МИНИСТЕРСТВО ПО РАЗВИТИЮ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И КОММУНИКАЦИЙ РЕСПУБЛИКИ УЗБЕКИСТАН

ТАШКЕНТСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ ИМЕНИ МУХАММАДА АЛ-ХОРАЗМИЙ

И.В.ХАН, Р.БАЙДУЛЛАЕВ, Ф.АБДУРАЗЗАКОВ, Б.ОТАХОНОВА

Учебное пособие по предмету «АРХИТЕКТУРА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ»

для студентов по направлению
5330600 – Программный инжиниринг

Ташкент 2022

Авторы: И.В.ХАН, Р.БАЙДУЛЛАЕВ, Ф.АБДУРАЗЗАКОВ, Б.ОТАХОНОВА. Учебное пособие по дисциплине «Архитектура программного обеспечения»/ТУИТ имени Мухаммада ал-Хоразмий. 88 с. Ташкент, 2022.

Нынешнее состояние науки и техники требует от инженерно-технических и научных работников знания средств вычислительной техники и умения обращения с современными программно-техническими комплексами. Эффективное применение компьютеров для решения инженерных и научных задач невозможно без знаний основных методов составления схем алгоритмов, написания действенного программного обеспечения на языке программирования, использования пакетов программ инженерной графики и математических систем.

Целью изучения дисциплины является изучение системы инженерных принципов для создания программного обеспечения с заданными характеристиками. После выполнения каждой лабораторной работы студент оформляет отчет, который включает следующее: название и цель работы, краткий порядок действий, ответы на контрольные вопросы. Полученные при изучении дисциплины знания и навыки могут быть востребованы при курсовом проектировании и в дальнейшем процессе обучения студента в вузе.

Учебное пособие предназначено для преподавателей и студентов высших учебных заведений по направлению «5330600 – Программный инжиниринг».

Напечатано по решению учебно–методического совета ТУИТ (протокол №_____ от «____» _____ 2022 г.)

Оглавление

Введение

Информационная система является системой людей, информационных технологий и бизнес-процессов. Существуют следующие подходы к классификации информационных систем:

По степени автоматизации:

- ручные (неэлектронные);
- электронные;
  - автоматические;
  - автоматизированные.

По процессам обработки данных:

- поисковые (информационно-поисковые) – ввод и хранение данных;
- решающие – преобразование (переработка данных).

По направлению деятельности различают:

- производственные системы;
  - автоматизированные системы управления производством;
  - автоматизированные системы управления технологическими процессами;
  - автоматизированные системы управления техническими средствами
- административные системы (человеческих ресурсов);
- финансовые и учетные системы;
- системы маркетинга;
- информационные системы здравоохранения.

Технология – это набор правил, методик и инструментов, позволяющих наладить производственный процесс выпуска какого-либо продукта. Информационная технология – процесс различных операций и действий над данными. Все процессы преобразования информации в информационной системе осуществляются с помощью информационных технологий.

Электронные информационные системы и технологии реализуются в виде программного обеспечения. Ввиду этого важной частью процесса проектирования информационных систем являются процессы проектирования и реализации программного обеспечения.

Проектирование программного обеспечения — процесс создания проекта программного обеспечения, а также дисциплина, изучающая методы проектирования. Целью проектирования является определение

внутренних свойств программного обеспечения и детализации его внешних свойств, на основе выданных заказчиком и впоследствии проанализированных требований. В процессе создания программного обеспечения проектированию подлежит следующее:

1. высокоуровневая архитектура программного обеспечения;
2. низкоуровневая (внутренняя) архитектура программных компонентов;
3. пользовательский интерфейс;
4. сценарии взаимодействия пользователя с программным продуктом;
5. структуры данных;
6. модель предметной области;
7. алгоритмы.

На начальном этапе развития программной инженерии основными задачами проектирования были проектирование алгоритмов и структур данных. Помимо этого, проектировалась модульная структура программных приложений. С появлением объектно-ориентированной парадигмы программирования проектируются взаимодействия между объектами классов, появляются объектно-ориентированные шаблоны микроархитектуры (шаблоны проектирования) такие как обозреватель, синглентон, фасад, прокси и другие. Более сложной организацией программного обеспечения является программная система, состоящая из компонентов. В настоящее время первоочередной задачей проектирования является проектирования архитектуры программных систем – макроархитектуры.

Проектирование интерфейса пользователя включает не только графический дизайн, но проектирование взаимодействия пользователя и программного приложения (*User Experience design, UX design*), где учитывается количество переходов и схемы между окнами, среднее время на выполнение определенного действия и другие параметры. Проектирование интерфейса пользователя существенно отличается для различных типов клиентских устройств и вида программного приложения (веб-приложение, полноэкранное игровое программное приложение, корпоративное приложение). Подход к проектированию программного приложения, при котором наибольшее значение имеет качество человеко-машинного взаимодействие, именуется *Look and Feed Driven Design* [Chung C. Pro Objective-C Design Patterns for iOS. - New York, USA: Appress, 2011.]. Данный подход имеет особую популярность для мобильных программных приложений, где эстетические предпочтения пользователя оказывают значительное влияние на решение о покупке и установке программного приложения.

1. Общие сведения об архитектуре программного обеспечения и проектировании

1.1. Архитектура программного обеспечения

Упрощенное определение архитектуры – это разделение целой системы на части с учетом отношений между этими частями. Подобное разделение позволяет группе людей продуктивно работать и решать вместе намного больше задач, чем каждый из них смог их решить индивидуально.

Архитектура программного обеспечения – описание структуры программной системы, включающее программные компоненты, их свойства и отношения между ними. Правильно выбранная архитектура помогает создать успешный программный продукт, который в дальнейшем может легко адаптироваться к изменяющимся требованиям заказчика. Благодаря правильно спроектированной архитектуре программный продукт удовлетворяет не только функциональным требованиям, но и атрибутом качества.

Мартин Фаулер в книге «Архитектура корпоративных программных приложений» приводит несколько возможных определений архитектуры ПО: «Разделение системы на составные части в самом первом приближении; принятие решений, которые трудно изменить впоследствии; выработка множества возможных вариантов архитектуры для системы; важность с точки зрения архитектуры различных аспектов может меняться в процессе жизненного цикла системы; и, в конечном счете, под архитектурой можно понимать то, что имеет значение» [1].

В книге «Архитектура ПО на практике, 2-е издание» дается следующее определение архитектуры программного обеспечения: «Архитектура программной или вычислительной системы – это структура или структуры системы, включающие программные элементы, видимые извне свойства этих элементов и взаимоотношения между ними. Архитектура касается внешней части интерфейсов; внутренние детали элементов – детали, относящиеся исключительно к внутренней реализации – не являются архитектурными» [1].

Исходные сведения к процессу создания архитектуры ПО могут быть получены в виде ответов на следующие вопросы:

- Как пользователь будет использовать приложение?
- Каким способом ПО будет развертываться?
- Кем оно будет обслуживаться в процессе эксплуатации?

- Какие выдвинуты требования к качеству (безопасность, производительность, возможность параллельной обработки, интернационализация и конфигурация)?
- Каким образом спроектировать программное обеспечения с учетом требований гибкости и удоства обслуживания?
- Существуют ли обстоятельства, которые смогут влиять на архитектуру программного обеспечения сейчас или после его развертывания?

Высокоуровневая архитектура программного обеспечения имеет следующие цели:

- Раскрывать структуру системы, но скрывать детали реализации ее составных частей.
- Реализовывать все варианты использования и сценарии.
- По возможности отвечать всем требованиям различных заинтересованных сторон.
- Удовлетворять требованиям не только по функциональности, но и по качеству.

В процессе проектирования программной архитектуры важными являются следующие вопросы:

- Какие части архитектуры являются фундаментальными, изменение которых в случае неверной реализации представляет наибольшие риски?
- Какие части архитектуры вероятнее всего подвергнуться изменениям, а также проектирование каких частей можно отложить?
- Какие условия могут привести к изменению спроектированной архитектуры?

Концептульная архитектра (conceptual architecture) – идейное представление об архитектуре программного приложения.

Архитектурный стиль (reference architecture, шаблонная архитектура) – это абстрактная архитектура, применимая к некоторому множеству программных продуктов. Распространенными архитектурными стилями являются следующие архитектурные стили:

1. клиент-серверный;
2. многослойный;
3. многоуровневый;
4. сервис-ориентированный;
5. шина сообщений.

Шаблонные архитектурные решения имеют следующие преимущества:

1. Предоставляют общую терминологию;
2. Являются решением, проверенным на уже существующем программном обеспечении.

Пилотная архитектура (*architectural spike*) – архитектурное решение, которое частично реализуется для проверки его применимости. Если пилотная архитектура доказывается свою эффективность, то пилотную архитектуру принимают в качестве **базовой архитектуры**.

Модуль (*unit of implementation*) – часть реализации программного обеспечения. Обычно модуль состоит из файлов с исходным кодом, но также может включать xml-файлы, текстовые конфиграционные файлы и другие элементы реализации.

Компонент (*unit of runtime and deployment*) тоже является частью программного обеспечения. В отличии от модуля компонент является единицей развертывания и исполнения. Компоненты распространяются в виде файлов, пригодных для исполнения. Компоненты в дальнейшем могут организовываться в более сложную программную систему сторонними пользователями. Не существует ограничений на способ распространения компонентов. Наиболее часто компоненты распространяются в виде исполняемых файлов с расширением *.exe. Однако, в случае интерпретируемых языков компонентов может быть представлен набором файлов, пригодных для обработки интерпретатором. И компоненты, и модули могут распространяться как динамически подключаемые библиотеки (в виде файлов с расширением *.dll). Например, веб-приложения (программные компоненты) ASP.NET, предназначенные для развертывания и исполнения на веб-сервере Microsoft IIS, распространяются в виде dll-файлов.

Сложные программные компоненты состоят из компонентных частей (*component part*). Каждая компонентная часть представлена одним или несколькими файлами (чаще всего динамическими подключаемыми библиотеками или исполняемым файлом).

При создании программного обеспечения один компонент может реализован с использованием одного или нескольких модулей.

Фреймворк (framework) – готовая инфраструктура для создания программных приложений определенного типа. Фреймворк обычно включает готовые архитектурные решения и методы разработки, которых следует придерживаться разработчику, программные утилиты для автоматического создания программного кода, готовые компоненты, реализующие базовую архитектуру. Распространенными фреймворками для создания веб-приложений являются *Microsoft ASP.NET MVC (C#), Yii framework (PHP), flask framework (Python)* и др.

Слабая связность (*loose coupling*) – принцип, при котором компоненты программного обеспечения используют минимальное количество связей с друг другом.

Сквозная функциональность (*crosscutting concern*) – логика программного приложения, обращение к которой многократно в исходном коде. К сквозной функциональности относят механизмы протоколирования, авторизации и аутентификации, кэширование, связь, обработка исключений, проверка корректности данных, введенных пользователем.

Основные принципы качественно спроектированной программной архитектуры:

- Разделение функций. По возможности программное обеспечение разделяется на отдельные составные части (модули и компоненты) с минимальным перекрытием функциональности. Важным фактором является предельное уменьшение количества точек соприкосновения, что обеспечит высокую связность (*high cohesion*) и слабую связанность (*low coupling*). Неверное разграничение функциональности может привести к высокой связанности и сложностям взаимодействия, даже несмотря на слабое перекрытие функциональности отдельных компонентов.
- Принцип основной ответственности. Каждый отдельно взятый компонент или модуль должен отвечать за реализацию одного свойства или функциональной возможности (группы связанных функций).
- Принцип минимального знания. Составные элементы программного обеспечения (объекты, компоненты и др.) не осведомлены о внутренней реализации других элементов.
- Отсутствие повторений. Функциальность должна быть реализована только в одном модуле или компоненте. Дублирование реализации не допускается.
- Проектирование наперед осуществляется только при необходимости. Если требования к программному обеспечению могут быть изменены или не определены однозначно, то следует проектировать только то, что необходимо в ближайшей перспективе. Обычно масштабное предварительное проектирование и тестирование архитектрных решений осуществляется в тех случаях, если неверная архитектура несет значительные издержки.

1.2. Роль архитектора программного обеспечения

За проектирование программного обеспечения отвечает человек, занимающий должность архитектора. Архитектор программного обеспечения является техническим лидером коллектива разработчиков. Однако, технические знания архитектора отличаются скорее не глубиной понимания, а широтой кругозора. В отличии от менеджера проекта, который больше озабочен сроками, стоимостью и доступными ресурсами, архитектор имеет преимущество в принятии технических решений. Однако, архитектор также задействован в планировании в распределении задач по частям реализуемой архитектуры.

В профессиональных компетенциях проектировщикам можно выделить следующие особенности:

- Знание цикла разработки программного продукта. Для архитектора является важным понимание ролей и обязанностей участников проекта ввиду постоянного вовлечения в процесс разработки. Часто архитектор указывает и объясняет, что и как должно быть сделано. Тем самым его роль значительно пересекается с ролью менеджера проекта.
- Знание предметной области. Позволяет архитектору предвидеть возможные изменения требований, а также выявить скрытые требования.
- Знание технологий. Архитектор преимущественно должен быть осведомлен о существенных деталях различных информационных технологий.
- Умение программировать. Владение данным навыком позволяет успешно взаимодействовать с разработчиками и ценить результаты их труда. Отсутствие опыта программирования может стать причиной коммуникационного барьера.
- Коммуникационные навыки. Данное умение позволяет приходить к соглашению со всеми заинтересованными сторонами, мотивировать разработчиков и в конечном итоге сделать архитектуру общим знанием среди команды разработчиков.
- Опыт принятия решений.
- Опыт ведение переговоров. Архитектор должен уметь находить компромиссы при создании архитектуры программного обеспечения между заинтересованными сторонами. Задача разработчика и

архитектора решения – найти равновесие между (зачастую противоречащими друг другу) требованиями и ограничениями, налагаемыми бизнесом, конечным пользователем, информационной инфраструктурой и инфраструктурой управления организации, экономической ситуацией, технологиями и инструментальными средствами.

Следует различать понятия роли архитектора и физического лица. Часто роль архитектора в команде разработчиков может быть переменно выполняться разными участниками.

В больших компаниях среди проектировщиков можно выделить следующие должности:

- Главный проектировщик (*chief architect).* Данная руководящая должность (*senior position*) включает обязанности по управлению проектировщиками в рамках предприятия. Главный проектировщик управляет не только отдельным продуктом, а линейкой программных продуктов. Поэтому данная позиция требует знаний маркетинга, экономики и других областей помимо технических знаний.
- Архитектор/проектировщик программного продукта (*Product/Technical/Solution Architect*). Данный инженер отвечает за проектирование отдельного программного продукта.
- Корпоративный проектировщик (*Enterprise Architect*). Данная должность включает обязанности понять, документировать и спроектировать основные программные системы для предприятия с точки зрения бизнеса, а не программной инженерии.

1.3. Процесс проектирования в процессе разработки программного обеспечения

Проектированию программного продукта предшествует процесс выявления требований к программному продукту и их анализа. После завершения фазы проектирования происходит реализация проекта – кодирование и тестирование.

Проектирование является одной из этапов создания программного приложения или системы программных приложений, которая присутствует в большинстве методологий разработки программного обеспечения. Длительность и значимость процесса проектирования варьируется в зависимости от методологий. В легковесных процессах разработки программного степень формализации стадии проектирования низка, как и количество создаваемой документации. Некоторые проекты предпола-

гают больший объем работ по проектированию именно на стадии конструирования; другие проекты явно выделяют проектную деятельность в форме фазы проектирования [2].

