

# **A critical analysis of two refactoring tools**

by Martin Zbigniew Drozd

Supervisor - Derrick Kourie  
Co-supervisor - Andrew Boake

Submitted in partial fulfilment of the requirements for the degree

Magister Scientia (Computer Science)

in the faculty of Engineering, Built Environment and Information Technology

University of Pretoria

November 2007

## **A critical analysis of two refactoring tools**

### **Abstract**

This study provides a critical analysis of refactoring by surveying the refactoring tools in IDEA and Eclipse. Ways are discussed to locate targets for refactorings, via detection of code smells from static code analysis in IDEA and during the compilation process in Eclipse.

New code smells are defined as well as the refactorings needed to remove the code smells. The impacts the code smells have on design are well documented. Considerable effort is made to describe how these code smells and their refactorings can be used to improve design.

Practical methods are provided to detect code smells in large projects such as Sun's JDK. The methodology includes a classification scheme to categorise code smells by their value and complexity to handle large projects more efficiently.

Additionally a detailed analysis is performed on the evolution of the JDK from a maintainability point of view. Code smells are used to measure maintainability in this instance.

## Acknowledgements

I would like to acknowledge:

- Firstly, the higher power that looks on, over every one of us.
- The love and support from my parents and friends.
- All researches whom I have referenced.
- My supervisor and co-supervisor for their guidance.
- All work colleagues who provided me with their support.

Martin Drozd, 11<sup>th</sup> February 2007, South Africa

## List of abbreviations

IDE	Integrated Development Environment
JDK	Java Development Kit
LOC	Lines of code
LHF	Low Hanging Fruit
SLOC	Source lines of code
SDK	Software Development Kit

# Table of Contents

<b>ACKNOWLEDGEMENTS.....</b>	<b>3</b>
<b>CHAPTER 1 INTRODUCTION .....</b>	<b>9</b>
1.1 BACKGROUND.....	9
1.2 MOTIVATION .....	9
1.3 RESEARCH OBJECTIVES.....	10
1.4 OVERVIEW .....	11
<b>CHAPTER 2 SOFTWARE DESIGN, MAINTENANCE.....</b>	<b>13</b>
<b>AND EVOLUTION.....</b>	<b>13</b>
2.1 DESIGN HEURISTICS AND DESIGN PATTERNS.....	13
2.1.1 <i>Design Heuristics</i> .....	13
2.1.2 <i>Design Patterns</i> .....	14
2.1.3 <i>Conclusions</i> .....	17
2.2 SOFTWARE MAINTENANCE .....	18
2.2.1 <i>Definition</i> .....	19
2.2.2 <i>When development becomes maintenance</i> .....	19
2.2.3 <i>Maintenance cost factors</i> .....	20
2.2.4 <i>Maintenance and change prediction</i> .....	20
2.2.5 <i>Source Code Metrics</i> .....	22
2.2.6 <i>Conclusions</i> .....	23
2.3 SOFTWARE EVOLUTION .....	23
2.3.1 <i>Design Evolution</i> .....	24
2.3.2 <i>The 8 laws of software evolution</i> .....	25
2.3.3 <i>Software reengineering</i> .....	25
2.3.4 <i>Conclusions</i> .....	26
2.4 SUMMARY .....	26
<b>CHAPTER 3 REFACTORING .....</b>	<b>27</b>
3.1 DEFINITION .....	27
3.2 OVERVIEW .....	27
3.3 APPLICABILITY .....	28
3.4 MOTIVATIONS.....	29
3.5 PROBLEMS .....	29
3.6 TOOL SURVEY MOTIVATION.....	30
3.7 CODE REVIEW MOTIVATION .....	31
3.8 SUMMARY .....	32
<b>CHAPTER 4 THE ROOTS OF REFACTORING .....</b>	<b>33</b>
4.1 INTRODUCTION .....	33
4.2 FOUNDING WORK .....	33
4.3 RECENT RESEARCH.....	34
4.4 SUMMARY .....	36
<b>CHAPTER 5 ANALYSIS METHODOLOGY .....</b>	<b>37</b>
5.1 JDK CODE BASE STATISTICS .....	37
5.2 CLASSIFYING CODE SMELLS .....	38
5.3 CHOOSING THRESHOLDS IN IDEA .....	41
5.4 JAVA AS THE LANGUAGE OF CHOICE.....	42
5.5 SUMMARY .....	43
<b>CHAPTER 6 ECLIPSE CODE REVIEW .....</b>	<b>44</b>

6.1	UNNECESSARY CODE.....	44
6.1.1	<i>Unused imports</i> .....	48
6.1.2	<i>Unread local variables</i> .....	49
6.1.3	<i>Unread parameter</i> .....	51
6.1.4	<i>Unnecessary throws clause</i> (method or constructor) .....	53
6.1.5	<i>Unused private members</i> .....	55
6.1.5.1	Unused private method .....	55
6.1.5.2	Unused private constructor .....	56
6.1.5.3	Unused private type/class .....	56
6.1.5.4	Unused private member field .....	57
6.2	SUMMARY .....	57
<b>CHAPTER 7 IDEA CODE REVIEW – PART 1 .....</b>		<b>59</b>
7.1	INTRODUCTION .....	59
7.2	CODE REVIEW METHODOLOGY .....	59
7.3	LOCATING DUPLICATES .....	61
7.3.1	<i>Introduction</i> .....	61
7.3.2	<i>Anonymity</i> .....	62
7.3.3	<i>Solutions</i> .....	63
7.3.4	<i>Dealing with Anonymity</i> .....	65
7.3.5	<i>Final Analysis and Conclusion</i> .....	66
7.4	INHERITANCE ISSUES .....	67
7.4.1	<i>Refused Bequest</i> .....	67
7.5	TYPE CODE .....	68
7.5.1	<i>Detection of Type Code</i> .....	68
7.5.2	<i>Replace with Class</i> .....	69
7.5.3	<i>Replace with Subclasses</i> .....	71
7.5.4	<i>Replace with State, Strategy or Command Pattern</i> .....	71
7.5.5	<i>Chains using Instanceof</i> .....	73
7.5.6	<i>Conclusion</i> .....	74
7.6	ABSTRACTION ISSUES .....	74
7.6.1	<i>Feature Envy</i> .....	74
7.6.2	<i>Magic Numbers</i> .....	75
7.7	ENCAPSULATION.....	76
7.7.1	<i>Public Field</i> .....	76
7.8	METHOD METRICS .....	77
7.8.1	<i>Long Method</i> .....	77
7.8.2	<i>Long Parameter List</i> .....	78
7.8.3	<i>Too Many Exceptions</i> .....	79
7.9	CLASS METRICS .....	80
7.9.1	<i>Inappropriate Intimacy</i> .....	80
7.9.2	<i>Large Class</i> .....	81
7.10	SUMMARY .....	82
<b>CHAPTER 8 IDEA CODE REVIEW – PART 2 .....</b>		<b>83</b>
8.1	INTRODUCTION .....	83
8.2	POOR METHOD COMPOSITION .....	83
8.2.1	<i>Method breakdown</i> .....	83
8.2.2	<i>Coupling and Cohesion</i> .....	84
8.2.3	<i>More informative smell detections</i> .....	85
8.3	CREATIONAL ISSUES .....	87
8.3.1	<i>Non-private Utility Class constructors</i> .....	87
8.3.2	<i>Confusing or too many constructors</i> .....	87
8.3.3	<i>Constructors with duplicate code</i> .....	88
8.3.4	<i>Distributed creation information</i> .....	90
8.4	REDUNDANT IF STATEMENTS.....	90
8.5	UNUSED CODE .....	91

