Published in IET Software
Received on 7th April 2008
Revised on 3rd August 2009
doi: 10.1049/iet-sen.2009.0010

ISSN 1751-8806

Evaluation of the ‘replace constructors with creation methods’ refactoring in Java systems

S. Counsell[1] *G. Loizou*[2,3] *R. Najjar*[2]
[1]*School of Information Systems, Computing and Mathematics, Brunel University, Uxbridge, Middlesex UB8 3PH, UK*
[2]*School of Computer Science and Information Systems, University of London, Birkbeck, London WC1E 7HX, UK*
[3]*Department of Computer Science, University of Cyprus, 1678 Nicosia, Cyprus*
E-mail: Steve.Counsell@brunel.ac.uk

Abstract: Class constructors play an indispensable role in the Java language as a mechanism for object creation. However, little empirical evidence exists on constructors, trends in their composition and how a class with too many constructors might influence its understandability by developers. Herein, the authors investigate the applicability of the ‘replace constructors with creation methods’ (RCwCM) refactoring of Kerievsky in five Java systems. The RCwCM was manually applied to a set of classes from each of the five systems in classes containing three or more constructors. The benefits of this refactoring include improved code readability and encapsulation, program understanding and to a lesser extent possible elimination of code duplication. Within each of the five systems, evidence of scope for applying the RCwCM refactoring based on the number of classes with multiple constructors was found. However, problems were encountered that limited its application. These are the nature of inheritance and the different styles of accessing class constructors. In the former, account has to be taken of multiple dependencies if the class is not a leaf class; in the latter, the ‘super’ construct requires careful handling. When considered against the benefits that the RCwCM provides, care needs to be exercised. As with any refactoring effort, the short and long-term benefits need to be compared with the expense outlay (developer time and effort) as well as the opportunity cost.

1 Introduction

The term ‘refactoring’ refers to the technique that seeks to improve code quality through the process of making one change or a series of changes to the internal structure of the software without necessarily changing its external behaviour [1–9]. Refactoring can be used to improve software design by tidying up code, moving code to the right place or removing code. We could say that refactoring requires the programmer to work more deeply on understanding what the code does and is therefore a potential aid to maintenance and reuse [2, 10]. Fowler [2] claims that refactoring is the reversal of software decay and, in this sense, any refactoring effort is worthwhile.

In the Java language, constructors are functions that assign the initial values of the data elements of an object being created. In contrast to normal methods, constructors have no return type and share the same name as the class in which they are defined. While in a procedural language like C [11], faults often arise because a developer forgets to initialise a variable, this problem is partially averted in Java through the requirement of at least one constructor. There is often a trade-off; however, in a class with many constructors, it is often difficult to tell the purpose of each constructor when they differ in only minor ways. In this study, we explore the scope for refactoring classes with a relatively large number of constructors in their body. Five Java systems were empirically investigated with respect to the RCwCM refactoring proposed by Kerievsky [10]. The principle underlying this refactoring is that transformation of constructors into non-constructor methods improves a developer’s ability to understand a class, saves lines of code (LOC) which would otherwise be duplicated across the multiple constructors and improves encapsulation by the creation of a single private catchall constructor (since all constructors were previously defined as public). The potential for improved class understandability (and, in theory, its maintainability) by the creation of non-constructor methods should make the task of developers easier, since they do not necessarily need to understand what each of a large set of constructors does. Finally, since in general the non-constructor methods created by transforming constructors will be named according to their function, it should be much easier for the developer to identify the appropriate method which invokes the constructor. While the RCwCM refactoring may offer many benefits, in a practical, empirical sense and where, for such a refactoring, human intervention is a fundamental aspect, we also need to consider possible limiting factors in its application.

In this paper, we describe the mechanics of the RCwCM refactoring and empirically explore both its benefits and factors limiting its applicability. In essence, we pose the question: when is applicability of RCwCM desirable, from a developer's perspective?

The remainder of this paper is organised as follows. In the following section, the motivation for the research and related works are presented. In Section 3, we give details of the five Java systems investigated, the RCwCM refactoring and the data collected from the sample of classes in each system. In Section 4, we present the data of our empirical study and then in Section 5 we show how features and nuances of the Java language can limit the scope of the RCwCM refactoring; we also describe the limited quantitative benefits that would be obtained from applying this refactoring to the five systems. A discussion of the issues arising from this research is given in Section 6 and, finally, some conclusions are made and further work outlined in Section 7.

2 Motivation and related works

The motivation for the work described in this paper arises from two sources. First, many claims about the benefits of refactoring have been suggested; these include making it easier to add code to existing systems, improving the design of existing systems, improving the understandability of code and making coding 'less annoying' [10] for current and future developers. While these are perfectly reasonable claims to make, the practicalities of a specific refactoring and empirical evidence from systems might limit the extent to which each (or any) of these benefits is realised; the research in this paper explores the applicability in this sense of the RCwCM refactoring.

Second, stated benefits of RCwCM are many (Section 3). The main benefit is that it seeks to eliminate the potential confusion arising from having large (and excessive) numbers of constructors and achieves this by converting them to 'regular' methods; in this sense, it is an aid to program understanding. However, it is only by empirically exploring the mechanics of the RCwCM refactoring that we can appreciate its scope, limitations and the likely benefits from its application.

The work described herein can be cast in the context of a number of other empirical and related studies. The study described supports earlier findings in Counsell *et al.* [12], where a set of library classes were empirically investigated. In that study, the 'substitute algorithm' refactoring [2] (i.e. modification of the body of a method to improve the way it functions) was found to be the most popular type of change identified. Here, we consider five Java systems in order to provide empirical evidence for the theme under consideration; one of these systems is the bean scripting framework (BSF) system. The stability of frameworks in terms of encapsulation trends is also highlighted in Counsell *et al.* [13], where it was shown that from five industrial-sized systems empirically studied, only one system (a framework) conformed to appropriate encapsulation principles in terms of private, protected and public attributes and methods. One feature of the remaining four systems was the existence of protected attributes and methods in classes with no inheritance links whatsoever. We would thus expect a framework system to require less refactoring because of its architectural stability and, where it is needed, for it to be a relatively easy task.

In terms of seminal refactoring literature, the work of Opdyke [14] describes a number of refactorings which should be applied to software. Johnson and Opdyke [4] presented a study in which they describe how to create abstract superclasses from other classes by refactoring. They decomposed the operation into a set of refactoring steps, and provided examples. They also discussed a technique that can automate these steps making the process of refactoring much easier. In [3], some common refactorings based on aggregation and how to convert from inheritance to aggregation are reported. An in-depth analysis of the refactoring trends in open-source systems is provided in [15]. Results showed the most common refactorings of the 15 studied therein to be generally those with a high in-degree and low out-degree when mapped on a dependency graph; the same refactorings also featured strongly in the remedying of bad code smells [2, 16]. Remarkably, inheritance and encapsulation-based refactorings were applied relatively infrequently. The paper thus identified 'core' refactorings central to many of the changes made by developers of open-source systems. A 'peak' and 'trough' effect in the pattern of refactorings was observed across all, but one of the systems studied suggests that refactoring is done in effort 'bursts'. Developing heuristics for deciding on different refactorings, based on system change data, was earlier investigated by Demeyer *et al.* [17]; a full survey of relevant refactoring work can be found in [7]. Recent work on the automation of refactoring can be found in Tokuda and Batory [18], where 14 000 LOC were transformed automatically that would otherwise have had to be carried out by hand. The problems and pitfalls of undertaking even a simple refactoring are described in [19] – the 'encapsulate field' refactoring [2] was used as a basis. The possibility that refactorings are linked in a composite form is considered in [7]. Furthermore, issues associated with poor architectural design and the implication this has for refactoring are discussed extensively in Brown *et al.* [20].

In this paper, we focus predominantly on the opportunity for refactoring class constructors. Several papers have recognised constructors as a confusing factor in the definition of object-oriented (OO) metrics. Briand *et al.* [21] identified constructors as a contributing factor to the problems of measuring cohesion, and in Bansiya *et al.* [22] cohesion metrics were empirically evaluated with and without constructors because of the significant effect they had on metrics values. We note that the work presented herein builds on a preliminary study of constructor refactoring [23]. Hereafter, we extend that work and analyse the role that inheritance plays in the refactoring process, factors limiting the applicability of RCwCM and to a lesser extent the potential for the elimination of code duplication. A number of quantitative analyses have been useful in recent years for uncovering traits in OO software that can inform a developer when modifying software, both at the code and design levels [8, 24–29]. The contents of this paper can be viewed in this spirit.