Независимо от задачи проектирования, в этом процессе представляется возможным выделить следующие составляющие:

1. Изучение проблемы.
2. Предварительно определить несколько или как минимум одно решение.
3. Выявить и описать составные элементы каждого решения.

Стоит отметить, что в процессе проектирования проект разрабатывается постепенно: от неформального к более формальному.

Обычно исходными данными являются варианты использования и сценарии поведения пользователя, функциональные требования, нефункциональные требования (включая параметры качества, такие как производительность, безопасность, надежность и другие), технологические требования, целевая среда развертывания и другие ограничения [3].

1.4. Проектирование и характеристики качества программного обеспечения

Высокие характеристики сопровождаемости являются одной из целей проектирования. С увеличением сложности программных систем увеличивается продолжительность жизненного цикла. Часто участники коллектива, создавшего программный продукт, не участвуют в процессе сопровождения или не могут предоставить и вспомнить некоторые делали реализации. Документация может быть устаревшей или утерянной. Сопровождаемость определяется тем как легко и быстро можно обновить программное обеспечение.

Гибкость и способность к расширению также являются желанными характеристиками проектируемой системы. Примером сценария, при котором данные характеристики имеют значение, является необходимость изменить систему для развертывания в облачной среде исполнения вместо обычного сервера.

Параллельная разработка может быть одной из целей при проектировании. Обычно отдельные компоненты реализуются небольшими командами разработчиков. Несмотря на то, что время разработки может существенно сократиться, появляется необходимость удостовериться, что созданные компоненты способны работать в составе системы.

Отложенное связывание компонентов программной системы учитывается при проектировании. Например, программное приложение может

поддерживать работу с несколькими реляционными базами данных. Путем декларативной конфигурации указывается модуль, который будет использован при выполнении операций с данными. Отложенное связывание позволяет проводить настройку программной системы без перекомпиляции.

Для достижения показателей производительности уделяется внимание параллелизму, кэшированию, частоте обращения к данным и коммуникации между процессами, выявляются возможные узкие места архитектуры [2].

При разработке проекта программного обеспечения могут учитываться следующие приоритеты:

- Быстрая и наиболее простая реализация;
- Наиболее удобное и простое сопровождение и портирование;
- Наибольшая надежность;
- Наибольшая производительность.

При проектировании могут соблюдаться следующие принципы:

- Иерархичность.
- Модульность.
- Независимость.

В процессе проектирования программного приложения не только создается архитектура, но также происходит выбор средств реализации программного продукта: сред разработки и тестирования, технологий и других инструментов. При проектировании программного приложения определяется его тип.

Распространенными типами программных приложений являются следующие:

- Программные приложения, предназначенные для исполнения на мобильных устройствах:
- Насыщенные клиентские приложения для выполнения преимущественно на клиентских ПК.
- Насыщенные клиентские приложения для развертывания из Интернета с поддержкой насыщенных UI и мультимедийных сценариев.
- Сервисы, разработанные для обеспечения связи между слабо связанными компонентами.
- Веб-приложения для выполнения преимущественно на сервере в сценариях с постоянным подключением.
- В зависимости от типа приложения на архитектуру могут накладываться дополнительные ограничения. Например, при реализации

приложений, исполняющихся на мобильных устройствах, предъявляются требования к энергосбережению.

- Приложения и сервисы, размещаемые в центрах обработки данных (ЦОД) и в облаке.

Приложение может развертываться в разнообразнейших средах, каждая из которых будет иметь собственный набор ограничений, таких как физическое распределение компонентов по серверам, ограничение по используемым сетевым протоколам, настройки межсетевых экранов и маршрутизаторов и многое другое. Существует несколько общих схем развертывания, которые описывают преимущества и мотивы применения ряда распределенных и нераспределенных сценариев. При выборе стратегии необходимо найти компромисс между требованиями приложения и соответствующими схемами развертывания, поддерживаемым оборудованием, и ограничениями, налагаемыми средой на варианты развертывания.

1.5. Типы клиентских программных приложений

Мобильные приложения могут разрабатываться как тонкое клиентское или насыщенное клиентское приложение. Основные характеристики этих приложений:

- Поддержка портативных устройств.
- Доступность и простота использования для мобильных пользователей.
- Ограниченные возможности ввода и навигации.
- Ограниченная область отображения экрана.

Насыщенные интернет-приложения (*rich internet applications*, RIA) – это веб-приложения с насыщенным графическим пользовательским интерфейсом, выполняющиеся в браузере. Такие программные приложения реализуются с использованием технологий *Flash*, *Silverlight* или средств HTML5 *canvas* и SVG [1]. Основные характеристики данного типа программных приложений:

- низкое время отклика;
- возможность использования потового мультимедиа;
- возможность использовать вычислительные мощности клиентского компьютера

- возможность использовать оборудование клиентского устройства (веб-камера, микрофон и др.).

Обычные веб-приложения поддерживают сценарии с постоянным подключением и могут поддерживать множество разных браузеров, выполняющихся под управлением множества различных операционных систем и на разных платформах. Веб-приложение применяется в случаях, если интерфейса пользователя должен быть стандартизованным, доступным для широчайшего диапазона устройств и платформ и работать только при постоянном подключении к сети. Также Веб-приложения хорошо подходят, если необходимо обеспечить доступность содержимого приложения для поиска средствами интернет-поиска. Основные характеристики веб-приложений:

- Широко доступный и основанный на стандартах интерфейс пользователя.
- Простота развертывания и внесения изменений.
- Необходимость устойчивого сетевого подключения.
- Сложно обеспечить насыщенный пользовательский интерфейс.

Консольные приложения предлагают альтернативный текстовый пользовательский интерфейс и обычно выполняются в командных оболочках. Такие приложения лучше всего подходят для задач администрирования или разработки и не используются как часть многослойного программного приложения.

Оконные приложения с графическим интерфейсом пользователя удобны благодаря возможности автономной работы без подключения к сети Интернет. Помимо этого, данные приложения имеют доступ ко всему аппаратному оборудованию. Разработчик имеет возможность использовать как стандартные графические элементы управления, так и разработанные самостоятельно.

1.6. Задание

Исследуйте сайты с вакансиями архитекторов программного обеспечения. Выясните, какие требования к квалификации, опыту работы и образования указаны.

Возможно использовать следующие сайты:

https://www.linkedin.com

http://www.technojobs.co.uk
http://jobs.monster.com

1.7. Контрольные вопросы

1. Какие типы должностей существуют для роли архитектора? Как отличаются требования к квалификации и должностные обязанности этих позиций?
2. В чем разница между микроархитектурой и макроархитектурой?
3. Какие существуют типы программных приложений? Каковы их преимущества и недостатки?
4. Каковы преимущества использования насыщенных интернет-приложений? Какое оборудование клиентского устройства доступно для использования таким приложениям?
5. Какие существуют способы развертывания программных приложений?
6. Как характеристики качества программного обеспечения влияют на процесс проектирования?

2. Документирование архитектуры программного обеспечения

2.1. Общие сведения о процессе документирования

В небольших коллективах разработчиков процессу документирования архитектуры не уделяется должного внимания. Обычно детали архитектуры передаются в устной форме. Для обсуждения архитектур используются маркерные доски. Проблема документирования архитектур программных систем возникает с необходимостью объяснять архитектуру все большей и большей аудитории. При документировании архитектуры часто требуют ответа следующие вопросы:

- Что именно нужно документировать?
- Каким образом создавать документацию? Какой шаблон документа использовать для описания архитектуры?
- Какие нотации использовать?
- Насколько детальным должно быть описание?

Цели документирования:

1. Образовательная. С помощью документации участники проекта имеют возможность ознакомиться с продуктом. Часто документацией пользуются новые участники проекта: архитекторы, аналитики, разработчики и др.
2. Средство коммуникации между заинтересованными сторонами. Участники проекта общаются между собой используя элементы документации. Специалисты по сопровождению определяют стоимость внесения изменений, тестировщики и интеграторы определяют взаимосвязанность компонентов системы.
3. Основа системного анализа. В нем документация выступает в роли входных данных.

Этапы создания документации:

1. Определение потребностей заинтересованных сторон. Если заинтересованные стороны и их потребности не выявлены, то возможно создание документации, которая будет бесполезна для всех участников проекта.

2. Сбор и документирование архитектурной информации в виде группы представлений и специального блока с общей информацией для всех представлений.

3. Проверка того, что созданная документация удовлетворяет требованиям заинтересованных сторон.

4. Подготовка архитектурной документации в вид, пригодный для той или иной заинтересованной стороны.

Правила документирования программной архитектуры:

1. Документация должна создаваться с точки зрения ее читателя.
2. Отсутствие повторений.
3. Отсутствие двусмысленности.
4. Объяснение нотаций, сокращений и прочих обозначений.
5. Следование стандартам организации и оформления.
6. Поддержка актуальности. В процессе проектирования часто вносятся изменения в документацию.

Созданная документация может быть прямым указанием для кодировщиков, спецификацией для автоматической кодогенерации, входными данными для планирования проекта.

Распространенными нотациями для документирования программной архитектуры являются следующие:

1. DFD (*Data Flow Diagram*) – диаграмма потоков данных.
2. Диаграммы, используемые в методологиях группы IDEF. Группа методологий IDEF используется для проектирования информационных систем.
3. UML-диаграммы.

Диаграммы DFD являются средством представления связей процессов обработки данных и внешних объектов через обмен данными между ними. Часто DFD-диаграммы используются в качестве средства предварительного обзора архитектуры программной системы.

В объектно-ориентированном подходе к проектированию и реализации программного обеспечения основной нотацией являеются диаграммы UML.

Требования к программному обеспечению могут быть задокументированы в техническом задании (ГОСТ) или документе Software Requirements Specification (стандарт IEEE). Проект программного обеспечения может быть описан с использованием следующих стандартов:

- ГОСТ 19.402-78. ЕСПД. Описание программы
- ГОСТ 19.202-78 - Единая система программной документации. Спецификация. Требования к содержанию и оформлению
- 1016-2009 - *IEEE Standard for Information Technology-Systems Design- Software Design Descriptions.*

2.2. Архитектурное представление

Описание архитектуры может состоять из набора представлений, каждое из которых показывает взгляд на программное обеспечения с точки зрения какой-либо заинтересованной стороны, и отдельного описания общих элементов, присутствующих во всех представлениях.

Идея представление архитектуры программного обеспечения в виде 5 представлений, каждое из которых отражает архитектуру программной системы с различных точек зрения, принадлежит Филиппу Крачтену – канадскому специалисту по разработке программного обеспечения.

Представление или точка зрения на архитектуру является основной единицей архитектурной документации программного обеспечения. Представление (*architectural view*) является описанием архитектры, которое пригодно для чтения одной из заинтересованных сторон.

Архитектурная модель 4+1, представленная на рисунке 2.1, состоит из следующих представлений архитектуры программной системы:

1. Варианты использования (сценарии);
2. Логическое представление;
3. Представление разработчика;
4. Процессное представление;
5. Физическое представление.

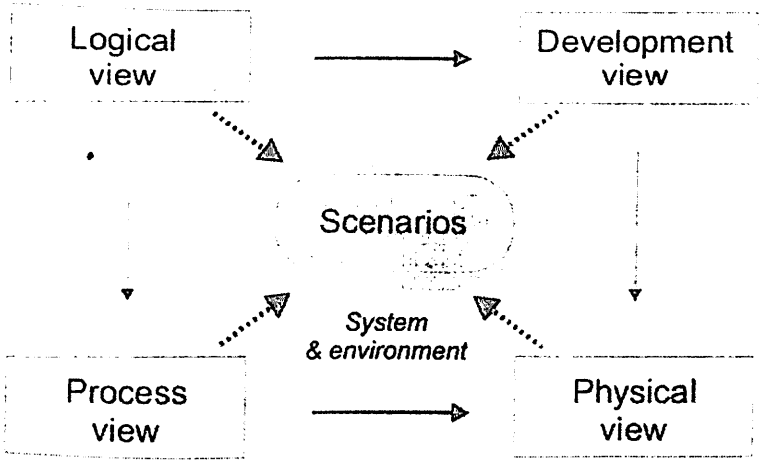

Рис. 2.1 Архитектурная документация 4+1 Design Views

Обычно сначала создается представление в виде вариантов использования (диаграммы *Use case*), отражающее взгляд пользователя на программную систему.

Логическое представление описывает составные части программной системы, показывает внутреннюю структуру компонентов и взаимодействие составные частей программной системы. В логическом представлении используют диаграммы классов, объектов и состояний.

Представление разработчика показывает как составные части программной системы организованы в модули компоненты. Используются *UML Package Diagram* и *UML Component Diagram.*

Физическое представление отображает с помощью *UML Deployment Diagram* как программная система развертывается на вычислительных устройствах.

Процессное представление служит для отображения процессов, происходящих в программной системе. Обычно для этих целей используют *UML диаграммы деятельности, последовательностей и временная диаграмма (UML Timing Diagram).*

Более усовершенствованным способом документирования программной архитектуры является подход Views and Beyond. Данный подход к документированию архитектуры был разработан Институтом программной инженерии Университета Карнеги - Меллон (США, Питсбург). Подход Views and Beyond не только дает рекомендации по созданию документации, но и описывает распространенные архитектурные шаблоны.

В подходе *Views and Beyond* для архитектурного представления предлагается следующая структура:

1. основная презентация (main presentation);
2. каталог элементов и отношений между ними;
3. контекстная диаграмма.

Основная презентация является отправной точкой документирования представления и показывает элементы представления и связи между ними. В основной презентации часто показывают наиболее существенные элементы и скрывают второстепенную информацию, такую как обработка ошибок, логгирование и др. Содержание главного представления зависит от используемой нотации. Часто в главной презентации используют графические нотации, но допустимым является использование таблиц, списков или иных способов представления текста. В случае необходимости главная презентация может содержать более одной диаграммы.

На контекстной диаграмме обычно показывается как часть архитектры, описанная в представлении, соотносится со всей программной системой и окружающей средой.

В подходе Views and Beyond представления делятся на следующие группы:

- Модульные представления (*Module views*) - описывает как программная система представлена в виде значимых частей программного кода.
- Компонентные представления (*Component-and-connector views*) - показывает программную систему с точки зрения взаимодействующих друг с другом элементов в процессе выполнения.
- Представления развертывания и распределения (*Allocation views*) - показывает в программной системе используются аппаратные средства.

Архитектурный документ содержит набор представлений, распределенный по упомянутым выше группам, и документацию, общую для всех представлений.

Общая документация для всех представлений содержит:

– Введение в документ, позволяющее познакомить читателя с его целями и структурой и помочь быстрее найти необходимую информацию

– Общее описание программной системы и связей между представлениями

– Обоснование и ограничения архитектуры

– Рекомендации по сопровождению документации.

2.3. Модульные представления программной архитектуры

2.3.1. Виды модульных представлений программной архитектуры

Разделение реализации программного приложение на модули является одним из распространраненных и очевидных стилей проектирования. Реализация может быть разделена на множество модулей и подмодулей. В качестве критериев деления на модули предлагаются следующие правила:

• Необходимость достижения определенных атрибутов качества программного обеспечения, включая модифицируемость. Для достижения удобства модифицируемости наиболее часто изменяемую логику лучше вынести в отдельные модули.

• Разделение обязанностей среди специалистов. Модульность позволяет вести параллельную разработку.

• Реализации семейства программных продуктов. Правильное определение общих модулей для линейки продуктов позволяет ускорить разработку и снизить затраты.

Наиболее распространенными модульными представлениями являются следующие:

- декомпозиционное представление (модуль-подмодуль);
- представления функциональных связей (зависимостей) между модулями;
- иерархическое представление;
- многослойное представление, которое накладывает ограничения на отношение использования между модулями;
- аспектное представление, показывающее реализацию сквозной функциональности;
- модель данных.

