IEEE Computer Society Press, Los Alamitos, CA, 2000.
- **[Boehm 1988]** B.W. Boehm, "A spiral model of software development and enhancement," *IEEE Computer, 21(5)*, 1988: 61-72.
- **[Carey 1994]** D. Carey, "Executive Round-Table on Business Issues in Outsourcing - Making the Decision," *CIO Canada*, June/July 1994.
- **[Deklava 1992]** S. Dekleva, "Delphi Study of Software Maintenance Problems," Presented at the *International Conference on Software Maintenance* (Orlando FL, 9-12 Nov. 1992).
- **[Dorfman & Thayer 2001]** M. Dorfman and R.H. Thayer, eds., *Software Engineering* (Vol. 1 & Vol. 2), IEEE Computer Society Press, Los Alamitos, CA 2001.
- **[Grady & Caswell 1987]** R.B. Grady and D.L. Caswell, *Software Metrics: Establishing a Company-Wide Program,* Prentice-Hall, Englewood Cliffs, NJ, 1987.

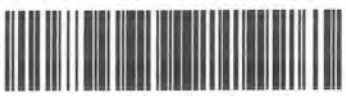
- **[IEEE 1219-1998]** IEEE Standard 1219-1998, *IEEE Standard for Software Maintenance,* IEEE Inc., New York, 1998.
- **[IEEE/EIA 12207.1-1996]** IEEE/EIA 12206.1-1997, *Guide for Information Technology - Software Life Cycle Processes,* IEEE, Inc., New York, 1996.
- **[IEEE1061-1998]** IEEE Standard 1061-1998, *IEEE Standard for a Software Quality Metrics Methodology*, IEEE, Inc., New York, 1998.
- **[IEEE610.12-1990]** [IEEE610.12-90] *IEEE Standard Glossary of Software Engineering Terminology,* IEEE, Inc., New York, 1990.
- **[ISO.IEC 9126:2001]** ISO/IEC 9126-1: *Software Engineering - Product Quality - Part 1: Quality Model,* International Organization for Standardization, 2001.
- **[ISO/IEC 14764-1999]** ISO/IEC 14764-1999, *Software Engineering-Software Maintenance*, ISO, and IEC, 1999.
- **[Jones 1998]** T.C. Jones, *Estimating Software Costs*, McGraw-Hill, New York, 1998.
- **[Lehman1997]** M.M. Lehman, "Laws of Software Evolution Revisited," Presented at EWSPT96, 1997.
- **[McCracken 2002]** B. McCracken, "Taking Control of IT Performance," Presented at InfoServer LLC, 2002.
- **[Pfleeger 2001]** S.L. Pfleeger, *Software Engineering: Theory and Practice,* 2nd edition, Prentice-Hall, Upper Saddle River, NJ, 2001.
- **[Pigoski 1997]** T.M. Pigoski, *Practical Software Maintenance: Best Practices for Managing your Software Investment*, First Ed., John Wiley & Sons, New York, 1997.
- **[Royce 1970]** W.W. Royce, "Managing the Development of Large Software Systems: Concepts and Techniques," *Proc. IEEE WESCON,* IEEE Computer Society Press, Los Angeles CA, 1970 (Ch.4).
- **[Wikipedia]** Wikipedia is a free web based encyclopedia enabling multiple users to freely add and edit online content. Definitions cited on Wikipedia and their related sources have been verified by the authors and other peer reviewers. Readers who would like to verify a source or a reference should search the subject on Google, and read the technical report found under Wikipedia.

Notes

Notes

Notes

Notes



Made in the USA
Charleston, SC
16 July 2014