8.5.1	<i>Redundant local variables</i> .....	91
8.5.2	<i>Unused method parameters</i> .....	92
8.5.3	<i>Redundant throws clause</i> .....	92
8.5.4	<i>Unused imports</i> .....	94
8.5.5	<i>Field can be local</i> .....	94
8.6	IDEA CODE REVIEW CONCLUSION.....	95
8.7	SUMMARY.....	97
<b>CHAPTER 9 AN IDE COMPARISON.....</b>		<b>98</b>
9.1	FILTERING REFACTORINGS BY CONTEXT IN ECLIPSE .....	99
9.2	PRODUCTIVITY ISSUES .....	100
9.3	HANDLING REFACTORING COMPLEXITY .....	101
9.4	SUMMARY.....	102
<b>CHAPTER 10 CODE EVOLUTION .....</b>		<b>103</b>
10.1	A STATISTICAL VIEW.....	103
10.2	QUICK WIN GROUP.....	106
10.3	STRATEGIC GROUP .....	107
10.4	LOW HANGING FRUIT GROUP .....	109
10.5	AVOID GROUP.....	110
10.6	REMOVING THE AVOID GROUP .....	110
10.7	SUMMARY.....	111
<b>CHAPTER 11 CONCLUSION .....</b>		<b>112</b>
11.1	REFACTORING CONTRIBUTIONS .....	112
11.2	ANALYSIS RESULTS .....	113
11.3	FUTURE WORK .....	114
11.4	FINAL THOUGHTS .....	116
<b>REFERENCES.....</b>		<b>117</b>
<b>APPENDIX.....</b>		<b>121</b>
A.1	REFACTORINGS .....	121
A.2	CODE SMELLS .....	126
A.3	RIEL HEURISTICS .....	130
A.4	METHOD HEURISTICS .....	134
A.5	CODE SMELL TAXONOMY .....	135
A.6	REFACTORING TOOL SUPPORT COMPARISON.....	138

## List of Tables

Table 1: JDK 1.4.2 Code Statistics	38
Table 2: Code smell count for different thresholds	41
Table 3: Using layering to handle exceptions (Eclipse)	54
Table 4: Eclipse code smell summary	58
Table 5: Duplicate group occurrences	67
Table 6: Magic Number example	76
Table 7: Long Method analysis results	78
Table 8: Long Parameter List analysis results	78
Table 9: Using layering to handle exceptions	79
Table 10: Method with too many exceptions analysis results	80
Table 11: Inappropriate Intimacy analysis results	81
Table 12: Large Class analysis results	81
Table 13: Method nesting depth analysis results	86
Table 14: IDEA code smell summary	96
Table 15: An IDE Comparison	99
Table 16: All JDK Code Statistics	104
Table 17: JDK Code Statistics (in percentages)	104
Table 18: JDK - Quick Win Group	106
Table 19: JDK - Strategic Group (low thresholds)	108
Table 20: JDK - Strategic Group (medium thresholds)	108
Table 21: JDK - Strategic Group (high thresholds)	108
Table 22: JDK - Strategic Group (Threshold independent smells)	108
Table 23: JDK – Low Hanging Fruit Group	109
Table 24: JDK - Avoid Group	110

## List of Figures

Figure 1: Singleton Design Pattern Structure diagram.....	15
Figure 2: Observer Design Pattern Structure diagram.....	16
Figure 3: Maintenance Prediction .....	21
Figure 4: A refactoring classification graph.....	39
Figure 5: Error categories under Java compiler settings in Eclipse.....	45
Figure 6: Revealing unnecessary code in Eclipse. ....	45
Figure 7: Filter icon.....	46
Figure 8: Warnings filtered by unused imports and sorted by Resource.....	46
Figure 9: Setting a filter search prefix.....	48
Figure 10: Variable oldValue is local and unused.....	49
Figure 11: Standard getter - getMnemonic().....	50
Figure 12: A mutable int Wrapper class.....	50
Figure 13: Side effects in getter method. ....	50
Figure 14: A mutable int Wrapper class.....	52
Figure 15: Children of <i>AbstractA</i> .....	52
Figure 16: <i>SomeException</i> and <i>AbstractA</i> .....	54
Figure 17: <i>A::b()</i> throws <i>SomeException</i> .....	54
Figure 18: IDEA file analysis settings .....	60
Figure 19: IDEA code smell selection. ....	60
Figure 20: IDEA Analysis Results .....	61
Figure 21: Duplicate location settings.....	62
Figure 22: Duplicate code found in <i>java.util.regex.Pattern</i> .....	63
Figure 23: Seventeen duplicates groups in <i>java.util.BitSet</i> . ....	65
Figure 24: Embedded type information.....	69
Figure 25: Typesafe Enum pattern .....	69
Figure 26: A type-safe class .....	70
Figure 27: Demonstration of <i>Car</i> and <i>TypeSafeCar</i> .....	70
Figure 28: Factory method for the <i>Car</i> class .....	71
Figure 29: Factory method for the <i>TypeSafeCar</i> class .....	71
Figure 30: Replacing a conditional dispatcher with <i>Command</i> .....	72
Figure 31: <i>Command</i> pattern structure diagram.....	73
Figure 32: Constructor code duplication settings.....	89
Figure 33: Redundant if - before refactoring.....	90
Figure 34: Redundant if - after before refactoring .....	90
Figure 35: Searching for unused code in IDEA .....	91
Figure 36: Exception side effects. ....	93
Figure 37: Class refactorings.....	99
Figure 38: Method refactorings.....	100
Figure 39: Constructor refactorings .....	100
Figure 40: Source lines of code per JDK version.....	104
Figure 41: Files per JDK version .....	105
Figure 42: Code smells per JDK version (High thresholds).....	105
Figure 43: Smells per 1000 SLOC .....	106
Figure 44: Code smells group per JDK version (low threshold values).....	109
Figure 45: Code smells group per JDK version (- avoid group) .....	111



# Chapter 1 Introduction

## 1.1 Background

Refactoring helps to improve the design of existing code and adds in the extra constraint requiring that the behaviour of the refactored code is to stay the same after a refactoring has improved the design of existing code. The intention is to make the code more maintainable.

The design of object-oriented software has been a topic of interest throughout the last decade. Fowler [1999] brought refactoring to the forefront of Java developers. Agile software methodologies such as XP [Beck 2000] have realised the need to accommodate change in software systems. With the need for change in software, come a new set of requirements: system flexibility and ease of maintenance.

With the acceptance and incorporation of refactoring facilities in mainstream integrated development environments (IDEs) such as Eclipse and IDEA, it is worthwhile to study refactoring as it has a widespread impact in a software development life cycle where change occurs frequently. Change can be a result of changing user requirements, a change in the system environment or a new business process requiring additions to the software system.

Identifying where and how change can occur in a software system will help understand what role refactoring can play in the change of a dynamic software system. The focus will mainly be on large software systems and use the Java development kit (JDK) as the basis for the analysis studies.

## 1.2 Motivation

Much has been written and many studies have been executed into the research field of refactoring. Through exposure to various programming languages and different problem domains one is able to see what factors are desirable in a language and which are not. For example, when comparing procedural and object-oriented programming languages one can see that there are clear advantages in terms of reuse, flexibility, encapsulation and so forth. However, these advantages only come with the proper use of object-oriented design, which needs to be taught and mastered before one can build a robust software system.

The use of such techniques as design patterns have been held in high regard in academic and commercial circles. It was only at the turn of the millennium that refactoring started coming to the forefront and more followers started to understand its benefits. Refactoring research is still strong in that recently

Kerievsky [2004] has a book about refactoring to design patterns. There are mail groups available, which support refactoring communities, providing answers and discussions around refactoring. The mail group ([refactoring@yahoogroups.com](mailto:refactoring@yahoogroups.com)) is popular example with over 3800 members at present.

The author of this dissertation was lucky enough to be involved in a J2EE software development project, which involved a major application involving stock trading with equity and future stocks. This is where he got the chance to get development experience as well a look into the world of Java development with Eclipse and a variety of open source products. In essence, his academic interest was married with that of his commercial career.

### ***1.3 Research objectives***

The early objective of this study was to gain as much information about refactoring as possible. This started with background reading into the domain, as well as experimentation with a refactoring tool.

Over time, the focus shifted towards the need for an in-depth tool survey for refactoring. The tool survey was to provide a comparison between some of the many different IDEs, which supported refactoring, and to highlight the effectiveness of these tools in terms of productivity and scope. IDEA and Eclipse were chosen as the two tools for the tool survey.