2.3.2. Декомпозиционное предсталение модульной архитектуры и представление зависимостей

В декомпозиционнном представлении основным отношением является отношение "модуль-подмодуль".

При проектировании модулей следует учитывать следующие особенности:

1. Для достижения высоких показателей модифицируемости (*modifiability*) наиболее часто изменяемые части программной системы следует выносить в отдельные модули.
2. При разбиение программного приложения на модули следует учитывать возможность приобретения уже разработанных модулей или использования модулей с открытым исходным кодом.
3. Для эффективной разработки семейства продуктов некоторые модули разрабатываются с учетом их повторного использования в других проектах.
4. Правильное разбиение на модули позволяет команде разработчиков реализовывать модули параллельно.

В декомпозиционном представлении используются диаграммы *UML Package Diagram*. Дополнительная информация о модулях приводится в виде поясняющего тектста. Также могут быть использованы стереотипы для указания вида модуля (рис. 2.2 и рис. 2.3).

В декомпозиционном представлении модули состоят в отношении включения (отношении модуль-подмодуль).

Представление зависимостей между модулями изображается в виде таблицы (таблица 2.1) или с использованием нотации диаграммы пакетов и отношения зависимости (рис. 2.4).

Таблица 2.1

Зависимости между модулям

Использующие модули / Используемые модули	WebUI	Services	Payment	DataModel
WebUI	0	0	0	0
Services	1	0	0	0
Payment	1	0	0	0
DataModel	1	1	0	0

Часто декомпозиционное представление и представление зависимостей совмещаются на одной UML диаграмме пакетов.

2.3.3. Программный модуль «Модель предметной области»

Распространенным архитектрным решением является реализация программного модуля «Модель предметной области». Данный модуль содержит набор классов, интерфейсов, структур и других типов данных, реализующий модель предметной области, в которой используется программное обеспечение.

Архитектура программного модуля документи- руется с помощью UML диаграмм классов, последовательностей, состо- яния. Для того, чтобы показать контекст (окружение) рассматриваемого

модуля, используется UML диаграмма пакетов. На рисунке 2.5 рассматриваемый модуль выделен желтым цветом.

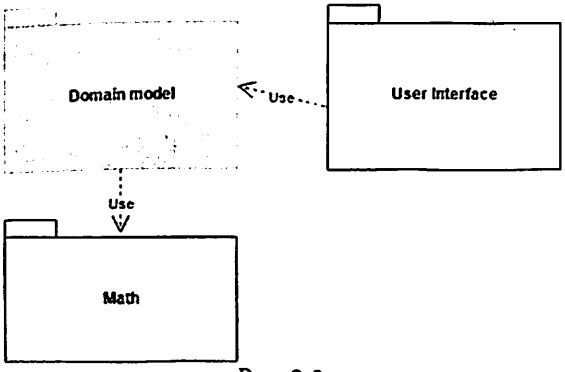

Рис. 2.5

2.3.3. Программный модуль «Сквозная функциональность»

Программный модуль «Сквозная функциональность» содержит программный код, реализующие логику, используемую повсеместно в других модулях программного обеспечения.

Аспектно-ориентированное программирование — парадигма программирования, основанная на идее выделения сквозной функциональности в отдельные модули.

При реализации крупных программных систем в какой-то момент разработчики понимают, что программный код содержит множество мелких, но не относящихся к решаемым задачам деталей. Хорошими примерами таких деталей является транзакционность и проверка прав на выполнение того или иного действия.

В аспектно-ориентированном программировании вводятся следующие элементы[5]:

1. Аспект (*aspect*) — модуль или класс, реализующий сквозную функциональность. Аспект изменяет поведение остального кода, применяя совет в точках соединения, определённых некоторым срезом.
2. Совет (*advice*) — средство оформления кода, которое должно быть вызвано из точки соединения. Совет может быть выполнен до, после или вместо точки соединения.

3. Точка соединения - строка кода программы, где следует применить совет. Многие реализации аспектно-ориентированного программирования позволяют использовать вызовы методов и обращения к полям объекта в качестве точек соединения.
4. Срез (*pointcut*) — набор точек соединения. Срез определяет, подходит ли данная точка соединения к данному совету.

Основными советами (способами оформления сквозной функциональности) является использование аннотаций или атрибутов.

Пример использования атриьутов для добавления сквозной функциональности:

```
        [ExceptionAspect]
        [LogAspect]
        static void Calc()
        {
            throw new DivideByZeroException("A Math Error
Occurred...");
        }
```

2.3.4. Многослойный архитектурный стиль

Многослойный стиль является частным случаем модульного стиля. В многослойном стиле накладывается ограничение на использования отношения зависимости между модулями. В данном стиле модулям более высокого уровня допускается обращаться к модулям низкого уровня. Зависимости в обратном направлении недопустимы.

Для изображения мно- гослойного стиля часто используют неформальную нотацию, нопоминя- ющую стек. В нестрогом многослойном стиле допускается обращение к ко всем слоям, находящимся на более низких уровнях иерарархии (рис. 2.6).

В строгом многослойном стиле каждый слой может зависеть только от ближайшего, находящегося на урвоень ниже (рис. 2.7).

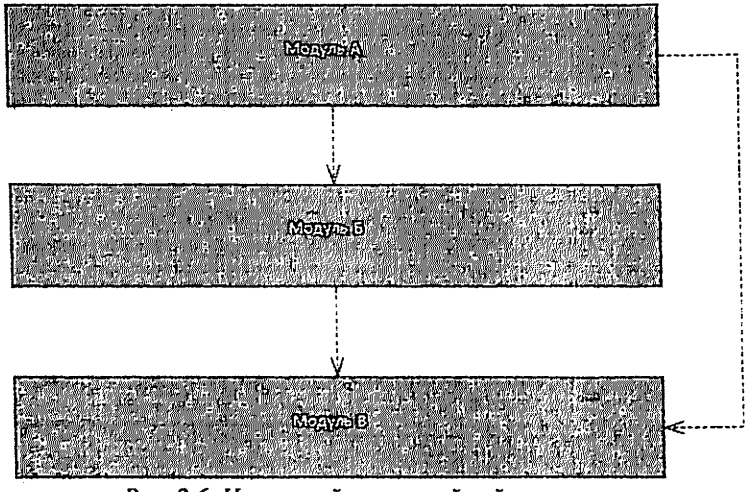

Рис. 2.6. Нестрогий многослойный стиль

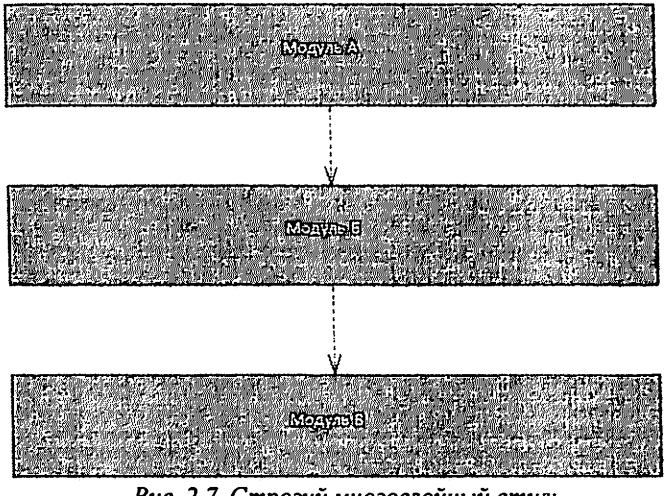

Рис. 2.7. Строгий многослойный стиль

Модульное представление программного приложения в нотации UML диаграммы пакетов приведено на рисунке 2.8.

Рис. 2.8. Многослойный стиль в нотации диаграммы пакетов

В многослойной архитектуре всегда присутствуют как минимум два слоя, но в большинстве случаев термин "Многослойный архитектурый стиль" употребляют в случае наличия трех или более слоев. Наиболее частый способ деления клиентского программного приложения приведен ниже:

- Слой представления. Данный слой содержит ориентированную на пользователя функциональность, которая отвечает за реализацию взаимодействием пользователя с системой, и, как правило, включает компоненты, обеспечивающие общую связь с основной бизнес-логикой, инкапсулириванной в бизнес-слое.
- Слой бизнес-логики. Этот слой реализует основную функциональность системы и инкапсулирует связанную с ней бизнес-логику. Обычно он состоит из компонентов, некоторые из которых предоставляют интерфейсы сервисов, доступные для использования другими участниками взаимодействия.
- Слой доступа к данным. Этот слой обеспечивает доступ к данным, хранящимся в рамках системы, и данным, предоставляемым другими сетевыми системами. Доступ может осуществляться через сервисы. Слой данных предоставляет универсальные интерфейсы, которые могут использоваться компонентами бизнес-слоя.

Слой представления обычно включает следующие компоненты:

- Компоненты пользовательского интерфейса. Это визуальные элементы приложения, используемые для отображения данных пользователю и приема пользовательского ввода.
- Компоненты логики представления. Логика представления – это код приложения, определяющий поведение и структуру приложения и не зависящий от конкретной реализации пользовательского интерфейса. При реализации шаблона Separated Presentation могут использоваться следующие компоненты логики представления: Презентатор (Presenter), Модель презентации (Presentation Model) и Модель Представления (View Model). Слой представления также может включать компоненты Модели слоя представления (Presentation Layer Model), которые инкапсулируют данные бизнес-слоя, или компоненты Сущности представления (Presentation Entity), которые инкапсулируют бизнес-логику и данные в форме, удобной для использования слоем представления.

Слой доступа к данным должен отвечать требованиям приложения,

работать эффективно и безопасно и обеспечивать простоту обслуживания и расширения в случае изменения бизнес-требований. При проектировании слоя доступа к данным следует руководствоваться следующими общими рекомендациями:

- Правильный выбор технологии доступа к данным. Выбор технологии доступа к данным зависит от типа данных, с которыми придется работать, и того, как предполагается обрабатывать данные в приложении. Для каждого сценария существуют наиболее подходящие технологии.
- Использование интерфейса доступа к данным. Этот подход можно реализовать путем определения интерфейсных компонентов, таких как шлюз с общеизвестными входными и выходными параметрами, который преобразует запросы в формат, понятный компонентам слоя. Кроме того, с помощью интерфейсных типов или абстрактных базовых классов можно определить совместно используемую абстракцию, которая должна быть реализована интерфейсными компонентами.
- Инкапсулирование функциональности доступа к хранилищу данных в слое доступа к данным. Слой доступа к данным должен скрывать детали доступа к источнику данных. Он должен обеспечивать управление подключениями, формирование запросов и сопоставление сущностей приложения со структурами источника данных.
- Сопоставление сущностей приложения со структурами источника данных. Тип сущности, используемой в приложении, является основным фактором при принятии решения о методе сопоставления этих сущностей со структурами источника данных. Обычно для этого используются шаблоны *Domain Model* или *Table Module* либо механизмы Объектно-реляционного сопоставления (*Object/Relational Mapping, O/RM*)
- Сократите количество сетевых вызовов и обращений к базе данных. Рассмотрите возможность группировки команд в одну операцию базы данных.
- Требования производительности и масштабируемости. Для слоя доступа к данным требования масштабируемости и производительности должны учитываться во время проектирования. Например, для приложения электронной коммерции именно производительность слоя доступа к данным, скорее всего, будет «узким местом» приложения. Если производительность слоя доступа к данным критична, используйте инструменты профилирования, чтобы понять и затем

сократить количество или разбить ресурсоемкие операции с данными.

2.3.5. Интерфейсные связи между программными модулями

Модульная архитектура программного обеспечения, в которой модули связаны посредством программных интерфейсов, обладает большей гибкостью. Отдельные модули программного компонента можно заменять новыми для адаптации к новым условиям использования, исправления ошибок или добавления новых функций. На рисунке 2.3.4 представлена архитектура использования модулей через интерфейсы. Модули А и Б зависят от модуля «Интерфейс взаимодействия». Обращение к функциональности модуля Б из модуля А происходит исключительно через интерфейс, определенный в модуле «Интерфейс взаимодействия».

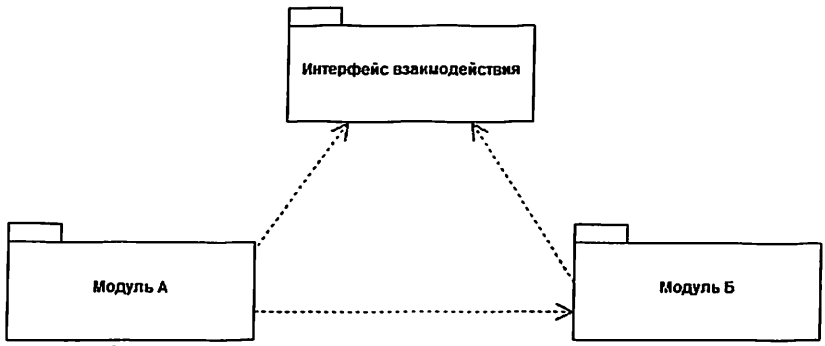

Рис. 2.3. – Взаимодействие программных модулей через интерфейс

Диаграмме пакетов на рисунке 2.3 соответствует диаграмма классов, представленная на рисунке 2.3.1.

Таблица 1. Описание элементов диаграммы классов

Элемент	Назначение	Модуль
Consumer	Использование функциональности динамически подключаемого модуля	Модуль А
ModuleLoader	Загрузка динамически подключаемого модуля и поиск классов, реализующий интерфейс IService	

IService	Определения интерфейса взаимодействия	Интерфейс взаимодействия
Service	Реализация интерфейса IService	Модуль Б

2.3.6. Шаблон модульной архитектуры «Микроядро»

Микроядро (*microkernel*) является шаблоном архитектуры подключаемых модулей (*plugins*). Данный шаблон модульной архитектуры программного обеспечения получил широкое распространения при реализации операционных систем, но также широко используется и для реализации прикладного программного обеспечения.

2.4. Компонентные представления программной архитектуры

Группа представлений компонентов и связей служит для описания программной системы во время ее выполнения. Применительно к данной группе представлений компонентами являются процессы, клиенты, серверы, хранилища данных. Связями являются протоколы, потоки информации, линии связи, доступ к общему хранилищу. В процессе создания этих представлений используются UML диаграммы компонентов и свободные нотации.

В отличии от диаграммы развертывания, на которой также присутствуют компоненты, UML диаграмма компонентов показывает связи между компонентами, а не физическими узлами. Основным отношением на диаграмме компонентов является отношение зависимости. Отношение зависимости может быть указано как с интерфейсом (рис. 2.9), так и без него (рис. 2.10).

Таблица 2.2

Нотация UML диаграммы компонентов

Номер	Элемент	Описание
1	Компонент	Допускающий повторное использование функциональный элемент системы. Компонент предоставляет и потребляет поведение через интерфейсы и может использовать другие компоненты.
2	Часть компонента	Атрибут компонента, тип которого, как правило, является другим компонентом. Часть используется при внутреннем проектировании ее родительского компонента. Графически части изображаются вложенными в родительский компонент.
3	Зависимость	Может использоваться для указания, что требуемый интерфейс одного компонента может соответствовать предоставленному интерфейсу другого компонента. Зависимости также можно использовать в более общем случае при работе с элементами модели, чтобы показать, что конструкция одного зависит от конструкции другого.
4	Делегирование	Связывает порт с интерфейсом одной из частей компонента. Указывает, что сообщения, отправленные компоненту, обрабатываются этой частью, или что сообщения, отправленные этой частью, отсылаются из родительского компонента.
5	Предоставленный порт интерфейса	Представляет группу сообщений или вызовов, реализуемых компонентом и доступных для использования другими компонентами или внешними системами. Порт — это свойство компонента, имеющее в качестве типа интерфейс.