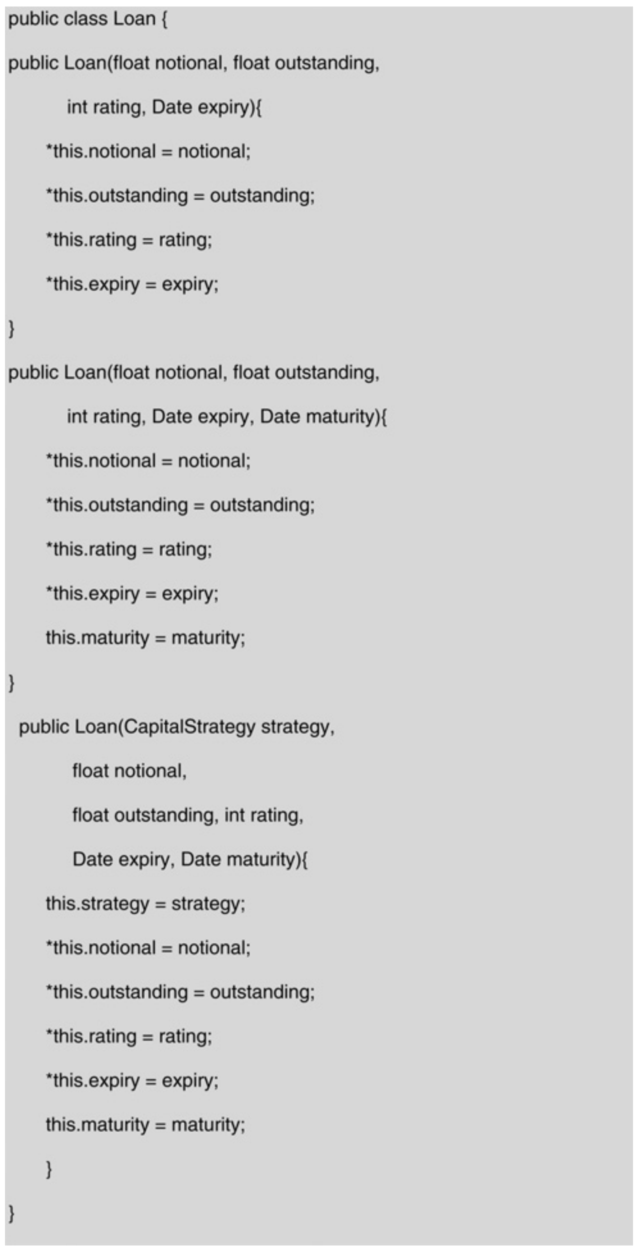
3 Refactoring the ‘Loan’ class

The motivation for refactoring constructors and, in particular, for employing the RCwCM refactoring stems from the fact that constructors do not communicate developer intentions efficiently or effectively. OO languages like Java and C++ insist that the name of the constructor be the same as the class name. This means that first, in the case of a class with many constructors, it is confusing for a developer to appreciate what each constructor does and then decide which of those constructors are amenable to RCwCM. Second, duplicated code in the body of a constructor also obscures the real intention of the constructor, since it becomes more difficult to spot any differences, and it is also possible that some constructors are no longer in use and are hence redundant. Furthermore, mature software systems are filled with duplicated constructor code, because it is simply easier to add another constructor to a class than to invest time and effort finding out invocations of specific constructors. Resulting code ‘bloat’ poses a danger in terms of both software comprehension and future maintenance. The potential benefits of applying the RCwCM refactoring therefore include a greater focus on a minimum number of constructors, the potential for improved code understanding and to a lesser extent reduction of code bloat around constructors.

We use an example of a Loan class [10], in order to demonstrate the principles of the RCwCM refactoring and another example from the Swing system (see Section 3.4), in order to demonstrate that constructors do not always communicate their intention (and therefore their semantics) clearly. To effect the transition from a class containing duplicate code in the constructors to a refactored class, the following steps need to be undertaken. First, the catchall constructor has to be identified. Second, an associated refactoring, the chain constructors (CC) refactoring [10] has to be applied and, finally, the resulting constructors have to be converted to creation methods.

3.1 Identifying the catchall constructor

The following identification of the catchall constructor (Fig. 1) is based on the Loan class example taken from Kerievsky [10]. We begin with an original class containing three constructors for a Loan class; the constructors, emphasising different types of loan, differ in only minor ways. The original class definition is as per Fig. 1 and



```
public class Loan {
public Loan(float notional, float outstanding,
        int rating, Date expiry){
    *this.notional = notional;
    *this.outstanding = outstanding;
    *this.rating = rating;
    *this.expiry = expiry;
}
public Loan(float notional, float outstanding,
        int rating, Date expiry, Date maturity){
    *this.notional = notional;
    *this.outstanding = outstanding;
    *this.rating = rating;
    *this.expiry = expiry;
    this.maturity = maturity;
}
 public Loan(CapitalStrategy strategy,
         float notional,
         float outstanding, int rating,
         Date expiry, Date maturity){
    this.strategy = strategy;
    *this.notional = notional;
    *this.outstanding = outstanding;
    *this.rating = rating;
    *this.expiry = expiry;
    this.maturity = maturity;
    }
}
```

Figure 1 *Original Loan class with duplication*

contains four lines of duplicated code in each constructor (the duplicated LOC have been asterisked ('*') in each constructor). The catchall constructor incorporates all parameters and possible assignments of the set of constructors. By definition, it will have at least as many parameters as the largest constructor, since its signature is the set of all constructor parameters (Fig. 2).

3.2 Applying the CC

The CC refactoring requires constructors with significant levels of duplication to be amalgamated into one constructor, with the remaining constructors using the 'this' self-referencing feature of Java to implement the different constructor calls. After applying the CC refactoring to the above class and carrying out the required testing to ensure no side-effects, the Loan class takes the form of a single catchall constructor with two further self-referencing calls (i.e. using 'this' to the constructor) (Fig. 3).

3.3 Converting constructors to creation methods

The next step of the refactoring is to replace the existing constructors with creation methods (i.e. non-constructor methods), which invoke the single catchall constructor with relevant parameters and null parameters whichever are appropriate. This gives us the following refactored class definition for Loan, in accordance with the RCwCM refactoring (Fig. 4).

We note that the catchall constructor has been declared as private, thus aiding encapsulation by only being accessible from the creation methods. We observe that in the case where the Loan class has subclasses, the constructor should be declared as protected in accordance with appropriate Java encapsulation principles. By reducing the duplication of assignment statements where possible, developers only have to use the method specific to that object, and that method should be more easily identifiable.

```
public Loan(CapitalStrategy strategy, float notional,
            float outstanding, int rating,
            Date expiry, Date maturity){
                this.strategy = strategy;
                this.notional = notional;
                this.outstanding = outstanding;
                this.rating = rating;
                this.expiry = expiry;
                this.maturity = maturity;
}
```