Once a tool survey was performed by using the 1.4.2 version of the JDK for code smell analysis and refactoring productivity, an analysis was done on the all of the other major versions of the JDK (from 1.0 to 1.5). This analysis provided a study into the evolution of the JDK by measuring the amount of code smells in each version and classifying them according to the proposed classification method.

Therefore, the research objectives can be summarised as follows:

1. To perform a tool survey that investigates refactoring support in common IDEs.
2. To perform a study into the evolution of the JDK from a code smell perspective.

There were a number of research questions that were posed, such as:

1. What are the major advantages of the individual tools?
2. What is the number of refactorings available in each tool?

3. How can code smells be detected within the tool?
4. How is the IDE designed to facilitate code smell detection?
5. Can one tool be considered better than another?
6. Are there any productivity issues with the tools?

Questions from a code evolution point of view where also posed, such as:

1. What is the impact of code smells on software systems as they start to grow and are not refactored in an evolving system?
2. How can one classify code smells, so that it is easier to filter out the less important code smells?
3. How can one measure the quality of a system as it evolves?
4. Can system quality to a certain degree, be measured by the amount of code smells that increase as code size increases over time?

For future work questions such as the following where posed:

1. What other methods can be used to organise the code smells when performing an analysis of the code base?
2. How can the code smell information be mined so that it can be of more use to developers in charge of software maintenance?

## **1.4 Overview**

The following provides a brief overview of all of the chapters contained in this dissertation:

Chapter 2 discusses how software design, maintenance and evolution are impacted by refactoring.

Chapter 3 answers many of the questions around refactoring. It also introduces motivations for the tool survey and JDK code review.

Chapter 4 gives an introduction into the early and recent research done on refactoring.

Chapter 5 provides a description into the analysis methodology used in the Eclipse and IDEA code reviews, which follow.

Chapter 6 presents the Eclipse code review. Ways are looked at to find and remove unnecessary code.

Chapter 7 is the beginning of the IDEA code review. Most of the refactorings here are from Fowler's work except for two identified and discussed in detail by the author.

Chapter 8 is the second part of the IDEA code review. This work provides an overview of the work from Kerievsky [2004], as well as contributions from the author of this dissertation.

Chapter 9 compares the two IDEs used for the code reviews.

Chapter 10 describes the code reviews from a code evolution perspective. It also reveals the statistics on code reviews performed on all major versions of the JDK, from version 1.0 to 1.5.

Chapter 11 provides final thoughts on all of the results in the previous chapters.

## Chapter 2    **Software design, maintenance and evolution**

Throughout programming history, there have been many attempts to improve the design of code. This chapter starts with a description of design heuristics and design patterns (section 2.1) in order to introduce the theme of design issues in relation to object-oriented programming.

Software maintenance and software evolution are also defined (in sections 2.2 and 2.3 respectively) to provide an understanding of the links between software design, maintenance, evolution and refactoring. This will serve as an introduction to the next chapter, which focuses on the answers to the many questions around refactoring. The following sections provide an understanding of the impact of refactoring on the many other phases contained in the software development life cycle.

### ***2.1 Design Heuristics and Design Patterns***

#### **2.1.1 Design Heuristics**

The more general design guidelines for developing object-oriented systems are often represented by design heuristics. The 68 design heuristics (found in the appendix) catalogued by Riel [1996] are good examples of such heuristics. Riel placed these heuristics into eight different categories, where each heuristic that he mentioned had a name, an outline and example of the problem, as well as a suggested approach to solve the problem.

Grotehen and Dittrich [1997] have an object oriented design method called MeTHOOD (Measures, Transformation Rules, and Heuristics for Object-Oriented Design) which discusses 20 design heuristics. For each heuristic, MeTHOOD describes the following: a heuristic name, a short description, a definition, its rationale, its position in life cycle, the heuristic's granularity, an example, subsuming/subsumed heuristics, checking rules, transformation rules, violated heuristics, justification for violating these heuristics, and its effect on measures.

Although these design heuristics are too numerous to memorize in a short space of time (consider Riel's 68 design heuristics), they are nevertheless useful tips on what aspects of the design to focus on. If considered before the initial implementation of an object-oriented program, design heuristics will help considerably in the design of the program. All of these heuristics are discussed in [Riel 1996].

The design heuristics themselves seem very simple at first glance. For deeper insight, one needs to understand why a given heuristic constitutes a guideline for good design in the first place, and this cannot always be easily understood from reading the heuristic itself. Therefore, to understand heuristics, one needs to understand what good design is and before using the heuristic, one should understand in what sense it can improve the design. This requires a detailed description and examples of the problem, which the heuristic proposes to solve.

An example heuristic would be ‘Keep related data and behavior in one place’. This is a very general heuristic and it explains that in order to be able to find related data and behavior, one should keep it in one place. The place in object-orientated programming is a class. Object orientated programming is about coupling data and behavior. Other software architectural styles such as SOA decouple data from behavior. The move (variable or method) refactoring can be used to move unrelated data to the correct classes.

A more complicated heuristic would be ‘Inheritance should only be used to model a specialization hierarchy’. In other words, one should not just use inheritance to share data between classes or just to eliminate duplicate code. Inheritance should rather be used to model the “is a” relationship. For example, a car is a vehicle and should inherit from the vehicle class if there are common behaviors between the vehicle and car. In this example, the car is a specialization of the vehicle.

Another interesting heuristic not mentioned in the appendix is: ‘Choose interfaces over abstract classes’. If one knows something is going to be a base class, the first choice should be to make it an interface, and only if one is forced to have method definitions or member variables should one change it to an abstract class. An interface talks about what the client wants to do, while a class tends to focus on (or allow) implementation details. The ‘extract interface’ refactoring can be used to extract an interface from a class and make that class implement it. All other classes with the same interface can implement it. This refactoring clearly helps in implementing the heuristic. The idea of using a common interface allows one to represent multiple objects, which have different behaviour, but the same interface.

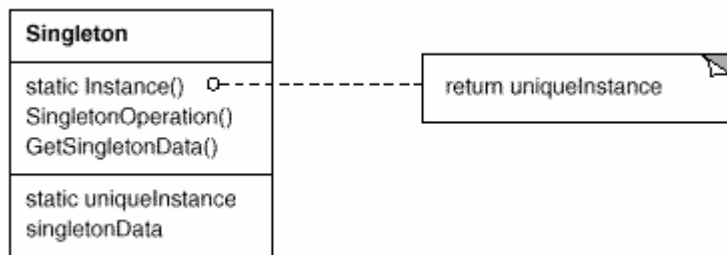
## 2.1.2 Design Patterns

Gamma et al. [1995] have a more specific approach to improving design than the more general approach of following design heuristics, as discussed above. Instead, they advocate the use of design patterns in order to improve design. The design patterns can be seen as solutions to re-occurring design problems and are categorized into behavioral, creational and structural design patterns.

Gamma et al. [1995] mention an important fact in relation to design: “Design patterns help one determine how to reorganise a design, and they can reduce the amount of refactoring one needs to do later. Useful abstractions and patterns are seldom found during analysis or even the early stages of design; they're discovered later in the course of making a design more flexible and reusable.”

When refactoring to patterns, Kerievsky [2004] realised that there is no need for one big static design when building a system that is easy to maintain. Refactoring to patterns will result in refactorings to existing code, which in turn makes further refactorings easier to do, because the initial refactorings take into consideration the design issues in the current code. Therefore, design evolution or continuous design [Shore 2004] plays a big role while implementing software, because it is almost impossible to think of all of the design issues before a project has begun its implementation phase.

The simplest example of a design pattern is probably the Singleton. This design pattern ensures a class only has one instance, and provides a global point of access to it.