6	Требуемый порт интерфейса	Представляет группу сообщений или вызовов, отправляемых компонентом другим компонентам или внешним системам. Компонент предназначен для соединения с компонентами, которые предоставляют хотя бы эти операции. Порт имеет в качестве типа интерфейс.

Внутреннее устройство программных компонентов может быть детализировано с помощью компонентных частей (рис. 2.11).

2.5. Архитектурные представления развертывания и распределения

2.5.1. Назначение представлений развертывания и распределения и их виды

Основным назначением архитектурных представлений разветывания и распределения является изображение связей программного обеспечения с внешней средой, в роли которой могут выступать заинтересованные стороны (*stakeholders*), вычислительные устройства, сторонние программные компоненты.

Распространены следующие виды ахитектурных представлений развертывания и распределения:

– представление развертывания - показывает как программные компоненты распределены на аппаратных устройствах;

– представление установки (*install*) - показывает как представлены развернутые программные компоненты в файловой системе;

– представление назначения заданий - показывает соответствие между модулями программной системы и командами разработчиков или отдельными разработчиками.

2.5.1. Архитектурное представление развертывания

Представление развертывания обычно содержит диаграммы UML развертывания (*UML Deployment diagram*). UML диаграмма развертывания служит для изображения связи программного обеспечения (компонентов) с внешним программным (узлы среды исполнения) и аппаратным окружением (физические узлы). Основным элементов UML-диаграммы развертывания является узел. Существует два типа узлов:

1. Физический узел (device node), представляющий собой вычислительное устройство, на котором исполняется программное обеспечение. Физический узел может быть контенером для узла среды исполнения. В тех случаях, когда среду исполнения указывать нецелесообразно, физический узел может выступать контейнером для компонента. Распространенными видами физических узлов являются серверы, персональные компьютеры, мобильные устройства, планшетные компьютеры.
2. Узел среды исполнения (execution environment node). Данный узел может быть контейнером как для другого узла среды исполнения, так и для программного компонента.

На диаграмме развертывания могут быть изображены такие физические узлы, как принтер, видео-камера, электронный измеритель температуры и прочие устройства, если они значимы при развертывании программного обеспечения.

Помимо узлов на диаграмме развертывания (рис. 2.12) также размещают программные компоненты, не указывая отношения между ними.

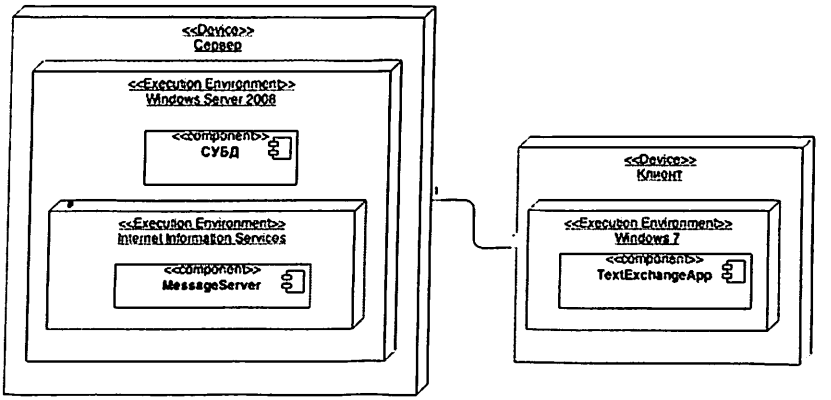

Рис. 2.12. Пример диаграммы развертывания

2.5.1. Архитектурное представление назначения заданий

Другим видом представления распределения и развертывания является представление назначения заданий.

Таблица 1.3

Назначение заданий

Модуль Преобразователь данных		Исполнитель
Сегмент	Подсистема	
Научные вычисления	Нормализация данных	Отдел вычислений А
	Экстраполяция	Отдел вычислений Б
	Решение уравнений	Отдел вычислений А
Графика	2D преобразования	Группа №1 по работе с графическими данными
	3D преобразования	Группа №2 по работе с графическими данными
.....	.....	.....

Документация представления содержите следующее:

- Графическая презентация основных элементов и связей между ними
- Список использованных элементов с описанием их свойств
- Спецификации интерфейсов элементов и их поведения

– Описание способов реализации данной части архитектуры;
– Обоснование принятых архитектурных решений и их описание

2.6. Документация архитектуры и заинтересованные стороны

Потребности в архитектурной документации различаются между заинтересованными сторонами проекта [2]. Менеджеры проекта наиболее обеспокоены расписанием и назначением заданий (рис. 2.13). Для создания плана (расписания) менеджеру необходимы сведения о модулях, сложности реализации каждого из них и зависимости между ними. Поэтому наиболее важными являются следующие представления:

- модульные представления: декомпозиция, зависимости и слои;
- представления распределения: развертывание, назначение заданий;
- прочие: высокоуровневые контекстные диаграммы, показывающие всю систему в целом.

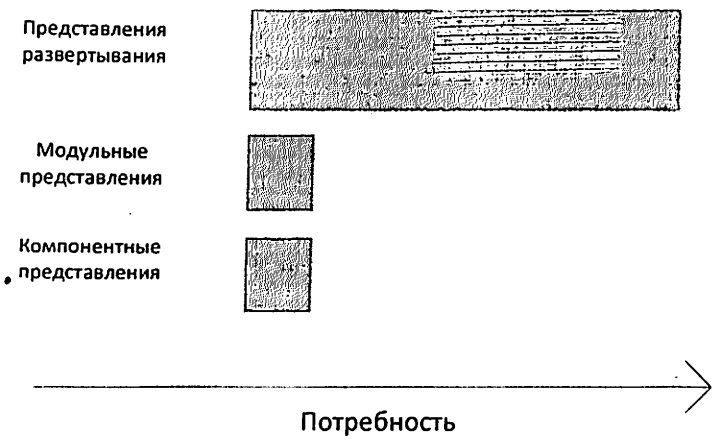

Рис. 2.13. Потребности в документации менеджера проекта

Разработчики (*developers*) и специалисты по сопровождению (*maintainers*) имеют следующие интересы (рис. 2.14):

- основная идея, лежащая в основе архитекты системы;
- сведения о части архитектуры, порученной для реализации.

- подробные сведения о требуемой функциональности и об используемой модели данных;
- сведения об используемых интерфейсах;
- возможность использования уже разработанных элементов.

Разработчикам сторонних программных систем интересны спецификации интерфейсов, информация о поведении системы и модель данных, с которой им предстоит работать. Их интересы схожи со специалистами по сопровождению. Для специалистов по сопровождению изучение архитектурной документации является отправной точкой в их деятельности. Обладание сведениями о зависимостях между модулями и компонентами позволяет оценить масштаб вносимых в программное обеспечение изменений.

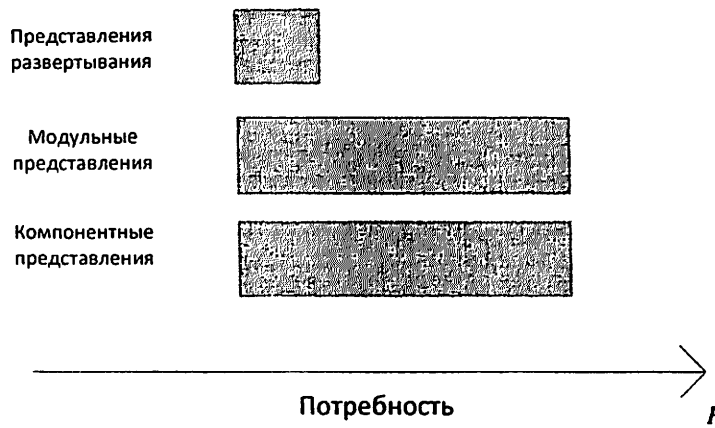

Рисунок 2.14. Использование документации специалистами по сопровождению

Заказчики также являются читателями архитектурной документации, но в основном их интересуют представления развертывания, показывающие как программное обеспечение будет работать в существующем окружении (рис. 2.15).

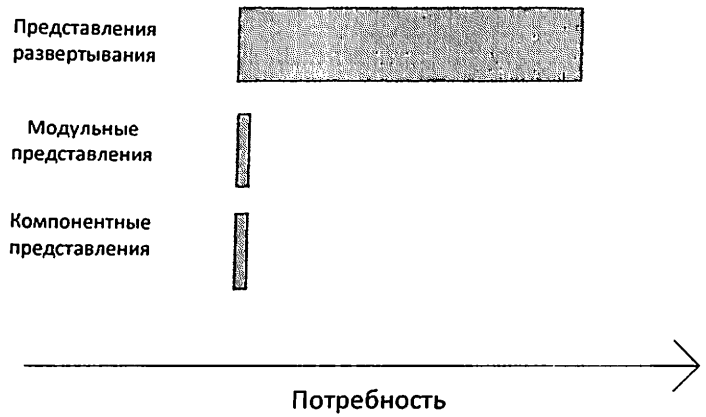

Рис. 2.15. Использование архитектурной документации заказчиком

2.7. Структура архитектурной документации

1. Введение (описание структуры документа, *Documentation Roadmap*) – содержит обзор документа и его аудитории.
2. Краткое описание программного обеспечения (*Overview*) – содержит описание программного обеспечения, функциональных и качественных требований.
3. Представления – данный раздел содержит набор архитектурных представлений.
4. Глоссарий и ссылки (*Directory*) - содержит объяснения использованных сокращений и терминов, а также ссылки на внешние источники.

Представления сложной программной системы могут содержать сотни элементов с различной степенью вложенности. Наиболее очевидным способом организации сложной архитектурной документации является разделение представлений на пакеты и подпакеты подобно тому, как разделяется программный код на пакеты.

2.8. Дополнительные источники

- Adventure Builder – Software Architecture Document. URL: https://wiki.sei.cmu.edu/sad/index.php/Main_Page

2.9. Контрольные задания

1. Постройте UML диаграмму компонентов по следующему описанию:

Программная система состоит из двух компонентов: СУБД MySql и веб-приложения WebClient. Компонент WebClient включает три компонентные части: DataProcessor (предоставляет интерфейс IData, зависит от интерфейсов MySQL_PHP_API и ITask), BackGroundTasks (предоставляет интерфейс ITask), UI (зависит от интерфейса IData)). СУБД MySQL предоставляет интерфейс программирования MySQL_PHP_API.

2. Постройте UML диаграмму развертывания по следующему описанию сценария развертывания программного обеспечения:

Программное приложение «А» для обработки потока видео-данных, получаемых от наружной камеры по протоколу UDP, установлено на аппаратном устройстве с операционной системой JavaOS. Распознанные приложением «А» данные отправляются программному приложению «Клиент» для дальнейшей обработки и долгосрочного хранения. Программное приложение «Клиент» развернуто на сервере с операционной системой Ubuntu. В качестве хранилища данных используется СУБД MongoDB.

3. Приведен следующий программный код. Постройте диаграммы *UML Sequence Diagram* (диаграмма последовательностей) и *UML Class Diagram* (диаграмма классов).

```
class Program
    {
        public static Store st = null;
        static void Main(string[] args)
        {
            st = new Store() { Name = "McNeil" };
            Customer cs = new Customer() { Id = "fdgfgb34r2e", Name = "Nicholas
Taleb" };
            if (cs.Name.Contains(" "))
            {
                st.AddCustomer(cs);
            }
        }
    }
```

```
class Customer
{
    public string Id;
    public string Name;
    public Store St;
    public void Associate(Store s)
    {
        this.St = s;
    }
}
class Store
{
    public string Name;
    public void AddCustomer(Customer c)
    {
        c.Associate(this);
    }
}
```

2.10. Контрольные вопросы

1. Как предлагается создавать архитектурную документацию согласно подходу *Views and beyond*? Как отличаются потребности в архитектурной документации заинтересованных сторон?
2. Что является единицей архитектрной документации? Из каких частей она состоит?
3. Какие виды модульных представлений программной архитектуры существуют? С помобщью каких способов могут быть задокументированы отношения зависимостей между программными модулями?
4. Какие виды компонентных представлений программной архитектуры существуют?
5. Какие виды представлений распределения программной архитектуры существуют?

3. Архитектурные шаблоны

3.1. Общие сведения о микроархитектуре

Микроархитектура программного обеспечения описывает внутреннюю структуру программных компонентов. К микроархитектуре относят модульные архитектурные стили, шаблоны проектирования, архитектурное решение по внедрению зависимостей и другие [1].

3.2. Шаблоны проектирования

Шаблон проектирования (*design pattern*) в разработке программного обеспечения — повторимая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста. Обычно шаблон не является законченным образцом, который может быть прямо преобразован в код; это лишь пример решения задачи, который можно использовать в различных ситуациях. Объектно-ориентированные шаблоны показывают отношения и взаимодействия между классами или объектами, без определения того, какие конечные классы или объекты приложения будут использоваться.

Появление шаблонов проектирования связано с областью строительной архитектуры. В 1970-е годы архитектор Кристофер Александр составил набор шаблонов проектирования. Однако, в области архитектуры эта идея не получила такого развития, как позже в области программной разработки. В 1987 году Кент Бэк и Вард Каннингем взяли идеи Александра и разработали шаблоны применительно к разработке программного обеспечения для разработки графических оболочек на языке Smalltalk. В 1988 году Эрих Гамманачал писать докторскую диссертацию при цюрихском университете об общей переносимости этой методики на разработку программ. В 1989—1991 годах Джеймс Коплин трудился над разработкой идиом для программирования на C++ и опубликовал в 1991 году книгу Advanced C++ Idioms. В этом же году Эрих Гамма заканчивает свою докторскую диссертацию и переезжает в США, где в сотрудничестве с Ричардом Хелмом (Richard Helm), Ральфом Джонсоном (Ralph Johnson) и Джоном Влиссидсом (John Vlissides) публикует книгу *Design Patterns — Elements of Reusable Object-Oriented Software*. В

этой книге описаны 23 шаблона проектирования. Именно эта книга стала причиной роста популярности шаблонов проектирования [6].

Можно выделить следующие категории часто используемых шаблонов проектирования [6]:

1. Порождающие шаблоны проектирования (creational design patterns). Данные шаблоны проектирования решают задачи создания объектов применительно к различным ситауциям. К данной группе относят следующие шаблоны: абстрактной фабрика, строителем, прототип.
2. Структурные шаблоны — шаблоны проектирования, в которых рассматривается вопрос о том, как из классов и объектов образуются более крупные структуры. Шаблоны Адаптер, Мост, Компоновщик, Декоратор, Фасад, Заместитель относят к группе структурных.
3. Поведенческие шаблоны (behavioral patterns) — шаблоны проектирования, определяющие алгоритмы и способы реализации взаимодействия различных объектов и классов. Цепочка ответственности, Команда, Интерпретатор, Итератор, Посредник, Хранитель, Наблюдатель и др.

В шаблоне проектирования Синглентон класс гарантируется, что можно создать только один экземпляр требуемого класса, и предоставляется средство для получения этого экземпляра. Синглентон используется в случаех, когда программному приложению необходим только один объект требуемого типа. Например, ведение отладочной информации, реализация сессий, кэш приложения, менеджер печати, доступ к аппаратному обеспечению и т. д. Нередко он используется вместе с другими шаблонами (Абстрактной фабрикой, Строителем и Прототипом) для обеспечения уникальности их экземпляра.

Пример класса, реализующего шаблон проектирования Синглентон:

```
public sealed class Singleton
{
     private Singleton() { }
     public static Singleton Instance
     {
          get { return InstanceHolder._instance; }
     }
     protected class InstanceHolder
     {
          static InstanceHolder() { }
          internal static readonly Singleton _instance = new
Singleton();
     }
}
```

Альтернативой шаблону проетирования Синглентон является обычный статический класс.