Figure 2 *Catchall constructor for Loan*

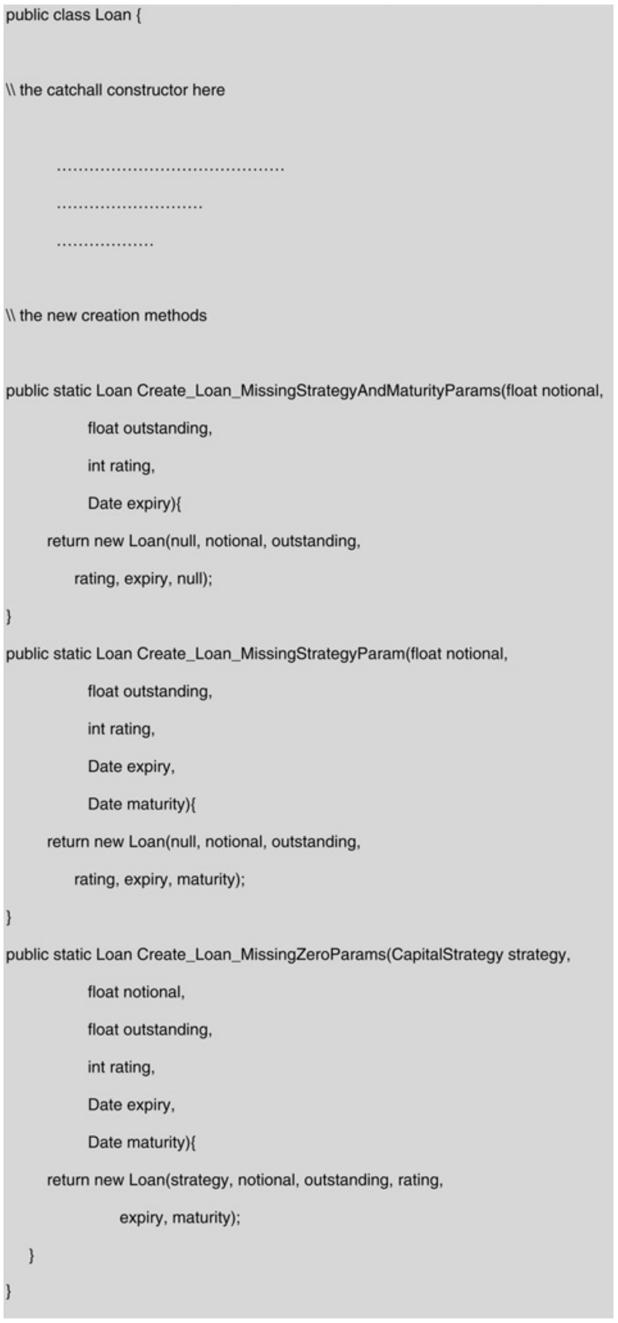
```
public class Loan {
public Loan(float notional, float outstanding,
            int rating, Date expiry){
    this(null, notional, outstanding,
         rating, expiry, null);
}
public Loan(float notional, float outstanding,
            int rating, Date expiry, Date maturity){
    this(null, notional,
         outstanding, rating, expiry, maturity);
}
public Loan(CapitalStrategy strategy, float notional,
            float outstanding, int rating,
            Date expiry, Date maturity){
   this.strategy = strategy;
   this.notional = notional;
   this.outstanding = outstanding;
   this.rating = rating;
   this.expiry = expiry;
   this.maturity = maturity;
   }
}
```

Figure 3 *Loan class with a single catchall constructor*

3.4 JPanel class of Swing

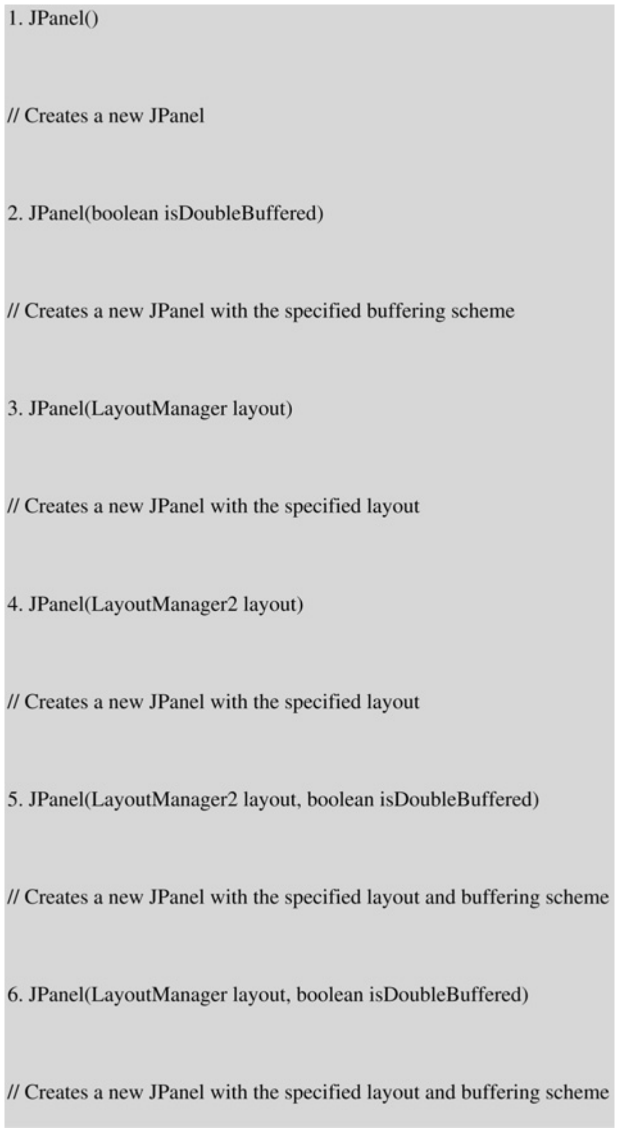
The example of the Loan class in the previous sections describes the principles upon which the RCwCM refactoring is based. One criticism that could be levelled at the Loan example is that it is unlikely that we would find such an ideal example in any real system. In such systems, we have to accept that code is unlikely to conform to an expected 'ideal' template even though it might start its life in such a template in version one of a system. It is also plausible that there would be few classes in any of the systems studied for which the entire mechanics of the RCwCM refactoring, as described for the Loan class, could be applied.

Fig. 5 shows the JPanel class taken from the Swing system (see Section 3.5). JPanel is a lightweight container class that we imagine would be used on a frequent basis by developers as part of any graphic user interface (GUI) application. It has six constructors in total. According to the associated documentation for this class, two pairs of constructors share the same description of what they do. Fig. 5 also shows that the same set of parameters is also shared by those two pairs. Fig. 6 shows the code for each of the constructors. While developers who regularly use the class are probably



```
public class Loan {

\\ the catchall constructor here

        ..........................................
        ...........................
        ..................

\\ the new creation methods

public static Loan Create_Loan_MissingStrategyAndMaturityParams(float notional,
             float outstanding,
             int rating,
             Date expiry){
       return new Loan(null, notional, outstanding,
            rating, expiry, null);
}
public static Loan Create_Loan_MissingStrategyParam(float notional,
             float outstanding,
             int rating,
             Date expiry,
             Date maturity){
       return new Loan(null, notional, outstanding,
            rating, expiry, maturity);
}
public static Loan Create_Loan_MissingZeroParams(CapitalStrategy strategy,
             float notional,
             float outstanding,
             int rating,
             Date expiry,
             Date maturity){
       return new Loan(strategy, notional, outstanding, rating,
                expiry, maturity);
   }
}
```

Figure 4 *Refactored class*



```
1. JPanel()

// Creates a new JPanel

2. JPanel(boolean isDoubleBuffered)

// Creates a new JPanel with the specified buffering scheme

3. JPanel(LayoutManager layout)

// Creates a new JPanel with the specified layout

4. JPanel(LayoutManager2 layout)

// Creates a new JPanel with the specified layout

5. JPanel(LayoutManager2 layout, boolean isDoubleBuffered)

// Creates a new JPanel with the specified layout and buffering scheme

6. JPanel(LayoutManager layout, boolean isDoubleBuffered)

// Creates a new JPanel with the specified layout and buffering scheme
```

Figure 5 *JPanel constructors and their descriptions*

familiar both with the function of the constructor code and with its effect, this might be problematic for developers who use the class for the first time.

Although we would not suggest that this class is necessarily an obvious candidate for the RCwCM refactoring, we could claim that it is not entirely clear under what circumstances each of the two pairs of constructors (3 and 5 and 4 and 6) should be used. They differ only in their use of LayoutManager2 rather than LayoutManager. Scrutiny of the comments associated with the constructors provides no further insight into the nature of the difference between the two pairs of constructors. In fact, investigation of the associated documentation revealed that LayoutManager2 is an interface which extends the LayoutManager interface. (Both interfaces provide mechanisms for describing the layout of a GUI panel.) We thus suggest that if these four constructors were to be refactored along the lines of the RCwCM refactoring, and the names of the methods changed accordingly, then they would convey the meaning of what they do more clearly, because there are only minor differences between the methods that the two interfaces implement. This is, in essence, the chief goal of RCwCM refactoring. The mechanics of undertaking such a refactoring, in this case, would also require a relatively low amount of work on creating the catchall constructor.

The example of the Loan class demonstrates that, ideally, the method names and parameters of the class constructors clearly communicate their intention and hence their semantics. On the other hand, this is not the case with the JPanel class, whose method names and parameters do not

```
1. public JPanel() { this(false); }

2. public JPanel(boolean isDoubleBuffered) { this(new swingwt.awt.FlowLayout(), false); }

3. public JPanel(swingwt.awt.LayoutManager layout) { setLayout(layout); }

4. public JPanel(swingwt.awt.LayoutManager layout, boolean isDoubleBuffered) { setLayout(layout); }

5. public JPanel(swingwt.awt.LayoutManager2 layout) { setLayout(layout); }

6. public JPanel(swingwt.awt.LayoutManager2 layout, boolean isDoubleBuffered) { setLayout(layout); }
```

Figure 6 *Code for the JPanel constructors*

convey their intention; for example, constructors 3 and 5 and 4 and 6. (In addition, the comments in Fig. 5 are far from illuminating.)