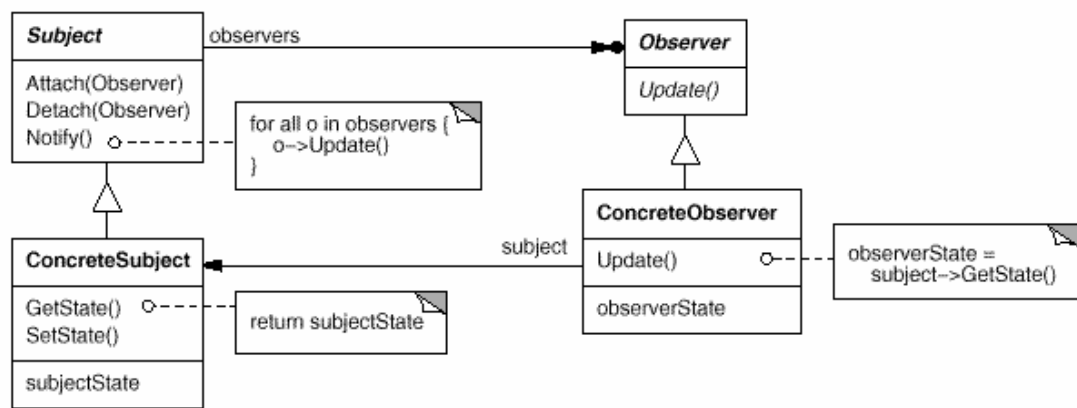


**Figure 1: Singleton Design Pattern Structure diagram**

The Singleton pattern from Gamma et al. [1995] has several benefits:

1. *Controlled access to sole instance.* Because the Singleton class encapsulates its sole instance, it can have strict control over how and when clients access it.
2. *Reduced name space.* The Singleton pattern is an improvement over global variables. It avoids polluting the name space with global variables that store sole instances.
3. *Permits refinement of operations and representation.* The Singleton class may be subclassed, and it is easy to configure an application with an instance of this extended class. One can configure the application with an instance of the class one needs at run-time.

4. *Permits a variable number of instances.* The pattern makes it easy to change one's mind and allow more than one instance of the Singleton class. Moreover, one can use the same approach to control the number of instances that the application uses. Only the operation that grants access to the Singleton instance needs to change.
5. *More flexible than class operations.* Another way to package a singleton's functionality is to use class operations (that is, static member functions in C++ or class methods in Smalltalk). But both of these language techniques make it hard to change a design to allow more than one instance of a class. Moreover, static member functions in C++ are never virtual, so subclasses cannot override them polymorphically.



**Figure 2: Observer Design Pattern Structure diagram**

A more complex design pattern from Gamma et al. [1995] is the Observer design pattern:

The Observer pattern lets one vary subjects and observers independently. One can reuse subjects without reusing their observers, and vice versa. It lets one add observers without modifying the subject or other observers.

Further benefits and liabilities of the Observer pattern include the following:

1. *Abstract coupling between Subject and Observer.* All a subject knows is that it has a list of observers, each conforming to the simple interface of the abstract Observer class. The subject doesn't know the concrete class of any observer. Thus the coupling between subjects and observers is abstract and minimal.

Because Subject and Observer aren't tightly coupled, they can belong to different layers of abstraction in a system. A lower-level subject can communicate and inform a higher-level observer, thereby keeping the system's layering intact. If Subject and Observer are lumped together,



then the resulting object must either span two layers (and violate the layering), or it must be forced to live in one layer or the other (which might compromise the layering abstraction).

2. *Support for broadcast communication.* Unlike an ordinary request, the notification that a subject sends needn't specify its receiver. The notification is broadcast automatically to all interested objects that subscribed to it. The subject doesn't care how many interested objects exist; its only responsibility is to notify its observers. This gives one the freedom to add and remove observers at any time. It's up to the observer to handle or ignore a notification.
3. *Unexpected updates.* Because observers have no knowledge of each other's presence, they can be blind to the ultimate cost of changing the subject. A seemingly innocuous operation on the subject may cause a cascade of updates to observers and their dependent objects. Moreover, dependency criteria that aren't well-defined or maintained usually lead to spurious updates, which can be hard to track down.

This problem is aggravated by the fact that the simple update protocol provides no details on *what* changed in the subject. Without additional protocol to help observers discover what changed, they may be forced to work hard to deduce the changes.

### 2.1.3 Conclusions

Overall, the above approaches have the following in common: they strive to reduce complexity and improve the flexibility of code. These are attributes of a well-designed system. Refactoring builds on these ideas, but adds in the extra constraint requiring that the behaviour of the refactored code is to stay the same after a refactoring has improved the design of existing code in order to make it more maintainable.

It is reasonable to suggest that at least some of the heuristics mentioned by the experts above could be implemented by refactoring. Therefore, if a piece of code breaks a certain heuristic then it can be considered a candidate for refactoring. The only requirement would be that the runtime behaviour resulting from the design changes would have to remain unchanged.

Code smells are targets for refactorings since they represent bad code design and they may lead to more un-maintainable code. The lack of heuristics (refer to the Appendix for examples) can be used to automatically check the design for potential design flaws and can therefore be seen as potential candidates for code smells.

When considering the sheer number of heuristics, design patterns, code smells and refactorings that are available, it makes sense to employ an IDE to manage and implement this knowledge more efficiently on a source code level. This will be discussed further on in the dissertation, specifically in Chapters 5, 6,

It is interesting to note that some design patterns may in fact violate a few heuristics, for very good reasons. A number of such examples can be seen in Molla's [2005] dissertation where a comparison between design patterns and heuristics is given. In such cases, it is up to the designer to decide which rule or patterns can be overridden.

For example the following violations are identified by Molla [2005] just for the Observer pattern alone. Molla [2005] provides a description of the pattern, the specific design heuristic violations, a reason as to why it violates certain design heuristics and comments:

### **Observer pattern**

Observer pattern describes how to establish relationships. In this pattern, the change in the state of an object can cause automatic updates in a list of dependents objects, that means each observer will query the subject to synchronize its state with all dependents state.

**Violate:** "Keep related data and behavior in one place" (see # 2.9 in Appendix A.3) and "Minimize the number of classes with which another class collaborates" (see # 4.1 in Appendix A.3).

**Reason for violation:** The motivation for this violation is to minimize the strength of the coupling between the subject and the observers.

**Comments:** When an object changes in the system, all its dependent objects are notified and updated automatically. For example, consider a spreadsheet program, when data insert or modify in that program, all the corresponding charts are change immediately, because document and chart class must know each other. The main idea of observer pattern is changing data in a window should be immediately reflect in all. This pattern also teaches how an object can tell other objects about events.

## **2.2 Software Maintenance**

Refactoring is used to make code easier to understand and maintain. In order to understand what maintainable code is one needs to understand what software maintenance entails.

### 2.2.1 Definition

The IEEE [1998] has the following definition for software maintenance:

1. Adaptive maintenance – “Modification of a software product performed after delivery to keep a computer program usable in a changed or changing environment”.
2. Corrective maintenance – “Reactive modification of a software product performed after delivery to correct discovered faults”.
3. Perfective maintenance – “Modification of a software product after delivery to improve performance or maintainability”.

Once a software system has been released, it is usually the case that a few bugs still exist and maintenance work still needs to be done. If the code was poorly designed it is possible to introduce more bugs into the system while trying to fix the new ones. This is especially the case for new developers that have no regard or knowledge of the existing design and its purpose.

Performance issues may also arise, once more users start using the system. These issues may be fixed by buying more machines to improve scalability, but often times it is cheaper to improve the software. When refactoring it is important to note that, it may cause performance issues. When improving design, it is sometimes the case that performance is increased. In fact, it normally decreases performance. Fowler [1999] argues that refactored code is easier to optimise. Usually a profiler will be used to profile a specific applications performance. A profiler will normally identify specific hot spots in the code, which take a long time to complete. In general, only 10% of the code takes 90% of the time to run. This means that if the code is refactored properly, then it will be easier to find the hotspots, since the methods should be small and well structured, opposed to a program that is not refactored. Smaller parts of the code are easier to tune.

The addition of new features also provides many challenges to software maintenance, as the system needs to be flexible enough to grow and simple enough to maintain. These issues will however be discussed in the next main sub-section (2.3) under the heading of software evolution.

### 2.2.2 When development becomes maintenance

Mantyla [2003] provides a literature survey on software maintainability in the context of refactoring and bad code smells as a whole.

From several references found in this dissertation, it is widely accepted that all changes to a product after it has been accepted by the client (or after its first public release) can be classified as maintenance.

When considering the above statement and the fact that most software would be maintained for much longer than the time it actually took to originally write the software, then it is clear that maintenance can become very costly, especially if the lifespan of a software product is expected to be long.