Однако, использование статического класса накладывает следующие ограничения:

- он не может быть наследником других классов и реализовывать интерфейсы;
- нельзя создать экземпляр статического класса. Следовательно, его невозможно использовать в качестве параметра или возвращаемого значения.

Шаблон проектирования Фабрика (рис. 3.1), как и Синглентон, отностся к группе порождающих шаблонов проектирования.

Фабрикой яв- ляется класс, которому делегирована логика по созданию объекта дру- гого класса. Обычно объекты классов создаются напрямую с помощью вызова конструктора класса.

Если для создания объекта класса необхо- димо создать объекты-зависимости (*dependency objects*) или выполнить настройку объекта после создания, то в этом случае применение шаблона проектирования Фабрика является целесообразным.

На рисунке 3.1 класс Client для создания объекта с интерфейсом IProduct обращается к методу CreateProduct класса Factory.

Пример работы с классом Factory приведен в следующем программном коде:

```
public class Client
    {
        public static void Main()
        {

        //.........
           IProduct pr =  Factory.CreateProduct();
        //..........

             }

    }
```

Шаблон Фасад объединяет группу объектов в рамках одного специализированного интерфейса и переадресует вызовы его методов к этим объектам. Шаблон используется если необходимо:

- упростить доступ к сложной системе;
- создать различные уровни доступа к системе;
- уменьшить число зависимостей между системой и клиентом.

Назначение Фасада – создать интерфейс, содержащий методы для решения определённой задачи или предоставляющий определённую абстракцию исходной системы. Фасад позволяет скрыть сложные взаимодействия объектов для решения определенной задачи. Это достигается за счет следующего:

- переадресация вызовов интерфейса шаблона объектам системы;
- уменьшение числа параметров метода подстановкой заранее определенных значений;
- создание новых методов, которые объединяют вызовы объектов системы и/или добавляют свою логику;
- сокрытие части исходных методов и свойств ввиду того, что не играют роли для решения поставленной задачи.

Шаблон Адаптер (рис. 3.2) предназначен для приведения интерфейса объекта к требуемому виду. Данный шаблон применяется в следующих случаях:

- существующий объект, называемый адаптируемым, предоставляет необходимые функции, но не поддерживает нужного интерфейса;
- неизвестно заранее, с каким интерфейсами придется работать адаптируемому объекту;
- формат входных или выходных данных метода не совпадает с требуемым.

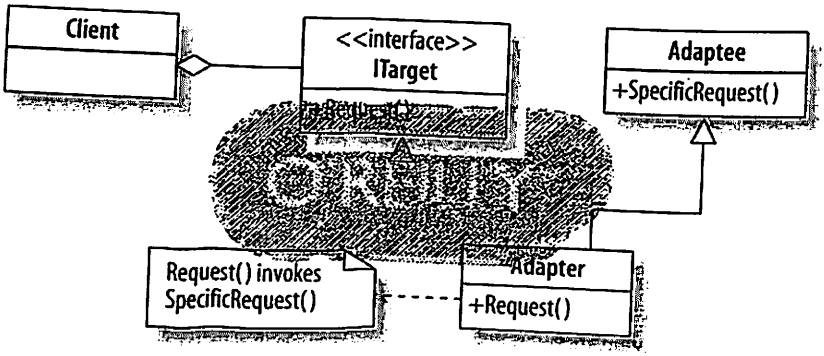

Рис. 3.2. Шаблон проектирования Адаптер

Наблюдатель (*Observer*) — поведенческий шаблон проектирования (рис. 3.3). Создает механизм у класса, который позволяет получать оповещения от других классов об изменении их состояния, тем самым наблюдая за ними. Шаблон проектирования Наблюдатель широко используется в событийной архитектуре оконных приложений с графическим интерфейсом пользователя.

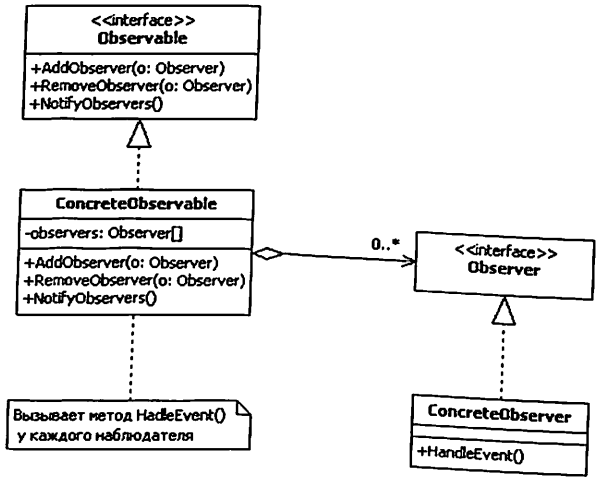

Рис. 3.3. Шаблон проектирования Наблюдатель

3.4. Внедрение зависимости

Внедрение зависимости — процесс предоставления внешней зависимости программному компоненту. Внедрение зависимостей позволяет уменьшить сильное связывание между программными компонентами. Вместо жесткого кодирования зависимостей (например, драйвера какой-либо базы данных), внедряется список сервисов, в которых может нуждаться компонент. После этого сервисы подключаются третьей стороной. Такой подход обеспечивает лучшее управление будущими изменениями и решение проблем в разрабатываемом программном обеспечении [7].

Эти функции называют *сквозной функциональностью* (*crosscutting concerns*), потому что они оказывают влияние на все приложение и, по возможности, должны реализовываться централизованно. Например, если код, формирующий записи журнала и выполняющий запись в журналы приложения, разбросан по разным слоям и уровням, в случае изменения требований, связанных с этими вопросами (например, перенесение журнала в другой каталог), придется искать и обновлять соответствующий код по всему приложению.

- Аутентификация и авторизация. Как правильно выбрать стратегию аутентификации и авторизации, передачи идентификационных данных между слоями и уровнями и хранения удостоверений пользователей.
- Кэширование. Как правильно выбрать технику кэширования, определить данные, подлежащие кэшированию, где кэшировать данные и как выбрать подходящую политику истечения срока действия.
- Связь. Как правильно выбрать протоколы для связи между слоями и уровнями, обеспечения слабого связывания между слоями, осуществления асинхронного обмена данными и передачи конфиденциальных данных.
- Управление конфигурацией. Как выявить данные, которые должны быть настраиваемыми, где и как хранить данные конфигурации, как защищать конфиденциальные данные конфигурации и как обрабатывать их в серверной ферме или кластере.
- Управление исключениями. Как обрабатывать и протоколировать исключения и обеспечивать уведомления в случае необходимости.
- Протоколирование и инструментирование. Как выбрать данные, подлежащие протоколированию, как сделать протоколирование настраиваемым, и как определить необходимый уровень инструментирования.

- Проверка корректности вводимых данных. Как определить, где и как проводить корректность данных; как выбрать методики для проверки длины, диапазона, формата и типа; как предотвратить и отклонить ввод недопустимых значений; как очистить потенциально злонамеренный и опасный ввод; как определить и повторно использовать логику проверки корректности данных на разных слоях и уровнях приложения.

Архитектурный шаблон *Dependency Injection* (рис. 3.4) может быть использован для внедрения экземпляров элементов сквозной функциональности в приложение на основании данных конфигурации. Это позволяет без труда изменять используемые в каждой подсистеме элементы сквозной функциональности без необходимости повторной компиляции и развертывания приложения. Библиотека *Unity* группы разработчиков *Patterns & Practices* обеспечивает полную поддержку шаблона *Dependency Injection*. К другим популярным библиотекам *Dependency Injection* относятся *StructureMap, Ninject u Castle Windsor*.

Основным объектом любой библиотеки, реализующей шаблон внедрения зависимости, является контейнер зависимостей. Контейнер зависимостей является фабрикой, которая создает объекты требуемого типа данных. Контейнер поддерживает два метода:

1. Resolve. Данный метод возвращает объект требуемого типа данных, автоматически разрешая зависимости.
2. Register. Данный метод регистрирует доступные интерфейсы и соответствующие им реализации. В нем так же указываются параметры конструктора. Помимо этого в этои методе также может задаваться начальное состояние объекта, включая значения свойств и полей.

Пример программной конфигурации контейнера зависимостей:

```
UnityContainer c = new UnityContainer();
c.RegisterType<ISender, SMSSender>( new InjectionProp-
erty("Recipient", "+792343523243"));
ISender n = c.Resolve<ISender>();
```

Альтернативой использования метода Register является конфигурирования контейнера зависимостей в XML-файле. Данный подход является более гибким и позволяет менять реализации интерфейсов, а также параметры конструктора и состояние объекта без перекомпиляции программного приложения [8].

Пример конфигурационного файла контейнера зависимости:

```
<unity>
<containers>
      <container name="container">
        <types>
<type type="Abstract.IPaymentProvider, Abstract"
              mapTo="Robokassa.RobokassaUrlManager, Ro-
bokassa">
            <lifetime type="singleton"/>
            <constructor>
              <param name="login">
                <value value="Banokin"/>
              </param>
              <param name="pass1">
                <value value="wydksgd23F"/>
              </param>
              <param name="pass2">
                <value value="nfnnsf65G"/>
              </param>
            </constructor>
          </type>
     </types>
      </container>
    </containers>
  </unity>
```

```
UnityContainer c = new UnityContainer();
UnityConfigurationSection s =     (UnityConfigurationSection)ConfigurationMan-
ager.GetSection("unity");
s.Configure(c);
IPaymentProvider n = c.Resolve<IPaymentProvider>();
```

Контейнеры зависимостей способны создавать объекты типов, которые не указаны в их конфигурации. Это возможно в тех случаях, когда

зависимости требуемого типа данных присутствуют в конфигурации контейнера.

3.5. Проблемно-ориентированное проектирование

Создание схемы данных является важной составляющей проекта программного обеспечения. Для создания схемы данных необходим анализ предметной области. Схема данных содержит перечень сущностей и отношений. Модель данных часто изображается средствами языка UML или набором ER-диаграмм. В процессе проектирования модель данных эволюционирует. Детализированная модель может уточнять способ хранения данных в том или ином хранилище.

Более комплексный подходом к созданию модели данных является проблемно-ориентированное проектирование. Проблемно-ориентированное проектирование (*Domain Driven Design*) — это объектно-ориентированный метод проектирования для создания модели предметной области. В эти модели содержится реализация не только схем объектов, но бизнес-логики. В модели предметной области наиболее точно реализуются реальные условия ведения бизнеса в виде программного кода. В процессе разработки модели предметной области происходит взаимодействие между специалистами в области информационных технологий и экспертами предметной области. При совместной устанавливается так называемый общеупотребительный язык (*ubiquitous language*), в котором значения слов одинаково понятны для представителей разных областей знаний и отсутствует двусмысленность языковых конструкций, исключены все технические жаргонизмы [9].

В проблемно-ориентированном проектировании используются следующие понятия:

1. Ограниченный контекст (*bounded context*). Обычно программные продукты используются в очень обширных предметных областях. Между некоторыми сферами деятельности могут иметь место различные толкования того или иного явления, термина или процесса. Под ограниченным контекстом понимается та часть обширной предметной области, для которой создается модель.
2. Сущность (*entity*) – это объект, который может быть уникально идентифицирован. Сущности имеют идентичность, состояние и поведение (способность изменять свое состояние или состояние других объектов). Обычно сущности реализуются как классы.
3. Объект-значение (*value object*) – это объект, идентичность которого определяется исключительно значениями его атрибутов. Объекты

значения используются для описания свойств сущностей. В программном коде реализуется как неизменяемый объект (immutable object) После создания внесение изменений в объект значения невозможно, а в случае необходимости создается новый объект значения.

Агрегат (agregate) – объект, имеющий сложную внутреннюю иерархическую структуру. Агрегат обладает всеми свойствами сущности, и объект агрегата обрабатывается как сущность.

Корень агрегата (agregate root) – это объект, через который происходит обращение ко всем внут- ренним объектам.

Следует различать агрегаты и обычные списки. Корень агрегата помимо ссылки на коллекцию содержит также методы для управления коллекцией зависимых объектов. Примером агрегата может быть объект Заказ (рис. 3.5), который содержит перечень подчиненных элементов (заказанных позиций).

Сервис (service) – операция, у которой нет идентичности и жизненного цикла (отсутствует состояние). Если логику невозможно отнести ни к одной из сущностей в модели предметной области, то такая логика реализуется в виде сервиса.

Главной функциональностью серви- сов является способность изменить состояние сущностей.

Результат ра- боты сервиса не зависит ни от предыдующих обращений к нему, ни от того, к какому именно экземпляру сервиса произошло обращение. Часто, но не всегда сервисы реализуются как классы со статическими методами. Модель в проблемно-ориентированном проектировании не вклю- чает логику, реализующую хранение данных.

Для реализации данной ло- гики используется шаблон проектирования репозиторий (рис. 3.6) – спе- циальный фасад, скрывающий логику доступа к хранилище данных. Использования репозитория позволяет в случае необходимости менять

реализацию хранения и получения данных, не затрагивая модель предметной области.

Листинг 1. Шаблонный интерфейс репозитория

```
public interface IGenericRepository<T> where T : class
{

    IQueryable<T> GetAll();
    IQueryable<T> FindBy(Expression<Func<T, bool>> pred-
icate);
    void Add(T entity);
    void Delete(T entity);
    void Edit(T entity);
    void Save();
}
```

Листинг 2. Пример реализации интерфейса репозитория

```
public  class GenericRepository<T> :
    IGenericRepository<T>
        where T : class
    {

        private JobContext _entities;
        public GenericRepository(string connectionString)
        {
            _entities = new JobContext(connectionString);
            _entities.Database.Initialize(true);
        }


        public virtual IQueryable<T> GetAll()
        {

            IQueryable<T> query = _entities.Set<T>();
            return query;
        }

       public IQueryable<T> FindBy(System.Linq.Expres-
sions.Expression<Func<T, bool>> predicate)
        {

            IQueryable<T> query = _enti-
ties.Set<T>().Where(predicate);
            return query;
```

```
        }

        public virtual void Add(T entity)
        {
            _entities.Set<T>().Add(entity);
        }

        public virtual void Delete(T entity)
        {
            _entities.Set<T>().Remove(entity);
        }

        public virtual void Edit(T entity)
        {
            _entities.Entry(entity).State = System.Data.En-
tityState.Modified;
        }

        public virtual void Save()
        {
            _entities.SaveChanges();
        }
    }
```

На практике (особенно в небольших проектах) модель предметной области не всегда полностью целесообразно изолировать от используемого хранилища. Класс Rubric, приведенный далее, содержит атрибуты, которые указывают на первичный ключ в таблице:

```
public class Rubric
    {
        [Key]
        public int Id { get; set; }
        [Required(AllowEmptyStrings = false)]
        public string Name { get; set; }
        public DateTime Posted { get; set; }
        [Required(AllowEmptyStrings = false)]
        public string Description { get; set; }

        public string Redirect { get; set; }

        public virtual ICollection<Article> Articles { get;
set; }
}
```

При проектировании модели предметной области следует уделять внимание тем ограничениям, которые накладывают средства реализации программного продукта. Если данные ограничения не учитывать, то в результате проектирования может получиться чисто аналитическая модель, реализовать в программном коде которую будет весьма затруднительно или невозможно. Модель предметной области в программном коде может быть реализована в виде библиотеки классов.

3.6. Компонентные архитектурные стили

Компонентные архитектурные стили описывают шаблонные способы организации программного обеспечения в виде набора программных компонентов.

Наиболее распространенным компонентным архитектурным стилем является стиль Клиент-Сервер. Программная система разделяется на два приложения, где клиент выполняет запросы к серверу. Во многих случаях в роли сервера выступает база данных, а логика приложения представлена процедурами хранения.