The JPanel class example illustrates that a developer should apply the RCwCM on a case-by-case basis rather than on any class with a sufficiently large number of constructors. It is also reasonable to suggest that the benefits of applying the RCwCM refactoring are likely to be directly proportional to the difficulty of unpicking what the constructors of a class currently do and how they do it. In the ensuing subsection, the application domains upon which we empirically investigated the RCwCM refactoring are presented.

3.5 Java systems

The five software systems were chosen according to the following criteria: the software systems are real, not toy systems, differ in their sizes (in terms of the number of classes) and belong to a wide range of application domains (two library packages, compiler, graph-editor and framework). The diversity of application domains and wide spectrum of sizes help to identify important common trends or differences in terms of those criteria across these applications. The five software systems could have been based on one type of application domain only; however, our decision is justified on the basis that we want to obtain more in-depth information about open-source systems across different application domains. The choice of the five systems provides a reasonable basis for comparison of different application domains. The benefit of choosing a wide variety of systems was that trends (if any) were more likely to be as generalisable as possible. The five Java systems were

- *Swing Java package library:* This provides a set of Java components that, to the maximum degree possible, perform the same on all platforms; 1248 classes were identified in this system. This system has 113 294 LOC.

- *BSF:* This is an architecture for incorporating scripting into Java applications and applets; 65 classes were identified in this system. It has 8870 LOC.

- *GraphDraw:* It is a tool for graph drawing and graph layout. Graphs can be input into visualising graph through Java in two ways: with textual descriptions or through a drawing the user creates using the graph editor; 52 classes were identified in this system. It has 11 009 LOC.

- *Libjava:* This language sub-library set has 89 Java classes available from the public domain at the Gnu gcc website (www.gnu.org). It has 6018 LOC.

- *Barat:* It is a compiler front-end for Java. Barat is a framework that supports static analysis of Java programs. It parses Java source code files and class files and builds a complete abstract syntax tree from Java source code files, enriched with name and type analysis information; 407 classes were identified in this system. It has 28 262 LOC.

We view the systems described as small sized in the case of the BSF, GraphDraw and Libjava systems and medium sized in the case of the Swing and Barat systems. We accept that there is no strict definition on what actually defines a large system and, for that matter, what constitutes a medium or small system. However, irrespective of system size, we see it as important to explore features of any industrial system that enjoys a wide user base and particularly it is freely available for download and subsequent use. We note that size of a system in the context of this study is determined by the number of classes (and not LOC).

3.6 Data collection

Wherever possible, data from the five Java systems were collected automatically using software written by the authors. Data collection was undertaken manually when it was clearly impractical to write appropriate data collection software [30]. We collected the number of classes containing three or more constructors automatically. Gathering classes with three or more constructors ensures that, after excluding the default constructor (which rarely has associated code and always has zero parameters), there are at least two constructors remaining. The following data items were collected manually from each of the five systems:

1. The duplicated LOC between each pair of constructors in each class. This data were collected in order to assess the potential for the removal of duplicated LOC. The

underlying hypothesis is that duplicated LOC between any two constructors is unnecessary; such LOC arise out of poor maintenance and should be removed using an appropriate refactoring technique.

2. The number of comment lines surrounding each of the constructors in classes where there were three or more constructors. Very little work has been done on the potential for eliminating comment lines as part of refactoring. Yet, comment line bloat may be a great impediment to effective refactoring as poorly written code leads to code bloat [10]. In the presence of continuous maintenance, a comment may become inaccurate when a developer forgets to change the appropriate comment after changing the code. Removal of such comment lines is a possible by-product of refactoring effort.

We note that the data that identified which sample of classes had three or more constructors had already been collected automatically as part of a range of previous studies [19, 23, 31] and was therefore already in an automated form (as an Excel spreadsheet). This automatic data were originally verified by the authors of those previous studies through a manual checking process. In other words, it is unlikely that any classes falling into the category of containing three or more classes were missed in the data collection process. We also emphasise that the actual refactorings using RCwCM were not implemented by a tool as such – only the effect of undertaking those refactorings was empirically evaluated, quantified and presented in this paper. To date, we know of no software tool that can implement the RCwCM refactoring; however, since we could see the significant benefits in refactoring to patterns, as a future work we plan to develop a tool suite that will explore the properties and quantifiable benefits of a range of Kerievsky's refactorings. Some research by the authors has already been carried out on the former, and more specifically, the number and type of composite refactorings generated by Kerievsky's patterns [32].

3.7 *Identical lines between constructors*

Counting LOC is a process normally fraught with difficulty [33]. However, in order to assess the benefits of constructor refactoring, LOC would seem to be a good indicator of the inefficiencies found in constructors, particularly as constructors tend to comprise simple assignment statements whose comparison with other statements is relatively easy. However, we still need to be clear on first, what a LOC is and, more importantly, what a duplicated LOC is. Hereafter, we consider two LOC common if they are syntactically identical. (We have already seen an example of duplicated LOC in Section 3.2 with four duplicated LOC between all constructors.) We also insist that the types of the parameters being assigned in a constructor must also be identical for two LOC to be considered identical. Finally, we note that in the case of 'if' conditions within constructors, only each line of the 'if' condition is considered for a match with other constructors (i.e. not the entire 'if').

4 Data presentation

In this section, 'cons' stands for 'constructors' and 'max' for the 'maximum' value in the sample. Table 1 gives the breakdown of maximum, mean and median number of constructors found for classes with three or more constructors (here 'nc' stands for not computable, since the sample size in this case was only two; as well as in the sample size, we also include the total number of classes in brackets).

From Table 1 we can see that the Swing and Barat systems contain the highest number of classes with three or more constructors. However, in terms of the proportion of such classes to the total number of classes in each Java system, BSF contains the lowest percentage (3.08%) compared with the highest percentage for Barat (11.79%). In addition, we mention that the Barat system also contained the highest proportion of abstract classes (30) among the 407 investigated for this system. This compares with 38 abstract classes for Swing, one for BSF, one for GraphDraw and three for Libjava.

Table 2 shows the frequencies for each of the constructors in those classes that have three or more constructors. For

Table 1 Number of classes with three or more constructors for the five Java systems

System	Occurrences	Cons	Max	Mean	Median
Swing	73 (1248)	333	9	4.56	4
BSF	2 (65)	7	4	3.5	*nc*
GraphDraw	6 (52)	19	4	3.17	3
Libjava	9 (89)	41	12	4.55	3
Barat	48 (407)	158	7	3.29	3

Table 2 Frequencies for each of the constructors in classes with three or more constructors

Number of cons	Swing	BSF	GraphDraw	Libjava	Barat
3	26	1	5	7	39
4	18	1	1	0	7
5	9	0	0	0	0
6	10	0	0	0	1
7	4	0	0	0	1
8	4	0	0	1	0
9	2	0	0	0	0
12	0	0	0	1	0

example, the Swing system contains nine classes with five constructors and the Barat system contains 39 classes with three constructors. The Libjava system is the only system containing a class with 12 constructors. From this distribution, it would appear that in both Swing and Barat, the potential for RCwCM refactoring is greater than in the other three systems.

5 Factors limiting the RCwCM refactoring

At the inception of the empirical study described, it seemed that refactoring constructors according to the RCwCM refactoring would be a straightforward task and that most classes would conform to theory. The example of the Loan class (Section 3) portrays a model example of how the RCwCM works. However, there were a number of factors that limited the scope for application of the RCwCM refactoring. In the next two sections we look at the two facets of Java (and OO in general) encountered during the course of the empirical study (Sections 5.1 and 5.2). The first of these is the role that inheritance plays in the five systems. When viewed from the perspective of subclass dependencies, inheritance could be seen as a factor limiting the opportunity for undertaking RCwCM. Thereafter, we investigate the different styles used for accessing class constructors that complicate the said refactoring; in particular, use of the ‘super’ construct and variations thereof.

Aside from the limiting factors outlined, we also consider some quantifiable benefits of RCwCM. This falls into two categories: removed duplicated LOC (i.e. ‘bloat’) between constructors and removal of comment lines from constructors. We note that the purpose of the study is to provide a critique of the RCwCM refactoring in terms of its practicalities.

The ensuing four sections, therefore, provide two possible reasons why undertaking RCwCM refactoring might be problematic in some cases followed by two specific, minor benefits, in addition to some of the general benefits referred to in Section 2.