### **2.2.3 Maintenance cost factors**

Sommerville [2004] describes how maintenance costs are usually two to a hundred times greater than development costs, depending on the application. He attributes maintenance costs to the following factors:

1. Team stability - Maintenance costs are reduced if the same staff is involved with the maintained software for some time.
2. Contractual responsibility - The developers of a system may have no contractual responsibility for maintenance so there is no incentive to design for future change.
3. Staff skills - Maintenance staff are often inexperienced and have limited domain knowledge.
4. Program age and structure - As programs age, their structure is degraded and they become harder to understand and change. Structure degradation can occur through constant addition of enhancements and changing requirements.

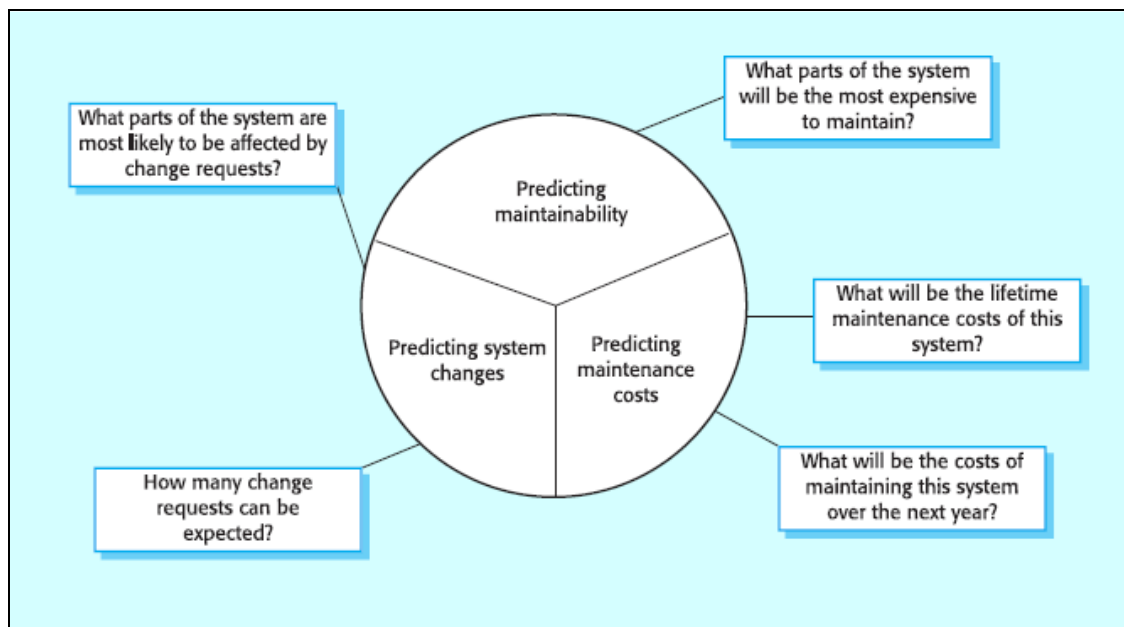
### **2.2.4 Maintenance and change prediction**

Sommerville [2004] mentions that maintenance prediction is concerned with assessing which parts of the system may cause problems and have high maintenance costs and adds the following facts about system maintainability:

1. Change acceptance depends on the maintainability of the components affected by the change;
2. Implementing changes degrades the system and reduces its maintainability;
3. Maintenance costs depend on the number of changes and costs of change depend on maintainability.

Sommerville [2004] goes on to say the following about change prediction:

1. Predicting the number of changes requires an understanding of the relationships between a system and its environment.
2. Tightly coupled systems tend to require changes whenever the environment is changed.
3. Factors influencing the difficulty of maintenance with regards to changes are:
  - Number and complexity of system interfaces;
  - Number of inherently volatile system requirements;
  - The business processes where the system is used.



**Figure 3: Maintenance Prediction**

Figure 3 above from Sommerville [2004] gives a good indication of the questions that need to be answered about predicting maintainability, system changes and maintenance costs.

As Sun Microsystems [1999] points out, code conventions are vital for a number of reasons:

- Eighty percent of the lifetime cost of a piece of software goes to maintenance.
- Hardly any software is maintained for its whole life by the original author.

- Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
- If one ships source code as a product, it should be ensured that it is as well packaged and as clean as possible.

Code conventions should be considered as a complementing technique for refactoring, as refactoring is also extensively used in the maintenance phase and leads to more maintainable code.

It is interesting to note that most modern IDE's can now format source code according to pre-defined code conventions. This allows code created by different programmers to be formatted in exactly the same way in regard to factors such as comments, indentation, white space, line wrapping etc.

If each programmer were allowed to have his or her own style, then it will be harder to read and understand the code when one is trying to adapt from reading one style to another. However, having one standardised set of code conventions to which programmers need to adhere, results in code which is easier to read and thus more maintainable. Therefore it is important to standardise on a code convention and to use a common IDE which can easily implement it. For example, at the time of writing this dissertation, Eclipse had built-in Java code conventions, which can be applied by formatting the code with the hot-keys CTRL-SHIFT-F.

### **2.2.5 Source Code Metrics**

Complexity metrics for object oriented programming go far beyond those of procedural programs. In the past the simple LOC metric was used to measure maintainability along with other measures such as McCabe's [1976] Cyclomatic complexity measure. Now measures include measures of complexity for control and data structures. Complexity also depends on object, method and module size. Many more complexity metrics will be discussed throughout this dissertation in order to locate possible design flaws and areas which will be particularly difficult to maintain.

Together with using the JDK compiler in conjunction with Eclipse to find unused code, IDEA's static code analysis techniques will be used to locate targets for refactoring. Mantyla [2003] argues that source code metrics are a reliable measure of maintainability and presents case studies by Bandi et al [2003] and by Li & Henry [1993a, 1993b] that prove this true for object-oriented source code metrics.

## 2.2.6 Conclusions

It is important to lay down the above facts as this dissertation will be talking about the maintainability of software. If maintenance is not properly done or if the product was not properly designed in the first place, then the product will be very hard to maintain and this will escalate maintenance costs and development effort needed for maintenance.

A common scenario in the commercial IT industry is for IT consultants to complete a software product, release it to the public or into production and then move onto another project, leaving the project to be maintained by other developers who have no prior knowledge of how the system was designed. A short handover process may take effect between the old and new developers, and a large portion of intellectual capital is still lost to the developers who leave with their experience.

The loss of intellectual capital impacts directly on increased maintenance costs caused through the lack of knowledge of the new developers. Not knowing how a system works will lead to a developer spending a lot more time figuring out a system, rather than actually being able to perform maintenance work.

Developers are paid for the time they spend during maintenance. Increased effort by new developer's results in increased costs and longer maintenance iterations than what was previously experienced with the original developers of the software product. This is because the original developers already had experience with the system since some of them actually wrote the code for it. The developers who first implemented the software product will be the ones with the most experience and knowledge of the software product and would therefore be the best potential candidates concerning any further maintenance tasks that would be required.

It is therefore important to realise the impact of losing the original developers. Loss of intellectual capital can also in effect directly reflect in the increased maintenance costs.

## 2.3 *Software Evolution*

Refactoring can be used to improve the design of existing code and thereby evolve the software. Sommerville [2004] explains that proposals for change are the driver for system evolution. Change identification and evolution continue throughout the system lifetime.

Evolution processes will depend on:

1. The type of software being maintained;
2. The development processes used;
3. The skills and experience of the people involved.

### **2.3.1 Design Evolution**

Tokuda [1999] notices that as applications evolve, so do their designs.

Designs evolve for many reasons:

1. Capability to support new features or changes to existing features.
2. Reusability to carve out software artifacts for reuse in other applications.
3. Extensibility to provide for the addition of future extensions.
4. Maintainability to reduce the cost of software maintenance through restructuring.

Tokuda observed that designs also evolve for human reasons:

1. Experience - Experienced employees may create better designs based on their domain knowledge.
2. New Perspective - New project members often have different ideas about how a design could or should be structured. Many organizations use a code ownership model, which empowers new employees with the ability to realize their ideas.
3. Experimentation - Arriving at a suitable design may require exploration of different design paths. Tokuda observed software cycles in which the principal development activity was experimentation with multiple designs.