Исторически архитектура клиент/сервер представляет собой программное приложение с графическим интерфейсом пользователя, обменивающееся данными с сервером базы данных, на котором в форме хранимых процедур располагается основная часть бизнес-логики, или с выделенным файловым сервером.

Если рассматривать более обобщенно, архитектурный стиль клиент/сервер описывает отношения между клиен- том и несколькими серверами, где клиент инициирует один или более за- просов, ожидает ответы и обрабатывает их при получении.

Обычно сервер авторизует пользователя и затем проводит обработку, необходимую для получения результата.

Для связи с клиентом сервер может использо- вать широкий диапазон протоколов и форматов данных.

Основные преимущества архитектурного стиля клиент/сервер:

- Большая безопасность. Все данные хранятся на сервере, который обычно обеспечивает больший контроль безопасности, чем клиентские компьютеры.
- Централизованный доступ к данным. Поскольку данные хранятся только на сервере, администрирование доступа к данным намного проще, чем в любых других архитектурных стилях.
- Простота обслуживания. Роли и ответственность вычислительной системы распределены между несколькими серверами, общающимися друг с другом по сети. Благодаря этому клиент гарантированно остается неосведомленным и не подверженным влиянию событий, происходящих с сервером (ремонт, обновление либо перемещение).

Многоуровневая и 3-уровневая архитектура являются стилями развертывания, описывающими разделение функциональности на сегменты, во многом аналогично многослойной архитектуре, но в данном случае эти сегменты могут физически размещаться на разных компьютерах, их называют уровнями.

Сервис-ориентированная компонентная архитектура является частным случаем клиент-серверной архитектуры, в которой роль серверного компонента выполняет набор веб-сервисов, а роль клиента – потребитель веб-сервисов.

3.7. Сочетание архитектурных стилей

Архитектура программной системы практически никогда не ограничена лишь одним архитектурным стилем, зачастую она является сочетанием архитектурных стилей, образующих полную систему. Например, может существовать SOA-дизайн, состоящий из сервисов, при разработке которых использовалась многослойная архитектура и объектно-ориентированный архитектурный стиль. Сочетание архитектурных стилей также полезно при построении Интернет Веб-приложений, где

можно достичь эффективного разделения функциональности за счет применения многослойного архитектурного стиля. Таким образом можно отделить логику представления от бизнес-логики и логики доступа к данным. Требования безопасности организации могут обусловливать либо 3-уровневое развертывание приложения, либо развертывание с более чем тремя уровнями.

3.8. Дополнительные источники

1. Developer's Guide to Dependency Injection Using Unity. URL: http://msdn.microsoft.com/en-us/library/dn223671%28v=pandp.30%29.aspx
2. Patterns & practices - Unity - Home. URL: https://unity.codeplex.com
3. Microsoft patterns & practices. URL: http://msdn.microsoft.com/ru-ru/library/ff921345.aspx
4. ASP.NET | The ASP.NET Site URL: http://asp.net
5. Synchfusion E-Books. URL:: https://www.syncfusion.com/resources/techportal/ebooks

3.9. Контрольные задания

Изучите архитектуру MVC на примере инфраструктуры ASP.NET MVC. Используе дополнительные источники, создайте веб-приложение, использующее архитектурные шаблоны Внедрение зависимости и Репозиторий.

3.10. Контрольные вопросы

1. Какие существуют типы клиентских программных приложений?
2. На какие группы можно разделить распространенные шаблоны проектирования?
3. Чем отличается архитектурный стиль от шаблона проектирования?
4. Является ли шаблон проектирования Репозиторий частным случаем шаблона Фасад? Какие операции поддерживает шаблон "репозиторий"?
5. По каким признакам определить, что объект является объектом значения? Как объекты значения реализуются в программном коде?

6. Какое назначение имеет агрегат в проблемно-ориентированном проектировании?
7. Какими основными характеристиками обладает сущность в проблемно-ориентированном проектировании? Как она реализуется в программном коде?
8. Чем отличается многослойная архитектура от многоуровневой?
9. Какой архитектурный стиль реализован в P2P-программных приложениях?
10. Какие ограничения накладывает многослойный архитектурный стиль на связь между модулями? Чем отличается строгий многослойный стиль от нестрогого?
11. Какие задачи решаются с помощью архитектурного шаблона "внедрение зависимости"? Какие существуют способы конфигурации контейнера зависимостей?
12. Что такое сквозная функциональность в программном обеспечении? Какие решения позволяют ее реализовывать?

4. Сервисно-ориентированная архитектура

4.1. Общие сведения об интеграции программных приложений

Приблизительно до 1980 гг. программные приложения для предприятий в большой степени были ориентированы на автоматизацию выполнения повторяющихся заданий.

В последующие вермя появились пони- мание возможности реализации бизнес-процессов посредствомпрограммного обеспечения, и как следствие пониманимание ценности интеграции программных приложений.

Первыми способами интеграции программных приложений были такие очевидные подходы, как исполь- зование общих файлов или использование общей базы данных. Эти спо-собы интеграции остаются актуальными и в настоящее время. Так, про- граммный продукт 1С:Предприятие может развертываться в режим клиент-файл-сервер или клиент-СУБД.

Использование общих файлов требует значительных трудозатрат для реализации проверки целостности данных, разрешения конфликтов одновременной записи данных несколь-кими клиентами. В то же время данный способ обладает важным преиму- ществом - быстрым развертыванием.

Промежуточное программное обес-печение является связывающим компонентом между программными приложениями.

Связывающее программное обеспечение (Middleware) — широко используемый термин, означающий слой или комплекс технологического программного обеспечения для обеспечения взаимодействия между различными приложениями, системами, компонентами.

Рис. 4.1. Интеграция через общие файлы

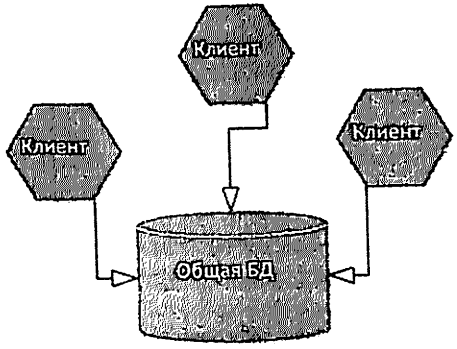

Рис. 4.2. Интеграция через общую БД

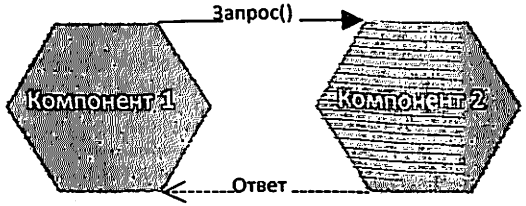

Рис. 4.3. Вызов удаленных процедур

Распространена также следующая классификация способов интеграции программного обеспечения [10]:

- Информационно-ориентированная интеграция основана на обмене информацией между источником и целевым приложением. Она происходит на уровне баз данных или интерфейсов про-граммирования (API).
- Интеграция с использованием порталов. Данный способ интеграции является частным случаем информационно-ориентированной интеграции. Обычно портал реализуется в виде веб-приложения, предоставляющего графический интерфейс пользователя для отображания информации, полученной от других программных компонентов.
- Интеграция, ориентированная на интерфейсы, связывает между собой интерфейсы приложений. Интерфейсы предоставляют услуги (контракты, содержащие перечень поддерживаемых услуг), выполняемые

одним приложением для другого. Интеграция, ориентированная на интерфейсы, не требует визуализации подробных бизнес-процессов, протекающих в приложениях. Поддержка услуг может быть информационной (когда поставляются данные) или операционной (когда выполняется часть функций) . В первом случае интеграция становится разновидностью информационно-ориентированного подхода. Второй способ требует изменений источника и целевых приложений и может привести к созданию нового приложения (составного).

Интеграция, ориентированная на процессы. Данный способ интеграции связывает программные компоненты с помощью определения нового яруса процессов поверх набора процессов в существующих приложениях. Эта интеграция представляет собой решение, в котором логика нового процесса отделена от логики участвующих приложений и возникает новая система, автоматизирующая задачи, которые ранее выполнялись вручную. Интеграция, ориентированная на процессы, начинается с создания модели нового процесса и требует полной видимости внутренних процессов интегрируемых приложений.

4.2. Брокер сообщений

Брокер сообщений является связывающим программным обеспечением. Брокер сообщений является программным компонентом-посредником между клиентом и сервером (веб-сервисом) и ответственен за преобразование форматов сообщений между отправителем и получателем и обеспечение гарантированной доставки сообщений.

4.2. Общие сведения о сервис-ориентированной архитектуре

Одним из ранних подходов было объектное промежуточное программное обеспечение. При использовании объектного промежуточного ПО клиент обращается не отправляет обычный запрос к серверу, а вызывает методы заранее подготовленного объекта. Объектное промежуточное ПО реализуется с помощью технологий DCOM, Microsoft Remoting и др. При использовании данных технологий программисту требуется решить вопросы:

- Как получить ссылку на удаленный объект?
- Когда удаленные объекты создаются и уничтожаются?

– Используется ли удаленный объект монопольно или несколькими клиентами?

Средства создания объектного промежуточного ПО достаточно эффективно решали поставленные задачи, но ввиду разных протоколов и стандартов интеграция оставалось затруднительной.

Сервис-ориентированная архитектура – способ организации программного приложения, при котором его функциональность представляется в виде набора сервисов. Сервис-ориентированная архитектура является одним из шагов для создания промежуточного программного обеспечения (middleware). Сервисно-ориентированная архитектура позволяет создавать распределнное программное обеспечение, состоящее из набора независмых сервисов. Благодаря сервисно-ориентированной архитектуре могут создаваться приложения, являющиеся композицей нескольких сервисов.

Описание веб-сервиса включает его название, расположение и требования к осуществлению процесса обмена данными.

Веб-сервис – программный компонент, идентифицируемый веб-адресом и выполняющие операции, продекларированные в поддерживаемом им интерфейсе. Другие программные компоненты взаимодействуют с веб-сервисами посредством обмена сообщениями.

Веб-сервисы в настоящее время становятся стандартом де-факто для интеграции распределенных систем программных приложений. Создание программных приложений, которые способны использовать веб-сервисы, недостаточно для полной поддержки бизнес-процессов. Организация взаимодействующих друг с другом веб-сервисов в сети позволяет реализовывать сложные рабочие процессы.

- Характеристики сервисов [11]:
  1. Слабая связность. Сервисы обычно независимы друг от друга и обладают минимумом информации о других сервисах.
  2. Автономность. Каждый сервис самодостаточен и способен к независимому функционированию (кроме управляющего сервиса сети веб-сервисов).
  3. Абстракция. Сервис предоставляет клиенту только интерфейс. Внутренние детали реализации скрыты от потребителя.
  4. Отсутствие состояния. Сервис не хранит состояние и не учитывает историю предыдущих обращений к нему.
  5. Композиция. Сервисы спроектированы с возможностью быть объединенными в сеть веб-сервисов. В такой сети веб-сервисов всегда существует управляющий сервис, координирующий работу остальных сервисов.

6. Обнаружение. Сервис должен обладать описанием. После обработки описания существует возможность обратиться к сервису без дополнительной настройки и конфигурации.
7. Отсутствие идентичности. При обращении к разным экземплярам сервиса с одним контрактом результат не зависит от того, к какому именно экземпляру произошло обращение.
8. Обращение через интерфейс. Веб-сервисы поддерживают интерфейсы с определенным перечнем операций

Распространены следующие два типа веб-сервисов:

- SOAP веб-сервисы;
- REST-сервисы.

4.3. SOAP веб-сервисы

Протокол SOAP – протокол обмена структурированными сообщениями в распределённой вычислительной среде.

SOAP-сообщение представляет собой XML-документ; сообщение состоит из трех основных элементов: конверт (SOAP Envelope), заголовок (SOAP Header) и тело (SOAP Body)[12].

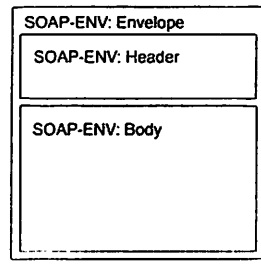

Рис. 4.4. SOAP-конверт

Заголовок – это дочерний элемент конверта. Не обязательный. Заголовок кроме атрибутов `xmlns` может содержать 0 или более стандартных атрибутов:

- `encodingStyle`
- `actor`
- `mustUnderstand`
- `relay`

Пример SOAP запроса:

```
GET /StockPrice
HTTP/1.1
Host: example.org
Content-Type: application/soap+xml;
charset=utf-8
Content-Length: 285
<?xml version="1.0"?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-en-
velope" xmlns:s="http://www.example.org/stock-service">
  <env:Body>
    <s:GetStockQuote>
      <s:TickerSymbol>
        IBM
      </s:TickerSymbol>
    </s:GetStockQuote>
  </env:Body>
</env:Envelope>
```

Элемент `Body` обязательно записывается сразу за элементом `Header`, если он есть в сообщении, или первым в SOAP-сообщении, если заголовок отсутствует. В элемент `Body` можно вложить произвольные элементы, спецификация никак не определяет их структуру. Определен только один стандартный элемент, который может быть в теле сообщения - `Fault`, содержащий сообщение об ошибке.

Протокол SOAP не различает вызов процедуры и ответ на него, а просто определяет формат послания (message) в виде документа XML. Послание может содержать вызов процедуры, ответ на него, запрос на выполнение каких-то других действий или просто текст. Спецификацию SOAP не интересует содержимое послания, она задает только его оформление [13].

SOAP основан на языке XML и расширяет некоторый протокол прикладного уровня — HTTP, FTP, SMTP и т.д. Как правило чаще всего используется HTTP. Вместо использования HTTP для запроса HTML-страницы, которая будет показана в браузере, SOAP отправляет посредством HTTP-запроса XML-сообщение и получает результат в HTTP-отклике. Для правильной обработки XML-сообщения процесс-«слушатель» HTTP (напр. Apache или Microsoft IIS) должен предоставить SOAP-процессор, или, другими словами, должен иметь возможность обрабатывать XML.

4.4. Язык описания веб-сервисов WSDL

Язык описания веб-сервисов WSDL основан на языке XML и служит для хранения следующей информации:

- URL веб-сервиса;

– механизмы коммуникации, корые понимает веб-сервис;
– операции, которые может выполнять веб-сервис;
– структура сообщений веб-сервиса.

Базовая стуктура документа WSDL выглядит следующим образом:

```
<wsdl:description xmlns:wsdl="http://www.w3.org/ns/wsdl">
   <wsdl:documentation>
  <wsdl:types/>
  <wsdl:interface/>
  <wsdl:binding/>
  <wsdl:service/>
</wsdl:description>
```

Корневым элементом документа WSDL является элемент Description.

Элемент types содержит все возможные элементы XML schema, описывающие сообщения веб сервиса. WSDL 2.0 открыта для использования разными системами типов, но практически используется только с XML-схемой.

Элемент interface определяет список операций веб-сервиса, включая описание входных, выходных сообщений и сообщений об ошибках для операций, а также порядка передачи.

Элемент binding определяет, средства коммуникации клиента с веб-сервисом. В случае REST веб-сервисов, в качестве средства коммуникации указывается HTTP.Элемент service ассоциирует адреса веб-сервиса с конкретными интерфейсами (interface) и средствами коммуникации (binding). (Т.е. задает соответствие URL операции веб-сервиса и элементу binding).

Часто WSDL-файлы не создаются разработчиков вручную. Многие средства разработчик сервис-ориентированных приложений содержат средства для их автоматической генерации.