5.1 Role of inheritance

Of the many in-built features that OO provides, inheritance has probably caused the most controversy. The problem that inheritance brings is that while it promotes and encourages code reuse through its natural structure, inheritance also causes potential dependencies between all subclasses of an inheritance tree. Consequently, from a maintenance viewpoint, any change to a superclass could affect any number of other subclasses which potentially use the constructors of the superclass.

The problem with applying the RCwCM refactoring to a system with deep inheritance structures is that there is both direct coupling (with the immediate subclasses) and indirect coupling (with classes below the immediate subclasses) and the class under review for refactoring. This complicates the mechanics of the refactoring since each dependent subclass needs to be considered for potential side-effects that might result from superclass constructor invocation. We note that the example in Kerievsky’s text does not use classes that extend any other classes and, in that sense, the example is relatively straightforward.

While in the context of the Java language, constructors are not inherited like regular methods, the constructor of any subclass can nonetheless invoke the constructor of a superclass. In other words, refactoring constructors in any class must take account of behaviour that may be affected in all subclasses. The implication of this feature of OO (and Java) is that if subclassing to deep levels of the inheritance hierarchy is common, then applying the RCwCM refactoring will, on average, be more problematic and time-consuming than if only shallow (or zero) levels of subclassing is a feature. There is also the ever-present possibility of faults being introduced [34] when greater numbers of dependencies (i.e. coupling) need to be considered. Briand *et al.* [25] have shown that the friend facility used in C++ (a form of coupling [35] often used in conjunction with inheritance) can be the cause of faults. Inheritance and the extent of its use can therefore have a significant effect on the scope (and more specifically the likely time and effort) for carrying out the RCwCM refactoring.

With reference to the above, Table 3 shows the frequencies of subclasses for those classes which have three or more constructors. For example, the Swing system has 17 classes each of which has one subclass, and in total those 17 classes have 77 constructors. On the other hand, 45 classes have no subclasses. For Barat, there are two classes with one subclass each and 42 classes with no subclasses.

From Table 3, we can see that Swing, and to a lesser extent Barat, have the widest spread in terms of the number of subclasses of a class, whereas GraphDraw has only one class with one subclass. The general trend observable from Table 3 is that most of the candidate classes for RCwCM have no subclasses. In the Swing system, 61.6% of the classes fall into this category and in Barat 87.5%. The other two systems, BSF and Libjava, have two and nine classes, respectively, with three or more constructors; none of those classes have any subclasses. The large number of classes with three or more constructors, which have no subclasses, simplifies the process of RCwCM refactoring, since complications associated with treating subclasses are eliminated. If a class has a large number of subclasses, every change made to that class may cause a ripple effect on its subclasses. Refactoring the constructors of classes towards the root of the inheritance hierarchy is therefore more problematic, because of the potential for large numbers of dependent subclasses.

Table 3 Frequencies of classes with three or more constructors in three Java systems

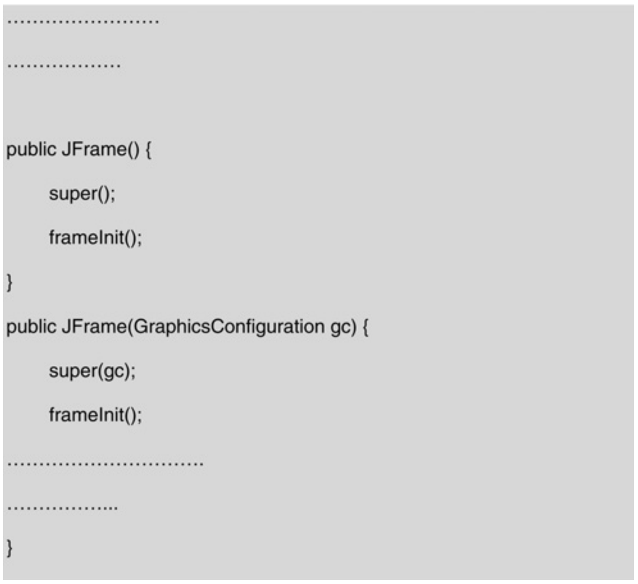
Number of subclasses	Swing	Total number of cons	GraphDraw	Total number of cons	Barat	Total number of cons
0	45	207	5	16	42	135
1	17	77	1	3	2	10
2	3	14	0	0	3	9
3	3	12	0	0	1	4
4	2	10	0	0	0	0
5	1	4	0	0	0	0
8	1	6	0	0	0	0
65	1	3	0	0	0	0

We can thus qualify our earlier statement with respect to the role that inheritance plays in RCwCM by stating that the mechanics of the RCwCM refactoring will tend to be simplified (in terms of time and effort spent on refactoring) if the class under consideration has no subclasses (i.e. the class is a leaf class). Although there is no 'silver bullet' to solve the issue of coupling dependencies, hence to avoid the complications of applying RCwCM where there are many subclasses, one potential strategy for solving this problem of application in deep inheritance hierarchies is to apply the refactoring to systems where there is a low median depth of inheritance and to avoid application of RCwCM to systems where there are deep levels of inheritance. Alternatively, we can apply the refactoring to only classes located at deep levels of the inheritance hierarchy and hence avoid the issue of many subclasses. In other words, from a developer's perspective, we could speculate that the RCwCM is a refactoring that should be applied with care and only under specific conditions that are conducive to, and simplify its application.

5.2 Alternative constructor formats

Java classes tend to call the constructor of their superclasses using the super construct with the appropriate parameter list rather than to explicitly declare a constructor of their own. For example, Fig. 7 illustrates this feature for a class called JFrame (and two constructors therein) taken from the Swing system. Table 4 shows the breakdown of classes in each of the five systems which contained a call to at least one super construct and the number of classes with no calls to super. A striking feature of Table 4 is the relatively low frequency of the use of super in the BSF system. This conforms with the earlier observation about the stability of a framework (Section 2, end of paragraph 3).

It is clear from Table 4 that the super feature is a frequently used construct in Java constructors – about half of the classes considered in Swing and nearly all in Barat. One difficulty that arises with the use of the super feature is the case when we have an invocation of a superclass constructor which contains conditions; the following example (Fig. 8), again taken from the Swing system, illustrates this point.



```
.......................
.................

public JFrame() {
      super();
      frameInit();
}
public JFrame(GraphicsConfiguration gc) {
      super(gc);
      frameInit();
..............................
.................
}
```

Figure 7 *Example of call to the superclass constructor*

Table 4 Number of classes with and without at least one 'super' construct

System	Occurrences	With super	Without
Swing	73	34	39
BSF	2	2	0
GraphDraw	6	0	6
Libjava	9	4	5
Barat	48	44	4

This last example would make the construction of a catchall constructor difficult. The mechanics of splitting the

```
public JWindow(Frame owner) {
    super(owner == null?
    SwingUtilities.getSharedOwnerFrame() : owner);
    windowInit();
}
```

Figure 8 *Constructor with embedded condition*

condition into two is relatively complex and may also detract from the overall comprehensibility of the class – a major motivation for constructor refactoring being undertaken in the first place. There is also the case of declaring an anonymous class in a constructor (an inner class with no name), again making the search for a catchall constructor difficult. To illustrate the use of the super construct, Fig. 9 shows the set of constructors for the JDialog class of the Swing system.

With reference to the difficulty of forming a catchall constructor, Fig. 9 poses a particularly problematic case, with frequent reference to 'super' and with different combinations and numbers of parameter types. While (as per the use of inheritance generally) there is no obvious solution to the problem of the use of the 'super' construct in the mechanics of the RCwCM refactoring as it is defined, there are a number of guidelines that a developer could adopt to ameliorate the issues raised as a result of its use. First, and perhaps most obvious, avoid applying the refactoring in parts of the inheritance hierarchy where there is frequent use of 'super' and/or where there is a deep inheritance hierarchy. The RCwCM should be used selectively. Second, refactor each of the superclasses using RCwCM (to eliminate 'super' from the relevant classes) and thus permit application of the RCwCM refactoring as defined in the target class. The problem with this last guideline is that the overall added effort required might be prohibitive. Third, considering the previous guidelines, there are a number of Fowler defined refactorings that could also assist in the process of applying the RCwCM; for example, use of the 'Collapse Hierarchy' [2] to render the inheritance hierarchy more shallow and malleable for the RCwCM. Finally, there is empirical and anecdotal evidence to suggest that GUI-based systems tend to have a deeper inheritance hierarchy than other 'types' of system [31]. If that is the case, then it might be prudent to avoid applying the RCwCM refactoring in systems or sub-systems which have GUI-based components. In the next two sections, we present two features of RCwCM that provide specific, minor benefits.