Tokuda's [1999] research assesses the capabilities of refactorings for evolving object-oriented designs and attempts to determine if refactoring technology can be successfully transferred to mainstream programming languages such as C++ and Java.



### 2.3.2 The 8 laws of software evolution

Process evolution dynamics involves the study of the processes of system change. Lehman & Belady [1985] conducted empirical studies into the process evolution dynamics of many organizations and concluded the following eight laws:

1. Continuing change - A program that is used in a real-world environment necessarily must change or become progressively less useful in that environment.
2. Increasing complexity - As an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserve and simplify the structure.
3. Large program evolution - Program evolution is a self-regulating process. System attributes such as size, time between releases and the number of reported errors is approximately invariant for each system release.
4. Organizational stability - Over a program's lifetime, its rate of development is approximately constant and independent of the resources devoted to system development.
5. Conservation of familiarity - Over the lifetime of a system, the incremental change in each release is approximately constant.
6. Continuing growth - The functionality offered by systems has to increase to maintain user satisfaction.
7. Declining quality - The quality of systems will appear to be declining unless they are adapted to changes in their operational environment.
8. Feedback system - Evolution processes incorporate multi-agent, multi-loop feedback systems and one has to treat them as feedback systems to achieve significant product improvement.

### 2.3.3 Software reengineering

When considering that software, either reengineering involves the complete re-writing of a system or the restructuring thereof, one can draw the conclusion that refactoring may be used for re-structuring under the condition that behaviour is to be persevered.

According to Sommerville [2004], reengineering of sub-systems, which are frequently maintained, is appropriate in order to make the frequently used sub-systems more maintainable.

Reengineering is seen as an alternative to developing a new system when one considers the advantages:

1. Reduced risk - There is a high risk in new software development. There may be development problems, staffing problems and specification problems.
2. Reduced cost - The cost of re-engineering is often significantly less than the costs of developing new software.

### **2.3.4 Conclusions**

Design and software evolution are complex topics, which deserve to have tools dedicated to them in order to cope with change. Refactoring may be used in the re-structuring portion of re-engineering, which may also include total re-writing of code. Reengineering is often seen as a means of saving money, by not having to develop a new system.

## **2.4 Summary**

This chapter started with a description of design heuristics and design patterns to introduce the theme of design issues in relation to object-oriented programming. Three examples of design heuristics are given. Examples are also given into how one can use refactorings in order to introduce the absence of heuristics. The Singleton design pattern is introduced and its major advantages are identified. A brief discussion is given towards the conflicts that can appear between design patterns and design heuristics as was found to be the case with the Singleton design pattern.

Software maintenance and software evolution are also defined (in sections 2.2 and 2.3 respectively) to provide an understanding of the links between software design, maintenance, evolution and refactoring. This will serve as an introduction to the next chapter, which focuses on the answers to the many questions around refactoring. The following sections provide an understanding of the impact of refactoring on the many other phases contained in the software development life cycle.

## Chapter 3 Refactoring

### 3.1 Definition

Refactoring helps to improve the design of existing code and adds in the extra constraint requiring that the behaviour of the refactored code is to stay the same after a refactoring has improved the design of the existing code. This will in turn make the code more maintainable.

One definition of the word “factor” means to influence something. If X has an influence in Y then X is a factor in Y. To factor in X in situation Y, is to take X’s influence into account in situation Y. Thus, to re-factor situation Y, is to reconsider (the influence of X) in situation Y. To refactor means to re-influence something that already exists. There is however a deeper meaning in the computer science context. To refactor a program, one would need to apply one or more refactorings in such a way as to improve the design of the program and simultaneously preserve the programs behaviour. These refactorings should be performed with the intention of improved software maintainability and design. A simple example would be to refactor a long method into shorter methods with more descriptive names in order to improve the readability of the code.

### 3.2 Overview

One needs to know how refactoring fits into the software engineering process in order to see how to benefit from it.

Foote [1997] explains: “Building systems from the ground up is expensive and time consuming. Moreover, it is difficult to tell if they really solve the problems they were intended to solve until they are complete.”

Agile runs the risk of evolving architectural chaos. The absence of a grand design risks the development of the software equivalent of a squatter camp, instead of a beautiful urban complex. This is exacerbated if revisions / updates also take place in an uncontrolled fashion. Refactoring is one of the ways of controlling the updates and revisions in a disciplined fashion. If one considers that no major upfront design takes place in agile practises then it should be clear that without an opportunity to redesign certain elements of the code, architectural chaos could ensue. This is especially the case in environments where change is inevitable and refactoring allows those software changes to take place in a controlled manner.

Agile software methodologies like XP [Beck 2000] claim an inherent inflexibility in designs which do not evolve with the development of a system,

and that the resulting cost of change is high in such systems. Continuous design [Shore 2004], which utilises refactoring, allows one to add more flexibility into the design by not having a big upfront design, but rather by adding to the design as the need arises. Thus, the design will evolve as the code grows. With the coming of agile software development, there is a shift from building software to growing it. The process of refactoring can be used to achieve the growth.

Garlan [1994] mentions: “Object-oriented systems have some disadvantages. The most significant is that in order for one object to interact with another (via a procedure call) it must know the identity of that other object. This is in contrast, for example, to pipe and filter systems, where filters do not need to know what other filters are in the system in order to interact with them. The significance of this is that whenever the identity of an object changes it is necessary to modify all other objects that explicitly invoke it. “

Automated refactorings manage to soften the disadvantages of the object oriented architectural style, by modifying the part of the object that is invoked and all of the invokers as well. Thus, any dependencies are seamlessly resolved, as the refactoring tool will usually utilise an abstract syntax tree or code database, which will hold the crucial code dependencies that need to be updated whenever a refactoring is made. More examples of this will be shown later on in Chapter 6 to Chapter 8.

### ***3.3 Applicability***

Refactoring can be used in different contexts. For example to:

1. improve the design and code quality of existing systems;
2. evolve the design of systems dynamically through incremental development with practises such as test driven development and agile methodologies, thereby negating the need for a big upfront design;
3. understand how existing code works [Fowler 1999];
4. manage change in a software organisation [Beck 2000];
5. perform refactoring at an architectural level [Van Kempen 2005]; and
6. refactor with design patterns as targets of the refactoring [Kerievsky 2004].

### **3.4 Motivations**

Gamma et al. [1995] explain the role that refactoring plays rather well: “Once software has reached adolescence and is put into service, its evolution is governed by two conflicting needs: (1) the software must satisfy more requirements, and (2) the software must be more reusable. New requirements usually add new classes, operations, and perhaps even entire class hierarchies. The software goes through an expansionary phase to meet new requirements. This cannot continue for long, however. Eventually the software will become too inflexible and arthritic for further change. The class hierarchies will no longer match any problem domain. Instead, they will reflect many problem domains, and classes will define many unrelated operations and instance variables. To continue to evolve, the software must be reorganized in a process known as *refactoring*. This is the phase in which frameworks often emerge.”

Gamma et al. [1995] warn that when faced with design issues like encapsulation, granularity, dependency, flexibility, performance, evolution and reusability it is almost impossible to create a perfect design first time, especially when considering that these design issues often conflict when one is trying to decompose a system into objects.

Although refactoring fits best in agile methodologies, it is not a technique only used in iterative development scenarios. Large projects have design issues, which can be detected through the correct analysis tools found in common IDEs. These design issues are in turn, potential targets for refactorings.

Developer intervention in refactoring is more desirable in some cases, as there are still a few problems with refactoring automations for specific code smells. These problems will be discussed in detail later in this dissertation. IDEs have advance to the point where they are able apply refactorings on a project or global scale, through the click of one button. This allows several thousand code smells to be eradicated without the need to manually remove each one.

### **3.5 Problems**

Refactoring is a good practise, but there are a few troubles in applying refactoring in the real world. Here are four possible reasons Opdyke [1992] mentions as to why one might still not refactor programs:

1. One might not understand how to refactor.
2. If the benefits are long-term, why exert the effort now? In the long term, one might not be with the project long enough to reap the benefits.