4.5. Пример создания SOAP веб-сервиса

В современных интегрированных средах разработки программных приложений SOAP веб-сервисы создаются с применением шаблонных проектов. Обычно веб-сервис реализуется как класс с несколькими методами. В среде Microsoft Visual Studio для создания веб-сервисов могут быть использованы следующие типы проектов: ASP.NET Web application и WCF application. При реализации веб-сервисов широко используется XML-сериализация и -десериализация объектов. Применение сериализации позволяет автоматически принимать и отправлять данные. В проекте веб-приложения ASP.NET веб-сервисы разрабатываются с применением

технологии ASMX. Технология ASMX обеспечивает более простое решение для создания Веб-сервисов на базе ASP.NET и их предоставления через Веб-сервер IIS.

```
[WebService(Namespace = "http://tempuri.org/")]
    [WebServiceBinding(ConformsTo = WsiProfiles.BasicPro-
file1_1)]
    [System.ComponentModel.ToolboxItem(false)]

    public class ProductService : System.Web.Services.Web-
Service
    {

        [WebMethod]
        public Product CreateProduct()
        {
            Product pr = new Product();
            pr.Name = "Tomato";
            pr.Price = 100;
            return pr;
        }
    }

    public class Product
    {
        public string Name;
        public int Price;
    }
```

Пример SOAP-конверта, содержащего ответ веб-службы, приведен далее:

```
<?xml version="1.0" encoding="utf-8"?>
<soap12:Envelope
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap12="http://www.w3.org/2003/05/soap-envelope">
  <soap12:Body>
    <CreateProductResponse xmlns="http://tempuri.org/">
      <CreateProductResult>
        <Name>string</Name>
        <Price>int</Price>
      </CreateProductResult>
    </CreateProductResponse>
  </soap12:Body>
</soap12:Envelope><?xml version="1.0" encoding="utf-8" ?>
```

4.6. REST веб-сервисы

REST — метод взаимодействия компонентов приложения с использованием протокола HTTP для вызова процедуры. Пот этом необходимые данные передаются в качестве параметров запроса. Этот способ является альтернативой более сложным методам, таким как SOAP, CORBA и RPC.

Передача состояния представления введен и определен в 2000 году Роем Филдингомв его кандидатской диссертации «Архитектурные стили и проектирование архитектур сетевого программного обеспечение» [14].

Веб-сервисы REST являются веб-сервисами, реализуемые с использованием HTTP и принципов REST. Как правило, Web-сервис RESTful определяет URI основного ресурса, поддерживаемые MIME-типы представления/ответа и операции.

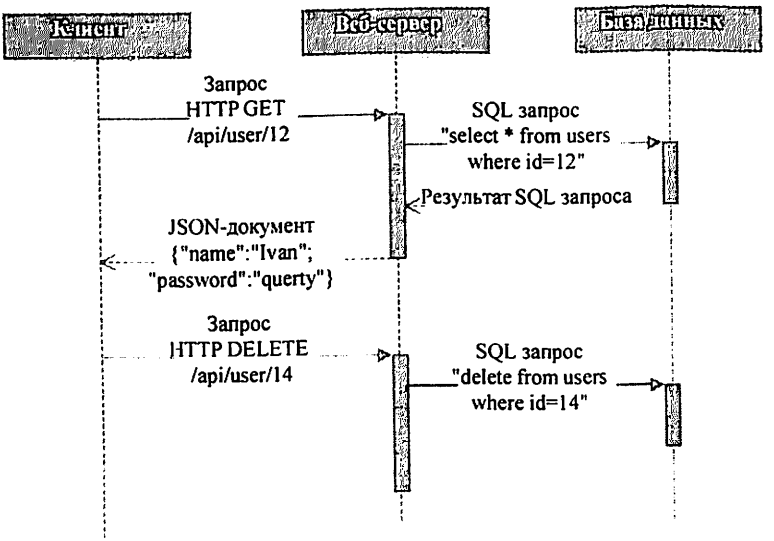

Рис. 4.5. REST веб-сервис

Листинг . Пример запроса к REST веб-сервису

```
GET /StockPrice/IBM HTTP/1.1
Host: example.org
Accept: text/xml
Accept-Charset: utf-8
```

Листинг . Пример ответа на запрос к REST веб-сервису

```
HTTP/1.1 200 OK
```

```
Content-Type: text/xml;
charset=utf-8;
Content-Length:nnn;
<?xmlversion="1.0"?>
        <s:Quote xmlns:s="http://example.org/stock-ser-
vice">
<s:TickerSymbol>IBM</s:TickerSymbol>
<s:StockPrice>45.25</s:StockPrice>
</s:Quote>
```

Ресурс может являться практически любым понятным изначимым адресуемым объектом.

Представление ресурса — это обычно документ, отражающий текущее или требуемое состояние ресурса. Ресурсы обычно представлются документами в форматах XML или JSON.

JSON (Javascript j.ject notation) — эффективный текстовый формат кодирования данных, обеспечивающий быстрый обмен небольшими объемами данных между клиентскими браузерами и веб-службами с поддержкой AJAX.

JSON-текст представляет собой (в закодированном виде) одну из двух структур:

- Набор пар ключ: значение. В различных языках это реализовано как объект, запись, структура, словарь, хэш-таблица, список с ключом или ассоциативный массив. Ключом может быть только строка, значением — любая форма.
- Упорядоченный набор значений. Во многих языках это реализовано как массив, вектор, список или последовательность.

REST и горизонтальный подход. Стратегия, опирающаяся на горизонтальный подход к протоколам, наиболее радикальна. Слово «горизонтальный» означает в данном случае сохранение существующего уровня, без выстраивания уровней поверх него. Предполагается отказаться от разработки новых протоколов, а использовать несколько хорошо проверенных, считая, что для работы с объектами вполне достаточно уметь выполнять четыре типа действий: создание (Creation), восстановление (Retrieval), изменение (Update) и уничтожение (Destruction). Из этих действий получается так называемый «шаблон проектирования» CRUD. Протокол Hypertext Transfer Protocol определяет методы GET/PUT/POST/DELETE, которые и реализуют шаблон CRUD. Абревиатура (шаблон) CRUD обозначает перечень основных операций с объектом: create (создать), read (считать, загрузить), update (обновить, изменить, отредактировать) и delete (удалить).

Таблица 4.1

HTTP-методы в REST веб-сервисах

HTTP-метод	CRUD операция
GET	Read
POST	Create (иногда используется для операций update, delete)
PUT	Create (иногда используется для операции update)
DELETE	Delete

Разработчики SOAP веб-сервисов создают свой собственных перечень имен существительных и глаголов (напримерgetUsers(), savePurchaseOrder(...)) для обозначения операций веб-сервиса. В этом смысле SOAP реализуют сервисный принцип работы, в соответствии с которым главную роль при взаимодействии со службами играют их методы. В основе архитектры REST веб-сервисов лежит ресурсный (объектный) подход.

REST веб-сервисы разрабатываются согласно следующим принципам:

1. Возвращайте любые данные по их идентификатору
2. Use standard methods — используйте стандарты, Имеется в виду, экономьте свои силы и деньги заказчика, используйте стандартные методы HTTP.
3. Одни и те же данные можно вернуть различных форматах. Например, в XML или JSON для проследующей программной обработки.
4. Передача данных без сохранения состояния. При обращении к REST-сервису не учитываются результаты ранее выполненных операций. REST приложение не сохраняет никакого состояния сессии на стороне сервера. Вся информация, необходимая для выполнения запроса, передается в самом запросе.

Часто REST веб-сервисы реализуются с помощью инфраструктруры создания MVC веб-приложений [15]. Так, компания Microsoft предлагает к использованию расширение WebAPI, позволяющее создавать классы-контроллеры для обработки запросов REST сервисов [16].

4.7. Пример REST веб-сервиса

Демонстрационный веб-сервис создан с использованием среды разработки Microsoft Visual Web Developer. Все упомянутые далее классы

размещены в проекте ASP.NET MVC Application. Для демонстрационного веб-сервиса создан класс-сущность Individual (физическое лицо):

```
public class Individual
    {
        public int Id;
        public string Name;
        public int Age;
    }
```

Также создан репозиторий-заглушка, реализующих методы CRUD:

```
public class IndividialsRepository
    {
        List<Individual> data = new List<Individual>();
        public IndividialsRepository()
        {
            data.Add(new Individual() { Id = 1, Name =
"Алексей", Age = 10 });
            data.Add(new Individual() { Id = 2, Name =
"Светлана", Age = 6 });
            data.Add(new Individual() { Id = 3, Name =
"Петр", Age = 5 });
        }
        public IEnumerable<Individual> Get()
        {
            return data;
        }
        public void Add(Individual item)
        {
            data.Add(item);
        }
        public void Update(Individual item)
        {
            var tmp = data.Where(x => x.Id ==
item.Id).FirstOrDefault();
            tmp = item;
        }
        public bool Delete(int Id)
        {
```

```
            var item = data.Where(x => x.Id ==
Id).FirstOrDefault();
            if (item != null)
            {
                data.Remove(item);
                return true;
            }
            return false;
        }
}
```

Непосредственно методы REST-сервиса реализованы в классе IndividualAPIController:

```
public class IndividualAPIController : ApiController
    {
        IndividialsRepository repository = new IndividialsReposi-
tory();
        // GET api/individualapi
        public IEnumerable<Individual> Get()
        {
            return repository.Get();
        }

        // GET api/individualapi/5
        public Individual Get(int id)
        {
            return repository.Get().Where(x=>x.Id==id).First();
        }

        // POST api/individualapi
        public void Post([FromBody]Individual ind)
        {
            repository.Add(ind);
        }

        // PUT api/individualapi/5
        public void Put(int id, [FromBody]Individual ind)
        {
            repository.Update(ind);
        }

        // DELETE api/individualapi/5
        public void Delete(int id)
        {
        }
    }
```

В классе-контроллере используется соглашение об именовании методо: названия методов класса должны совпадать с методами протокола HTTP.

Возвращаемое XML- значение по HTTP-запросу GET /api/individualapi формируется в результате сериализации массива и выглядит следующим образом:

```
<ArrayOfIndividual>
     <Individual>
          <Age>10</Age>
          <Id>1</Id>
          <Name>Алексей</Name>
     </Individual>
     <Individual>
          <Age>6</Age>
          <Id>2</Id>
          <Name>Светлана</Name>
     </Individual>
     <Individual>
          <Age>5</Age>
          <Id>3</Id>
          <Name>Петр</Name>
     </Individual>
</ArrayOfIndividual>
```

Также возможно получить JSON-результат:

```
[{"Id":1,"Name":"Алек-
сей","Age":10},{"Id":2,"Name":"Свет-
лана","Age":6},{"Id":3,"Name":"Петр","Age":5}]
```

Для получения JSON-результата требуется указать в параметрах запроса тип принимаемых данных:

```
GET /api/individualapi HTTP/1.1
User-Agent: Fiddler
Host: localhost:9883
Content-Type: application/json; charset=utf-8
```

4.8. Средства отладки веб-сервисов

Среди подходов для отладки веб-сервисов можно выделить следующие:

1. Отладка на стороне сервера
2. Отладка на стороне клиента
3. Использование специализированных HTTP-отладчиков, наиболее популярным их которых является программное приложение Fiddler.

Сервисная шина предприятия (*Enterprise service bus*) – архитектурное решение для построения распределённых корпоративных информа- ционных систем.

Обычно включает в себя промежуточное ПО, которое обеспечивает взаимосвязь между различными приложениямипо различным протоколам взаимодействия.

Основной принцип сервисной шины — концентрация обмена сообщениями между различными системами через единую точку, в которой, при необходимости, обеспечивается транзакционный контроль, преобразование данных, сохранность сообщений.

Все настройки обработки и передачи сообщений предполагаются также сконцентрированными в единой точке, и формируются в терминах служб, таким образом, при замене какой-либо информационной системы, подклю- чённой к шине, нет необходимости в перенастройке остальных систем.

Решения ESB поставляются компаниями Oracle (Oracle Service Bus), IBM(IBM WebSphere ESB), Microsoft (Microsoft BizTalk Server).

Также существуют бесплатные шины сервисов с открытым исходным кодом: Open ESB, Ultra ESB, Apache Service Mix и др [17].

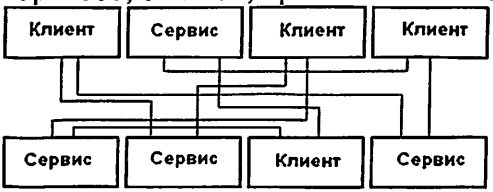

Рис. 4.6. Использование веб-сервисов без ESB

Основные функции ESB представлены в приведенном ниже списке:

- поддержка синхронного и асинхронного способа вызова служб;
- использование защищённого транспорта, с гарантированной доставкой сообщений, поддерживающего транзакционную модель;
- статическая и алгоритмическая маршрутизация сообщений;
- доступ к данным из сторонних информационных систем с помощью готовых или специально разработанных адаптеров;
- обработка и преобразование сообщений;
- оркестровка и хореография служб;
- протоколирование.

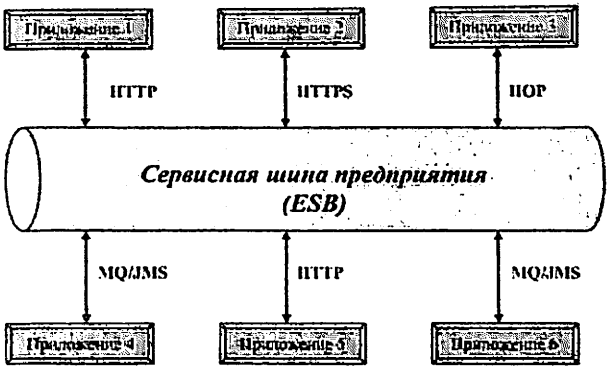

Рис. 4.7. Сервисная шина предприятия

Организация OMG определяет оркестровку как «моделирование направленных,в нутренних бизнес-процессов», а хореографию как «спецификацию взаимодействий между автономными процессами». Оркестровка в бизнес-процессах – это серия действий в управляемом потоке работ, обычно имеющем одну линию выполнения.Хореография отражает видимый обмен сообщениями, правила взаимодействий и соглашения между двумя и более сервисами.

BPEL (*Business Process Execution Language*) — язык на основе XML для формального описания бизнес-процессов и протоколов их взаимодействия между собой. Для создания BPEL процесса необходим BPEL-визуальный редактор. В качестве редактора может использоваться среда разработки Eclipse с установленным дополнением BPEL. Для исполнения BPEL необходи веб-сервер, такой как Tomcat ODE.

BPEL проект состоит минимум из двух файлов:

1. XML-файл с расширением *.bpel.
2. WSDL-файл, описывающий веб-службу.

Процесс, определенный в BPEL4WS, состоит из:

- Действий (activities), которые являются отдельными бизнес-этапами внутрипроцесса.Действия могут быть простыми или состоять из других действий(структурированными).
- Ссылок на партнеров, которые определяют внешние сущности,взаимодействующие с процессом или, наоборот, использующие WSDL-интерфейсы.

- Переменных, хранящих сообщения, передаваемые между действиями, и,следовательно, представляющих состояние.
- Корреляционных наборов (correlation sets), использующихся длякорреляциинескольких сообщений запросов сервиса и ответов с одним экземпляром бизнес-процесса. (К сервису могут обращаться различные бизнес-процессы, нельзясмешивать результаты работы для разных бизнес-процессов)
- Обработчиков неисправностей (fault handlers), занимающихся исключительнымиситуациями, которые могут возникнуть во время работы бизнес-процесса.
- Обработчиков событий (event handlers), принимающих и обрабатывающихсообщения параллельно с обычным выполнением процесса
- Корректирующих обработчиков(compensation handlers), определяющих логикукоррекции для отката действия или нескольких действий при возникновенииисключительной ситуации.