5.3 Duplicated LOC

Table 5 gives the breakdown in terms of the number of duplicated LOC found for each pair of constructors in each class. For example, in the Swing system, one line of common code was found between pairs of class constructors 39 times; two lines of common code were found to exist between pairs of class constructors 22 times, and so on. From Table 5, it is seen that Swing and Barat systems contain the widest spread in terms of duplicated LOC as well as having the largest number of classes with three or more constructors as stated earlier. However, it is Barat that provides the greatest opportunity for eliminating LOC during refactoring. The Swing system, although containing large numbers of classes with three or more constructors, tends to have relatively low numbers of duplicated LOC. In terms of refactoring,

```
public JDialog() { super(); }
public JDialog(swingwt.awt.Dialog owner) { super(owner); }
public JDialog(swingwt.awt.Dialog owner, boolean modal) { super(owner, modal); }
public JDialog(swingwt.awt.Dialog owner, String title) { super(owner, title); }
public JDialog(swingwt.awt.Dialog owner, String title, boolean modal) { super(owner, title, modal); }
public JDialog(swingwt.awt.Dialog owner, String title, boolean modal, GraphicsConfiguration gc) {
super(owner, title, modal, gc); }
public JDialog(Frame owner) { super(owner); }
public JDialog(Frame owner, boolean modal) { super(owner, modal); }
public JDialog(Frame owner, String title) { super(owner, title); }
public JDialog(Frame owner, String title, boolean modal) { super(owner, title, modal); }
public JDialog(Frame owner, String title, boolean modal, GraphicsConfiguration gc) { super(owner, title,
modal, gc); }
```

Figure 9 *Code for the JDialog constructors*

Table 5 Frequency of duplicated LOC in each Java system

Duplicated LOC	Swing	BSF	GraphDraw	Libjava	Barat
one	39	0	1	1	54
two	22	0	4	0	3
three	3	0	3	0	7
four	5	0	3	0	5
five	0	0	1	0	4
six	1	0	0	0	5
seven	0	0	0	0	2
eight	0	0	0	0	1
ten	0	0	0	0	2
eleven	0	0	0	0	1
eighteen	0	0	0	0	1

therefore, the Barat system in particular would seem to have evolved with code bloat to its constructors. In fact, one class in the Barat system has no less than 18 lines of duplicated LOC between two of its constructors.

In practical terms, after refactoring these constructors using the CC and then the creation methods template, we would eliminate 222 LOC from the Barat system (0.8%), 118 from the Swing system (0.1%), 35 from GraphDraw (0.32%), just a single line of code from Libjava (0.02%) and none from BSF (0%). Once again, the BSF system appears to be the system where perhaps evolution (however long that may be) has not caused deterioration in its constructors such that they need refactoring [36]. The same can be said (but to a lesser extent) for the Libjava and GraphDraw systems. In terms of all five systems, this is only a small reduction in LOC, and questions the viability and effort investment in RCwCM with respect to this minor benefit.

5.4 Reduction in comment lines

When any refactoring is undertaken, it is likely that some reduction in, or modification of, comment lines is possible. According to Fowler [2], large numbers of comments around code suggest that the code is a bad smell. Fowler also suggests that comments are often superfluous after a refactoring has taken place. As part of our empirical study, the number of comment lines associated with constructors was collected. Table 6 shows the total number of comment lines that surrounded, or were embedded within, constructors for classes with three or more constructors in each of the five Java systems. For example, in Libjava there are nine classes with 41 such constructors and 102 comment lines in total.

From Table 6, we observe that there are varying patterns in the number of comment lines for each of the Java systems under consideration. In general, developers do not tend to follow any particular strategy in writing comment lines related to constructors. The Swing system appears to have the widest range in terms of the number of comment lines surrounding its constructors. In terms of the RCwCM refactoring, an opportunity arises for reducing the number of comment lines when a constructor is replaced by a creation method. The basis of this claim is that the new method should be self-explanatory and does not need as many (if any) comment lines. The only comment lines which would remain would be those surrounding the catchall constructor. Other things remaining equal, allowing comment lines for the catchall constructor and for each of the creation methods, we would expect to eliminate the majority of the 2469 comment lines across the five Java systems. Thus, the RCwCM refactoring employed in this empirical study would provide us with the opportunity to reduce very few LOC but many comment lines. Of course, this saving has to be considered against the possibility that the creation methods themselves will have comment lines, so the net quantitative saving might be less than the figures provided.

5.4.1 Statistical analysis of comment lines:
The following null and alternative hypotheses were investigated to test if a difference existed in the trends of comment lines among the five Java systems.

- H_0 (null hypothesis): There is no variation in the number of comment lines that surrounded, or are embedded within, constructors in Java software systems.

Table 6 Breakdown of comment lines across constructors

System	Sample size	Total number of cons	Total number of comment lines	Min.	Max.	Std. deviation
Swing	73	333	2270	0	115	23.83
BSF	2	7	16	0	16	nc
GraphDraw	6	19	27	0	14	5.58
Libjava	9	41	102	4	22	4.90
Barat	48	158	54	0	21	3.58

Table 7 Mean rank values of the comment lines for constructors in the five Java systems

System (group, see [14])	Occurrences	Median (comment lines)	Mean rank
Swing (5)	73	29	93.5
BSF (2)	2	8	52.3
Graphdraw (1)	6	2.5	50.3
Libjava (3)	9	11	73.3
Barat (4)	48	0	35.4

- H_A (alternative hypothesis): There is variation in the number of comment lines that surround, or are embedded within, constructors in Java software systems.

The non-parametric Kruskal–Wallis test [37] was used for testing differences between the five Java systems. Table 7 shows a summary of the ranked data, the median and the mean rank values of the comment lines in each system; the test statistic H is a function of these ranks.

Table 8 shows the Kruskal–Wallis H-statistic, its associated degrees of freedom (DF) (in this case we have five groups, hence DF $= 4$) and the significance (p-value). We can conclude that the Java software systems under consideration significantly vary (p-value is $0.00 < 0.01$) in terms of the number of comment lines that surrounded, or were embedded within, constructors. Therefore we accept the alternative hypothesis. That is, developers do not tend to follow any particular strategy in writing comment lines related to constructors, and there are varying patterns in the number of comment lines for each system under consideration.

From the preceding analysis, a number of conclusions can be drawn about the nature of comment lines. A previous study by the authors [38] found that one of the main causes of manual data collection errors in the same systems, as studied herein, was the haphazard means with which comment lines had been arranged around methods. Fig. 10 shows a code snippet taken verbatim from the class InnerClassVisitor of Barat. No less than three different styles of commenting are used for four lines of the body of the method. We would suggest that from a readability point of view, mixing comment lines might detract from the effectiveness with which a developer can understand the code and then possibly make changes to that class. In the context of the RCwCM refactoring, appropriately worded comments can help the developer choose the most applicable constructor. In theory, if all the comments were written appropriately, then there would be less need for RCwCM. If the comments are out of date, then this poses an even greater risk of misinterpretation. One of the benefits of RCwCM (albeit qualitative in nature) is that the intention of the 'regular' methods created by the refactoring is, in theory, conveyed more effectively. We could then postulate that comments play less of a role in describing what a method does. In other words, there is less dependence on comment lines if a developer is given a free rein with respect to method naming.