3. Refactoring code is an overhead activity; one is paid to write new features.
4. Refactoring might break the existing program.

An organisation's management structure may be inflexible and may not allow the proper environment in which one would need to refactor. Such an environment would typically need unit testing to test the refactorings. Beck and Gamma [JUnit 2005] created a popular Java unit test framework.

If no unit tests had been done during the development of the project then the amount of work needed to implement these unit tests as part of a refactoring process, could outweigh the benefits received from such refactoring. Generally, one should only consider refactoring if unit tests were properly carried out in the first place. However, there have been recent suggestions for the automatic generation of test cases for a Java class file. This approach makes use of a symbolic Java virtual machine that can generate constraints representing the conditions for the control flow under consideration. No feasible tool has been developed yet, but there is future work planned by Müller et al [2006].

### **3.6 Tool Survey Motivation**

The aim of the tool survey was to see what the capabilities of the common open source and commercial IDEs were, in terms of their refactoring tool support. For the purpose of the survey, IBM's Eclipse and IntelliJ's IDEA was chosen. IDEs such as Netbeans, Jbuilder Enterprise 2005, CodePro's Studio, Jrefactory, CodeGuide and Jfactor were compared. Eclipse and IDEA had the most amounts of automated refactorings available and were therefore chose for the tool survey. This information was sourced from another refactoring plug-in Refactorit [2006].

The automated refactorings in most modern IDEs allow one to preview the resulting impact of the refactorings on code. The preview also shows warnings when code could be negatively impacted. The automated refactorings consider the entire project code and thus save one a lot of time compared to manual refactorings. The latter are error prone and rely on the compilation process to dig out errors caused by unchecked dependencies that might have been missed by the manual work of the programmer.

Currently, there is a choice between manual and automated refactorings. The programmer alone performs manual refactorings, which are more error-prone when compared to the automated refactorings available in IDEs. Automated refactorings are performed in such a way as to automate most of the code restructuring, so that limited input is needed from the developer and the chance of a successful refactoring will be high. In most cases all that is needed is for

the programmer to click on a specific piece of code and to select an applicable refactoring from a drop down menu. There is also a choice between searching for targets for refactorings and having the IDE do it through static code analysis or through a compiler refactoring approach. IDEA uses a static code analysis approach while Eclipse utilises a compiler refactoring approach.

These issues can make or break the ease with which refactoring is performed. The problem is to find practical, inexpensive solutions for enterprises that wish to evolve their designs through refactoring.

Building refactoring tools, which preserve behaviour, requires that a set of pre conditions be met before each refactoring is run and post conditions after the refactoring is run. These conditions are different for each refactoring and research by Roberts [1999a, 1999b] and Kataoka et al [2001] was done to ensure behaviour is indeed preserved in automated refactorings. Korman [1998] also provides a star diagram method, which is used for exploring, planning and carrying out refactorings of Java code. Opdyke [1992] defines pre and post conditions that hold for all of his refactorings in order to preserve behaviour.

### ***3.7 Code Review Motivation***

After using the different IDEs on a few code bases, the code review was performed with IDEA and Eclipse. Results from both of these reviews will be presented. A description of code smells that can be found is given as well as the refactorings needed to fix these issues. There are issues of productivity, which separate the two IDEs, and the weak and strong points will be highlighted in the chapters to come.

Through the code review, a detailed study of the code smells present in the JDK code base was given. Through the experience of the author, information is given on how to recreate the code review. An insight is given into the many different code issues and how to fix them. A classification method is proposed to be able to sift through the many code smells that are found. Classification enables us to focus on problems, which are more urgent, and to leave less severe problems for later.

An opportunity arose from the Java JDK to find and analyse code smells as well as their target refactorings. By reviewing all the different versions from 1.0.2 to 1.5.0, a more detailed understanding was gained of how code smells grow along with a large code base (Chapter 10). Chapter 10 also highlights the potential risks identified in the case where refactoring is not applied on an ongoing basis. These risks come in the form of code smells.

### **3.8 Summary**

A continuous design process [Shore 2004] allows one to evolve a design to make the maintenance phase less costly by adding more flexibility and reusability into the code as well as solving any design issues overlooked previously.

Refactoring plays an important role in continuous design and certainly has benefits, but the process must be adequately managed to ensure that it is done properly. The following chapter provides an insight into the roots of refactoring where old and recent research on refactoring is discussed.

In this chapter, a stronger refactoring background was given. Motivations for the tool survey and code reviews were discussed. The following chapter give more insight into the previous and current research work done in the research field of refactoring.

Chapter 5 describes the analysis methodology used to perform the code reviews.

Chapter 6 contains the Eclipse code review and Chapter 7 and Chapter 8 the IDEA code review. The IDEA code review was divided into two parts in order to distinguish the two main influences ([Fowler 1999] and [Kerievsky 2004]) that governed the code review.

The rest of the chapters include an IDE comparison, which forms part of the tool survey as well as a code evolution chapter, which forms part of the code review.



## Chapter 4 The roots of refactoring

This chapter indicates where refactoring first started and introduces important research recently conducted in the field.

### 4.1 Introduction

Opdyke was the first to coin the term “refactoring” and proceeded to introduce 8 basic refactorings, which were broken down into 26 low-level refactorings and 3 more abstract, high-level refactorings (made up of low-level ones). Although Opdyke’s work had examples in C++, Fowler later used most of these refactorings in his book [Fowler 1999], but used Java as the language of choice. Fowler provides a catalogue of over 70 refactorings for class design. He also explains how to identify hotspots for refactorings, which he calls “bad smells”.

Opdyke focused on refactoring object-oriented programs, since the underlying language structures were far richer than other languages. He mentions three cases where refactorings may be applied, namely to:

1. Extract re-usable components.
2. Improve consistency among components.
3. Support an iterative design approach.

Refactoring owes its firm foundations to research by Griswold [1991] and Opdyke [1992]. Griswold started to reason about program restructurings and how software maintenance (enhancement and repair) remains disproportionately expensive, relative to the expected cost of the required changes and the quality of the resulting software.

### 4.2 Founding Work

Griswold [1991] provides research to support that the size of the system tends to grow linearly with respect to the release interval number; the complexity of the system grows exponentially in relation to its size. This model for complexity equates to the cost of a change, since to make a correct change requires crosschecking for consistency over an exponential number of relationships.

Griswold [1991] further points out: Anti-regressive techniques or techniques, which aim to improve software maintainability, are ignored under financial and time pressures and, because they are not usually as psychologically satisfying

as progressive activities. As complexity increases so does the need for anti-regressive activity. This should be maintained only while complexity is not too large.

The idea is that automated restructuring will have constant cost with respect to release number, and thus constant complexity. Refactoring cannot reduce the number of existing faults, since meaning is preserved. If refactoring is targeted to the change, so that it is localized within a module, the number of layer elements and their pair-wise interactions of elements for the proposed changes is reduced drastically [Griswold 1991]. Simply put, if the change is localised to influence only one module, then the number of dependencies needed to be refactored decreases. Whereas the opposite case would be to influence a module with many dependencies, resulting in many more interactions that would need to be refactored. As automated refactoring tools are able to deal with more complexity, so the cost of such restructuring becomes linear with respect to complexity.

Opdyke's research focus was on using refactorings in order to support an iterative design approach. Three options come to mind when improving an existing system: Redesign, rewriting or re-structuring. Opdyke realised that generally re-structuring was the better choice. He describes a refactoring as a program restructuring that needs to provide meaningful abstractions. The abstractions are used to refactor the program easier to re-use and extend. He puts great effort into realising an automated approach to refactoring, but also warns that an automated tool should only be made to aid the designer to apply the refactorings correctly and not to decide which refactorings to perform.

### **4.3 Recent Research**

Related work from Tokuda [1999] stresses that being able to adapt software to change will result in lowering of project costs. The process is to refactor a system without changing its behaviour and improving its design so as to be able to extend the system easier. Tokuda [1999] focuses mainly on design evolution and stresses the fact that as an application evolves so too does its design. He goes on to say that, design evolution patterns need to be identified and that these patterns can be recognized as program transformations, which are automatable with object-oriented refactorings.