Windows Workflow Foundation является средством определения, выполнения и управления рабочими процессами. WWF ориентирована на визуальное программирование и использует декларативную модель программирования. Процесс *Windows Workflow Foundation* может быть развернут не только в виде исполняемого файла (консольное или оконное приложение) как SOAP веб-сервис, а в процессе своей работы сам процесс может выполнять обращения к другим веб-сервисам. Соответственно, технология Windows Workflow Foundation может быть использована для реализации оркестровки веб-сервисов [18, 19].

4.10. Пример создания сети управляющего веб-сервиса с помощью технологии Windows Workflow Foundation

В рассматриваемом примерсоздается управляющий веб-сервис с использованием технологии *Windows Workflow Foundation.* Создаваемый веб-сервис обращается к сервису для получения названия населенного пункта по его почтовому индексу. Для создания сети веб-сервисов необходимо в среде разработке Microsoft Visual Studio создать новый проект WCF Workflow Service Application.

В данном проекте веб-сервис создается в режиме визуального программирования. Начальным и конечным элементами рабочего процесса являются компоненты *ReceiveRequest* и *SendRequest.* Операторы цикла и

условия указываются с помощью компонентов группы *Control Flow* (рис.).

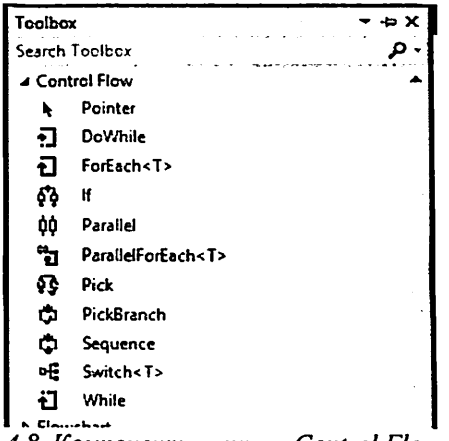

Рис. 4.8. Компоненты группы Control Flow

Для того, чтобы осуществить композицию SOAP-сервисов, необходимо добавить ссылку на них через специальное меню инструмента *Solution Explorer*.

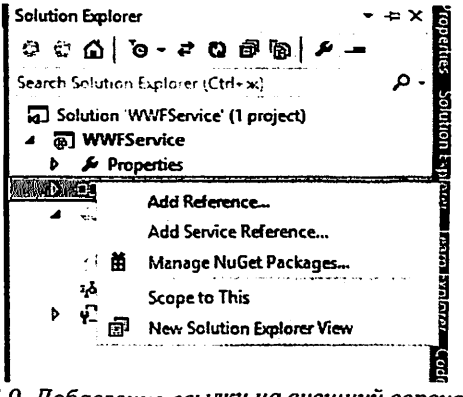

Рис. 4.9. Добавление ссылки на внешний сервис

После добавления ссылки на внешний веб-сервис и постороения (компиляции) решения, в панели инструментов появляются новые компоненты для обращения к данному сервису.

Если построение решения не выполнено, новые компоненты не будут доступны для использо- вания.

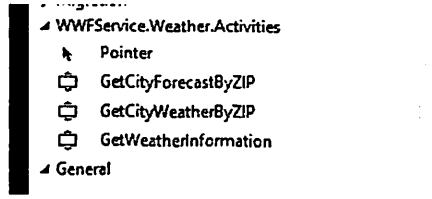

Рис. 4.10. Сгенерированные компоненты для обращения к внешнему веб-сервису

Пример управляющего Windows Workflow Foundation веб-сервиса приведен на рисунке 4.11.

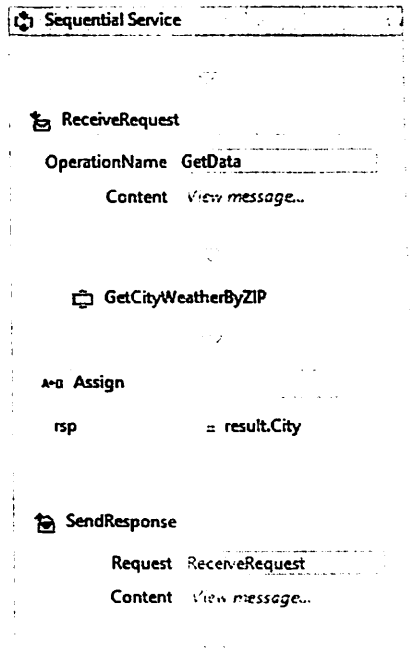

Рис. 4.11. Пример композиции сервисов.

4.11. Безопасность веб-сервисов

Общим подходом является использование защищенной версии протокола HTTP - HTTPS. Если обращение к REST-сервису происходит через *JavaScript*-клиент, то для аутентификации доступны HTTP-сессии и файлы *cookies*.

Популярным способом авторизации для REST и SOAP веб-сервисов является механизм авторизации *OAuth[20]*. OAuth — открытый протокол авторизации, который позволяет предоставить третьей стороне ограниченный доступ к защищенным ресурсам пользователя без необходимости передавать третьей стороне логин и пароль.

OAuth определяет три роли: клиент, сервер и владелец ресурса. Эти три роли присутствуют в любой операции *OAuth*, в некоторых случаях клиент также является владельцемресурса.

Технология OAuth авторизует клиентские приложения по токе-нам. Каждый токен — это цифро-буквенная последовательность, в кото-рой зашифрована следующая информация:

- идентификатор пользователя, который разрешил доступ к своим данным;
- идентификатор приложения, которому разрешен доступ;
- набор действий, доступных приложению.

Каждый *OAuth*-токен имеет время жизни. Время жизни токена — это срок, в течение которого токен принимается сервисами владельцем сервисов. Максимальное время жизни зависит от прав, выбранных при регистрации приложения:

- Вечный токен. Никогда не устаревает и может быть отозван только пользователем. При регистрации приложения отображается время жизни «бесконечно».
- Продлеваемый токен. Устаревает после нескольких месяцев, но продлевается при каждой авторизации с этим токеном.При регистрации приложения отображается минимальное время жизни, например, «не менее, чем 1 год».
- Ограниченный токен устаревает через время, установленное для соответствующих прав доступа.

Любой токен независимо от времени его жизни может быть отозван владельцем ресурса.

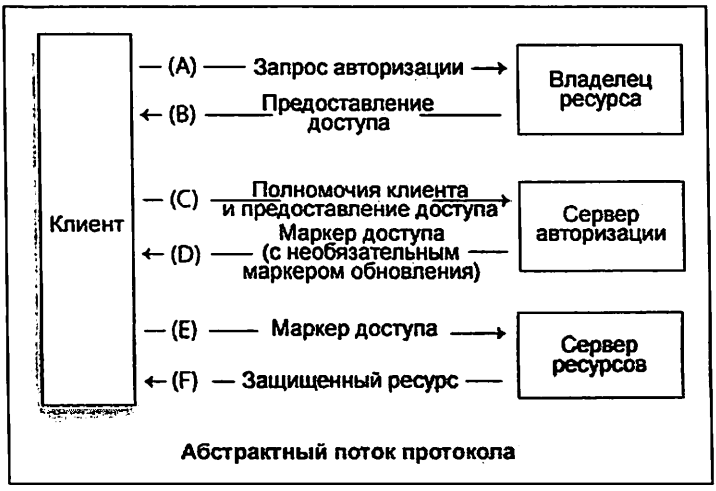

Рис. 4.12. REST веб-сервис

Технология OAuth в настоящее время широко используется в REST-интерфейсах приложений, работащих по принципу социальных сетей: *Twitter*, *Facebook*, ВКонтакте. Также *OAuth* используется в проекте Веб-обсерватории [21].

Задание

Изучите документацию по одному из предложенных API. Разработайте программное приложение-клиент, выполняющие операции чтения данных и записи, используя REST веб-сервисы.

- REST APIs | Twitter Developers URL: https://dev.twitter.com/rest/public
- REST | LinkedIn Developer Network URL: https://developer.linkedin.com/rest
- Справочник по API - Google+ Platform — Google Developers URL: https://developers.google.com/+/api/latest/

Контрольные вопросы

1. Какие способы интеграции программных приложений существовали до появления веб-сервисов? Каковы их достоинства и недостатки? Какие преимуществами обладают веб-сервисы?
2. Размер каких сообщений больше: SOAP и JSON-сообщений REST-сервиса?
3. Каким образом определяется операция при обращении к REST веб-сервису?
4. Какими способами может происходить аутентификация пользователя при обращении к REST-сервису
5. Как расшифровывается абревиатура CRUD? Сопоставьте операции CRUD и типы HTTP-запросов.
6. В каком формате передаются данные при взаимодействии с REST веб-сервисом?
7. Каким образом кодируются данные в формате JSON?
8. Чем отличается ресурсно-ориентированный подход от сервис-ориентированного? Какой подход реализован в REST-сервисах и какой в SOAP-сервисах?
9. К каким сервисам (REST или SOAP) применим стандарт WSDL?
10. Чем отличается ресурсно-ориентированный подход от сервисно-ориентированного?
11. Чем отличается хореография от оркестровки веб-сервисов?
12. С помощью каких инструментов может быть реализована оркестровка веб-сервисов?
13. Какую функциональность реализует шина сервисов предприятия (ESB)?

Спикок литературы

1. Руководство Microsoft по проектированию архитектуры приложений. 2е издание. 2010. Дата обновления: 01.01.2010. URL: http://apparchguide.ms/Book (дата обращения: 01.09.2013).
2. Clements P., Bachmann F., Bass L. Documenting Software Architectures: Views and Beyond (2nd Edition). - Boston: Addison-Wesley Professional, 2010.
3. Sommerville I. Software Engineering (9th Edition). – Boston: Addison-Wesley. – 792 p.
4. Gomaa H. Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures. – NY: Cambridge University Press, 2011. – 578 p.
5. MSDN: Ascpect oriented programming. URL: https://msdn.microsoft.com/en-us/library/aa288717(v=vs.71).aspx
6. Bewis T. C# Design Pattern Essentials. – NY: Ability First Limited, 2012. – 264 p.
7. Seemann M. Dependency Injection in .NET. – NY: Manning Publications, 2011. – 584 p.
8. Betts D., Melnik G, Simonazzi F. Dependency Injection with Unity (Microsoft patterns & practices). - NY: Microsoft patterns & practices, 2013
9. Evans, E. Domain-Driven Design Reference: Definitions and Pattern Summaries. – NY: Dog Ear Publishing, 2014. – 88 p.
10. Мацяшек Л.А. Анализ и проектирование информационных систем с помощью UML 2.0. – 3-е изд. : Пер. с англ. – М.: ООО "И.Д. Вильямс", 2008. – 816 с.
11. Zimmermann O, Tomlinson M., Peuser S. Perspectives on Web Services: Applying SOAP, WSDL and UDDI to Real-World Projects. – NY:Springer, 2013. – 643 p.
12. W3C. SOAP Specification. URL: http://www.w3.org/TR/soap/
13. Kalin M. Java Web Services: Up and Running– NY: O'Reilly Media, 2009. – 360 p.
14. Fielding, Architectural Styles and the Design of Network-based Software Architectures. Dissertation for the degree of doctor of philosophy in Information and Computer Science. *www.ics.uci.edu/~fielding/pubs/dissertation/top.htm*
15. Freeman A. Pro ASP.NET MVC 5 (Expert's Voice in ASP.Net). – NY: Apress, 2013. – 832 p.
16. Kurtz, Jamie. ASP.NET MVC 4 and the Web API. – NY: Appress, 2013. – 141 p.

17. Message Bus (Шина сообщений). URL: http://msdn.microsoft.com/en-us/library/ms978579.aspx (дата обращения: 01.09.2013).
18. Arking J., Millett S. Professional Enterprise .NET. – NY: Wrox, 2009. – 504 p
19. White B. Pro WF 4.5.– NY: Appress, 2013. – 652 p.
20. Bihis C. Mastering OAuth 2.0 Charles. – United Kingdom, Birmingham: Packt, 2015. –238 p.
21. Tiropanis, Thanassis, Wang, Xin, Tinati, Ramine and Hall, Wendy (2014) Building a connected Web Observatory: architecture and challenges. In, *2nd International Workshop on Building Web Observatories (B-WOW14), ACM Web Science Conference 2014, Bloomington, US, 23 - 26 Jun 2014.* 10pp.

Приложение А. UML диаграмма классов

UML диаграмма классов используется для представления классов и отношений между ними. Основные элементы диаграммы классов:

1. Класс
2. Интерфейс

Стереотипы являются механизмом расширяемости в унифицированном языке моделирования UML. Стереотипы широко используются на диаграмме классов. Они позволяют проектировщикам расширять словарь UML для создания новых элементов моделирования, получаемых из существующих, но имеющих определенные свойства, которые подходят для конкретной проблемы предметной области или для другого специализированным использования. Термин происходит от первоначального значения слова «стереотип», который используется в книгопечатании. Например, при моделировании сети вам могут понадобиться символы для представления маршрутизаторов и концентраторов. С помощью стереотипных узлов вы можете представлять их в виде примитивных строительных блоков. Графически стереотип отображается как имя, заключенное в кавычки («», или, если такие кавычки недопустимы, <<>>) и расположенное над именем другого элемента.

Допустимые отношения на диаграмме классов:

1. Наследование
2. Реализация
3. Использование (зависимость) - кратковременное отношение. Объект обладает ссылкой на искользуемый объект временно. Ссылка на используемый объект передается как входной параметр метода. После того, как метод завершил свою работу, владение ссылкой на объект прекращается.
4. Ассоциация - длительное отношение между объектами. Обычно ссылка на объект реализована как поле класса.

Частными случаями ассоциации между классами бывают:

1. Двунаправленная ассоциация - объекты классов, участвующих в отношении, длительно обладают ссылками друг на друга.
2. Однонаправленная ассоциация - только один из двух объектов обладает ссылкой на другой.

3. Агрегация. Показывает отношение Часть-Целое.
4. Композиция (композитная агрегация). Композиция имеет жёсткую зависимость времени существования экземпляров класса контейнера и экземпляров содержащихся классов. Если контейнер будет уничтожен, то всё его содержимое будет также уничтожено.

`name`	Минимальное возможное описание – указано только имя атрибута
`+name`	Указаны имя и открытая видимость – предполагается, что манипуляции с именем будут производиться непосредственно
`-name : String`	Указаны имя, тип и закрытая видимость – манипуляции с именем будут производиться с помощью специальных операций
`-name[1..3] : String`	В дополнение к предыдущему указана кратность (для хранения трех составляющих; фамилии, имени и отчества)
`-name : String="Svetlana"`	Дополнительно указано начальное значение
`+name : String{readOnly}`	Атрибут объявлен не меняющим своего значения после начального присваивания и открытым[V]

Приложение Б. UML диаграмма последовательностей

Диаграмма последовательностей чаще всего используется для отображения взаимодействия между объектами классов в виде обмена сообщениями. Однако, данная диаграмма также применяется и для иллюстрации более высокоуровневых взаимодействий. Вызов метода объекта класса является передачей сообщения. Передача результата работы метода является ответным сообщением.

Диаграмма последовательностей позволяет содержит следующие элементы:

1. Объект с линией жизни и фокусом управления.
2. Фрагменты OPT (блок if), ALT (блок if-else), LOOP (цикл).
3. Фрагмент REF для ссылки на другую диаграмму последовательностей.

```
public class MessageService
{

        public MessageRepository r;
        public void Post(Message m)
        {
            r.Add(m);
            r.Save();
        }
}
//Вызов метода Post класса MessageService происходит из ме-
тода Main неизвестного объекта
......
public static void Main()
{
......
        service.Post(new Message());
......
}
```