Table 8 Kruskal–Wallis test statistic for comment lines

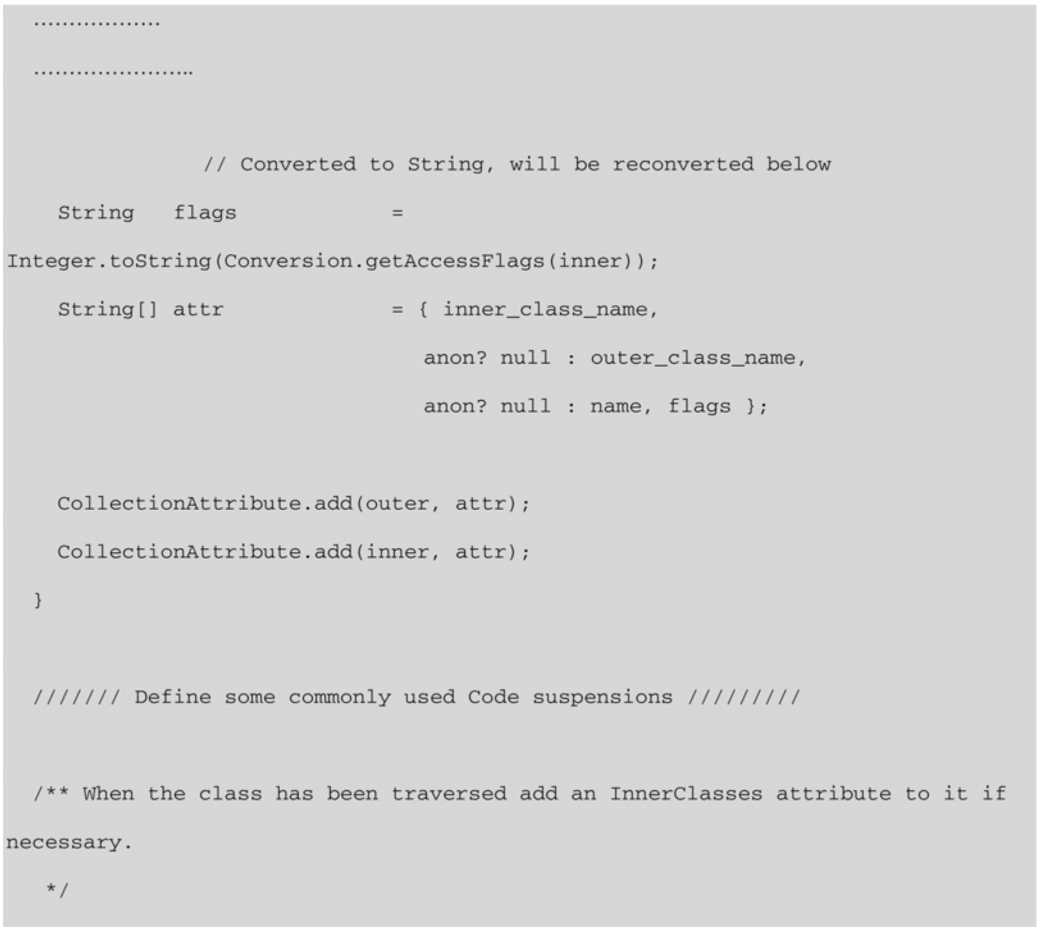
Statistic	Value
H	65.20
DF	4
p-value	0.00

While we could, in theory, suggest that the more the comments are there in code, the better the aid to code comprehension by a developer; an alternative viewpoint is that comments can clutter up the code.

6 Discussion

A number of caveats to the validity of the study need to be considered. The first is that we chose only classes with three or more constructors to consider for the RCwCM refactoring. We accept that a class with two constructors may be appropriate for refactoring effort. However, on the basis that the two constructors would include a default constructor, we feel it inappropriate to consider for refactoring any class with less than three constructors. The second is that we chose five Java systems of largely differing application domains; systems of identical application domain may have provided more relevant results. In defence of such a criticism, we would claim that for the results described in this paper to be generalisable, we would prefer systems of different application domains. Finally, we feel that the study is repeatable for other Java systems and for other languages, for example C++, but due to the different emphasis of the language different problems will arise; for example, the different emphasis of inheritance in C++, combined with the presence of the friend feature [39].

One practical reality that we need to consider is that some classes may have large numbers of constructors; this presents a problem in deciding which constructors to refactor. In Kerievsky [10] it is accepted that a compromise could be reached in such circumstances; the developer is at liberty to be selective in their choice of which constructors should be transformed into creation methods. It is also suggested that the creation methods themselves should be subject to parameterisation, that is, amalgamating the parameter lists of certain methods where there is ample scope for doing this. In this sense, selective refactoring is more appropriate. The point that we make is that not all refactorings are simple in practice when real systems are being analysed. Equally, the way that Java features are used by developers (i.e. inheritance and use of comment lines) can influence the decision as to what to refactor. We also remark that



```
  ..................

  ......................


              // Converted to String, will be reconverted below

    String   flags            =
Integer.toString(Conversion.getAccessFlags(inner));

    String[] attr             = { inner_class_name,

                                  anon? null : outer_class_name,

                                  anon? null : name, flags };


    CollectionAttribute.add(outer, attr);

    CollectionAttribute.add(inner, attr);

  }


  /////// Define some commonly used Code suspensions /////////


  /** When the class has been traversed add an InnerClasses attribute to it if
necessary.
   */
```

Figure 10 *Code snippet from the Barat system*

any refactoring will take time and resources, both of which are in short supply. What is not clear from our study is whether classes with large numbers of constructors have acquired them over time, or had that many constructors from the start; this is an intriguing topic for further work.

In his refactoring text, Fowler suggests a number of reasons why developers do not tend to refactor. One suggestion is that they do not have the time. We therefore accept that refactoring 'everything' in sight is not always feasible or desirable. In certain circumstances, we have suggested that refactoring of constructors is a more difficult process than theory suggests. This is related specifically to situations where the super construct was invoked inconsistently. We are not suggesting that these constructors should be ignored, because they do not fit in with the template for refactoring suggested in Kerievsky; instead, extra time and resources should be allocated for refactoring classes with this format of constructor call if appropriate.

Herein, we have assumed that developers would apply RCwCM as they would like any other refactoring. However, we have to appreciate that there are practical considerations that guide the choice of refactorings that a developer will undertake. First, there is the possibility that there are some code 'smells' [16, 38] which would receive preferential treatment over RCwCM; second, the implications for testing also need to be considered, since different refactorings have mechanics of different complexities and may require different levels of testing [38]. More research needs to be undertaken into the relative trade-offs between refactorings before we can understand the rationale behind developer decisions. Finally, it is important in a paper, where creation methods are featured, to mention their differences with factory methods. The distinction is often a source of confusion among developers and academics alike. Both are methods that create classes and in that sense they are identical. However, Kerievsky [10] makes an important distinction between the two. A factory method may not be static and a factory method '...must be implemented by at least two classes, typically a superclass and a subclass'. In other words, every factory method is a creation method but not vice versa.

We have described how certain factors limit the use of the RCwCM refactoring. These could be viewed as flaws in the refactoring. However, from our empirical study a number of general questions remain. The first is whether the RCwCM refactoring could be improved to take account of these flaws? We would suggest that simply tailoring a refactoring to the vagaries of a particular language is a retrogressive step and might become more trouble than it is worth. If we attempted to modify the RCwCM refactoring to cater for deep inheritance hierarchies, then this would render a relatively simple refactoring (in terms of its mechanics) into an unnecessarily complex one; similarly, with the handling of the super construct. The more sensible route to take might be as we suggested – to re-engineer the code prior to application of RCwCM or to simply avoid using the refactoring where these limiting factors apply. In Fowler's text [2], there are, for many of the mechanics of the 72 refactorings, a set of conditions which could apply at any step in those mechanics. Kerievsky provides a set of alternative 'implementations' of refactorings to cater for different designs. One suggestion therefore, rather than modify the RCwCM, would be to offer an alternative implementation in the case of (a) large numbers of subclasses and (b) extensive use of the super construct. This then gives the developer the choice of which of the two they apply and an idea of the likely effort involved.

The second question relates to whether the RCwCM is actually worth the effort of learning and applying if the said limitations exist? The essence of the RCwCM is the benefit in terms of improved comprehensibility of the code; this is a difficult benefit to quantify. We would suggest that knowledge of the RCwCM is useful even if it is only applied on limited occasions; when used in targeted areas of code or areas that are becoming (or have been) problematic could help immeasurably in saving developer time in both the short and long-term. While we might expect many of the standard renaming and moving refactorings to be undertaken on a frequent basis, application on an 'as-needed' basis of more complex refactorings is a sensible strategy to adopt. In general, we would not expect the RCwCM to be as regular a refactoring as other simple refactorings (e.g. Move Field, Move Method [2]).

7 Conclusions and further work

In this empirical study, we have evaluated RCwCM and, in particular, the role that constructors play in the context of this refactoring. Five Java systems from different application domains were used as a basis for our study. Making constructors intelligible and easy to comprehend is a key to easy maintenance of class definitions. We have shown that while refactoring of constructors according to specific principles described in Section 3 is feasible, there are a number of limiting factors that inhibit the refactoring process for RCwCM. The role that inheritance plays in determining which of the candidate classes to choose for refactoring is an important consideration. The use of the super construct and embedded conditions in signatures also make application of RCwCM problematic. In addition, we empirically showed that the same refactoring is capable of saving only minimal lines of duplicated LOC. Potential benefits in terms of comprehension and encapsulation through use of a private constructor could be accrued. A beneficial side-effect of removing constructors and replacing them with creation methods allowed us to remove comment lines around those constructors.

One aspect of future work will be to analyse the relationships between all 72 refactorings that Fowler suggests [2] and other refactorings of Kerievsky [10] to gain a greater understanding of the nature of composite refactorings. A further aspect of future work will be to monitor the maintenance of systems from the time they are developed to see the changing patterns in constructor trends. Finally, this empirical study represents only a small part of a refactoring toolkit. We would welcome other studies into refactoring of constructors (and other refactorings) to support or refute the findings presented in this paper. To that end, the data upon which this study rests is available to other researchers upon request to the lead author.

8 Acknowledgment

The authors appreciate the highly helpful comments of the anonymous reviewers.

9 References

[1] FOOTE B., OPDYKE W.: 'Life cycle and refactoring patterns that support evolution and reuse', in COPLIEN J., SCHMIDT D. (EDS): 'Pattern languages of programs' (Addison-Wesley, 1995)

[2] FOWLER M.: 'Refactoring (improving the design of existing code)' (Addison-Wesley, 1999)

[3] JOHNSON R., OPDYKE W.: 'Creating abstract superclasses by refactoring'. Proc. ACM 1993 Computer Science Conf. (CSC'93), Indianapolis, USA, 1993, pp. 66–73

[4] JOHNSON R., OPDYKE W.: 'Refactoring and aggregation. Object technologies for advanced software' Springer, 1993, (LNCS, 742), pp. 264–278

[5] JOHNSON R., FOOTE B.: 'Designing reusable classes', *J. Object-Oriented Program.*, 1988, **1**, (2), pp. 22–35

[6] MENS T., VAN DEURSEN A.: 'Refactoring: emerging trends and open problems'. Proc. First Int. Workshop on Refactoring: Achievements, Challenges, Effects (REFACE), University of Waterloo, 2003

[7] MENS T., TOURWE T.: 'A survey of software refactoring', *IEEE Trans. Softw. Engng.*, 2004, **30**, (2), pp. 126–139

[8] O'CINNEIDE M., NIXON P.: 'Composite refactorings for Java programs'. Proc. Workshop on Formal Techniques for Java Programs, ECOOP Workshops, Brussels, Belgium, 1998

[9] TOURWE T., MENS T.: 'Identifying refactoring opportunities using logic meta-programming'. Proc. European Conf. on Software Maintenance and Reengineering (CSMR'03), Benevento, Italy, 2003, pp. 91–100

[10] KERIEVSKY J.: 'Refactoring to patterns' (Addison-Wesley, 2004), Also partially available online at: www.industriallogic.com

[11] KERNIGHAN B., RITCHIE D.: 'The C programming language' (Prentice-Hall, 1978)

[12] COUNSELL S., HASSOUN Y., JOHNSON R., MANNOCK K., MENDES E.: 'Trends in Java code changes: the key identification of refactorings'. Proc. ACM Int. Conf. on Principles and Practice of Programming in Java, Kilkenny, Ireland, 2003, pp. 3–6

[13] COUNSELL S., LOIZOU G., NAJJAR R., MANNOCK K.: 'On the relationship between encapsulation, inheritance and friends in C++ software'. Proc. Int. Conf. on Software System Engineering and its Applications (ICSSEA'02), Paris, France, 2002

[14] OPDYKE W.: 'Refactoring object-oriented frameworks'. PhD thesis, University of Illinois, 1992

[15] ADVANI D., HASSOUN Y., COUNSELL S.: 'Extracting refactoring trends from open-source software and a possible solution to the "related refactoring" conundrum'. Proc. ACM Symp. on Applied Computing, Dijon, France, 2006, pp. 1713–1720

[16] MANTYLA M., VANHANEN J., LASSENIUS C.: 'Bad smells – humans as code critics'. Proc. IEEE Int. Conf. on Software Maintenance (ICSM'04), Chicago, USA, 2004, pp. 399–408

[17] DEMEYER S., DUCASSE S., NIERSTRASZ O.: 'Finding refactorings via change metrics'. Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), Minneapolis, USA, pp. 166–177

[18] TOKUDA L., BATORY D.: 'Evolving object-oriented designs with refactorings', *Autom. Softw. Engng.*, 2001, **8**, pp. 89–120

[19] NAJJAR R., COUNSELL S., LOIZOU G.: 'Encapsulation and the vagaries of a simple refactoring: an empirical study'. Technical Report, BBKCS-05-03-02, SCSIS-Birkbeck, University of London, 2005

[20] BROWN W., MALVEAU R., MCCORMICK R., MOWBRAY T.: 'AntiPatterns: refactoring software, architectures, and projects in crisis' (Wiley, 1998)

[21] BRIAND L., DALY J., WUST J.: 'A unified framework for cohesion measurement in object-oriented systems', *Empir. Softw. Engng.*, 1998, **3**, (1), pp. 65–117

[22] BANSIYA J., ETZKORN L., DAVIS C., LI W.: 'A class cohesion metric for object-oriented designs', *J. Object-Oriented Program.*, 1999, **11**, (8), pp. 47–52

[23] NAJJAR R., COUNSELL S., LOIZOU G., MANNOCK K.: 'The role of constructors in the context of refactoring object-oriented software'. Proc. European Conf. on Software Maintenance and Reengineering (CSMR'03), Benevento, Italy, 2003, pp. 111–120

[24] BRIAND L., BUNSE C., DALY J.: 'A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs', *IEEE Trans. Softw. Engng.*, 2001, **27**, (6), pp. 513–530

[25] BRIAND L., DEVANBU P., MELO W.: 'An investigation into coupling measures for C++'. Proc. Int. Conf. on Software Engineering (ICSE'97), Boston, USA, 1997, pp. 412–421

[26] EL EMAM K., BENLARBI S., GOEL N., RAI S.: 'The confounding effect of class size on the validity of object-oriented metrics', *IEEE Trans. Softw. Engng.*, 2001, **27**, (7), pp. 630–650

[27] HARRISON R., COUNSELL S., NITHI R.: 'Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems', *J. Syst. Softw.*, 2000, **52**, (1), pp. 173–179

[28] HARRISON R., COUNSELL S., NITHI R.: 'Coupling metrics for OO design'. Proc. IEEE Int. Symp. on Software Metrics, Bethesda, Maryland, USA, 1998, pp. 150–157

[29] RUMBAUGH J., JACOBSON I., BOOCH G.: 'The unified modeling language reference manual' (Addison-Wesley, 1998)

[30] COUNSELL S., LOIZOU G., NAJJAR R.: 'Quality of manual data collection in Java software: an empirical investigation', *Empir. Softw. Engng.*, 2007, **12**, (3), pp. 275–293

[31] NAJJAR R.: 'An empirical study on encapsulation and refactoring in the object-oriented paradigm'. PhD thesis Birkbeck, University of London, UK, 2008

[32] HAMZA H., COUNSELL S., HALL T., LOIZOU G.: 'Code smell eradication and associated refactoring'. Proc. European Computing Conf., Malta, September 2008, pp. 102–107

[33] ROSENBERG J.: 'Some misconceptions about lines of code'. Proc. Int. Software Metrics Symp., Albuquerque, New Mexico, USA, 1997, pp. 137–142

[34] OSTRAND T., WEYUKER E., BELL R.: 'Where the bugs are'. Proc. ACM SIGSOFT Int. Symp. on Software Testing and Analysis, Boston, MA, USA, 2004, pp. 86–96

[35] HENDERSON-SELLERS B., CONSTANTINE L., GRAHAM I.: 'Coupling and cohesion (towards a valid metrics suite for object-

oriented analysis and design)’, *Object-Oriented Syst.*, 1996, **3**, (3), pp. 143–158

[36] PERRY D.: ‘Laws and principles of evolution, Panel Paper’. Proc. Int. Conf. on Software Maintenance, Montreal, Canada, 2002, pp. 70–71

[37] FIELD A.: ‘Discovering statistics using SPSS’ (Sage Publications, 2005)

[38] COUNSELL S., HASSOUN Y., LOIZOU G., NAJJAR R.: ‘Common refactorings, a dependency graph and some code smells: an empirical study of Java OSS’. Proc. ACM/IEEE Int. Symp. on Empirical Software Engineering, Rio de Janeiro, Brazil, 2006, pp. 288–296

[39] COUNSELL S., NEWSON P.: ‘Use of friends in C++ software: an empirical investigation’, *J. Syst. Softw.*, 2000, **53**, (1), pp. 15–21