Cinnéide [2000] develops design pattern transformations by taking a pattern, decomposing it into its constituent mini-patterns, developing a mini-transformation for each mini-pattern, and finally specifying the complete transformation as a sequential composition of these mini-transformations. A mini-transformation here is seen as a refactoring, as only the design is improved, while behaviour remains constant. Similar work is done by Kerievsky [2004] who focuses on performing refactorings as targets for design patterns.

Mens et al. [2004] provide a thorough look at research in the domain of software refactoring and software restructuring. Five different categories were researched:

1. Refactoring activities supported:

The refactoring process consists of a number of distinct activities:

- Identify where the software should be refactored;
- Determine which refactoring(s) should be applied to the identified code smells;
- Guarantee that the applied refactoring preserves behavior;
- Apply the refactoring;
- Assess the effect of the refactoring on quality characteristics of the software (e.g., complexity, understandability, maintainability) or the process (e.g., productivity, cost, effort);
- Maintain the consistency between the refactored program code and other software artifacts (such as documentation, design documents, requirements specifications, tests and so on)

2. Specific techniques and formalisms that are used to support these activities:

A wide variety of formalisms and techniques have been, proposed and used to deal with one or more refactoring activities. Mens et al. [2004] discuss two such techniques in detail: the use of assertions (preconditions, postconditions and invariants) and the use of graph transformation. Next, Mens et al. [2004] discuss how formalisms can help us to guarantee program correctness and preservation in the context of refactoring. Finally, Mens et al. [2004] provide an indicative, but inevitably incomplete, list of other useful techniques to support refactoring activities.

3. Kinds of software artefacts that are being refactored:

Mens et al. [2004] argue that although contemporary IDEs limit support for refactoring to the source code only, refactoring can be applied to any type of software artifact. For example, it is possible and useful to refactor design models, database schemas, software architectures and software requirements. Refactoring of these kinds of software artifacts rids the developer from many implementation-specific details, and raises the expressive power of the changes that are made. On the other hand, applying refactorings to different types of software artifacts introduces the need to keep them all in harmony.

4. Important issues when building refactoring tools:

Although it is possible to refactor manually, tool support is considered crucial. Today, a wide range of tools is available that automate various aspects of refactoring. Mens et al. [2004] explore the different characteristics that affect the usability of a tool. More specifically, Mens et al. [2004] discuss the notions of automation, reliability, configurability, coverage and scalability of refactoring tools.

5. The effect of refactoring on the software development process:

Refactoring is an important activity in the software development process. Mens et al. [2004] discusses how refactoring fits into the processes of software reengineering, agile software development, and framework-based software development.

Mens et al. [2004] say that although commercial refactoring tools have begun to proliferate, research into software restructuring and refactoring continues to be very active, and remains essential to reveal and address the shortcomings of these tools. This dissertation aims to identify any such shortcomings in the tools selected.

Van Kempen [2005] shows the refactoring of the Pipe & filter architectures into the Blackboard and Client/Server architectures by mapping UML state charts into CSP (Communicating sequential processes). He also elaborates on refactoring distribution responsibilities of a Software Architecture expressed as an UML model.

## **4.4 Summary**

This chapter introduced us to where refactoring started originally. Attention was also paid to recent research done in the field. Specifically focus was put onto research performed by Mens et al. [2004], which was a wide study into five different areas of refactoring.

The following chapter introduces the analysis methodology. Eclipse and IDEA are used to find code smells in the Java Software Development Kit. A code smell classification method is introduced which will be used through the rest of the dissertation.

## Chapter 5 Analysis Methodology

To perform the analysis, a code review was performed on a popular and open-source code base. A project as large as the Java SDK (version 1.4.2) was likely to have detectable code smells. Eclipse 3.1.0 and IDEA 5.0 were used for analysis. All detected code smells are reported and discussed in two separate code reviews (one review for each IDE used). The first Eclipse review is covered in Chapter 6. This review deals only with unnecessary code. The IDEA review is covered in Chapter 7 and Chapter 8 and covers a wide array of code smells. In addition, an analysis on the evolution of the JDK from version 1.02 to version 1.5.0 was performed in order to see the effects of the software evolution of the JDK in terms of maintainability. Code smells will be used to measure the maintainability of the code. Some of the code smells are generated by object-oriented software metrics such as class and method metrics relating to size, coupling and complexity. A comparison of the two IDEs is given in Chapter 9.

This chapter includes refactorings for fixing code smells, ways of classifying various code smells by value and complexity, and refactoring productivity issues. Code smell classification is used as a guide for when it is best to refactor. Comparisons are made on how the two IDEs handle the analysis results as well as their capabilities in finding and removing code smells.

When considering complexity it is useful to learn from Griswold [1991]. He provides research to support the claims that the size of the system tends to grow linearly with respect to the release interval number; and that the complexity of the system grows exponentially in relation to its size. This model for complexity predicts the cost of a change, since to make a correct change requires crosschecking it for consistency for an exponential number of relationships.

### 5.1 JDK Code Base Statistics

The 1.4.2 version of the JDK will be used to perform a detailed analysis. An analysis of the entire JDK from version 1.02 to version 1.5 will be performed in Chapter 10 for the purposes of seeing how the code evolves from version to version.

Statistics were gathered on the number of files, number of source lines of code (SLOC, means no commented or blanks lines), number of commented lines of code, number of blank lines and the total number of lines of code in each code base. Thus Total number of lines of code (TLOC) = SLOC + Comments + Blanks.

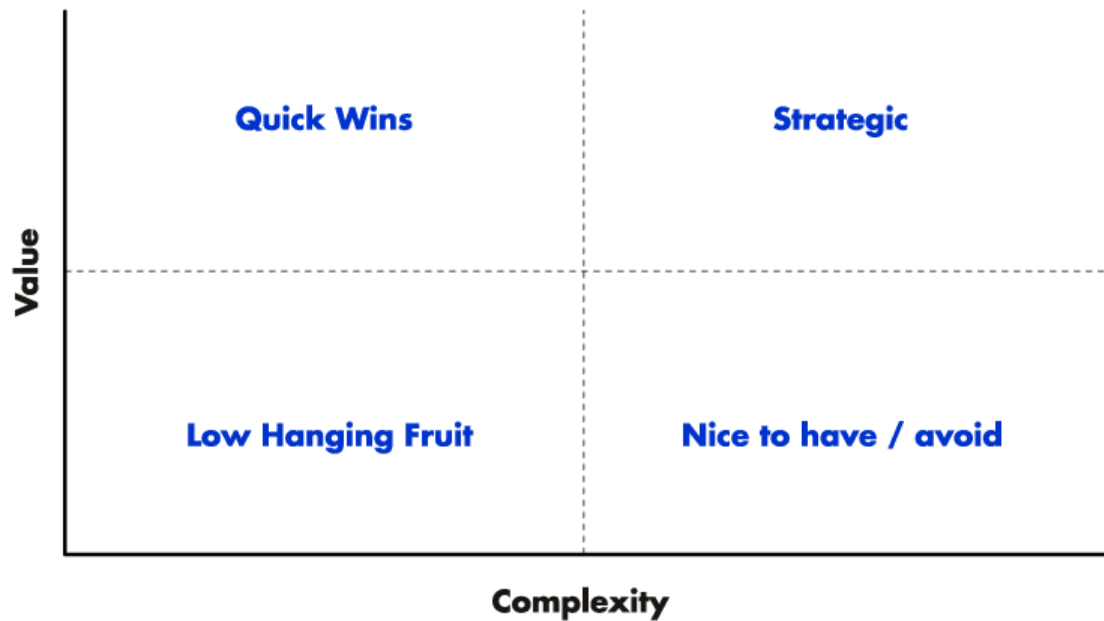
Version	Files	SLOC	Comments	Blanks	TLOC
1.4.2_03	4,141	563,073	569,285	161,504	1,293,862
		43.5%	44.0%	12.5%	100%

**Table 1: JDK 1.4.2 Code Statistics**

This JDK version has 457,801 lines of Javadoc comments, which is about 80% of all the comments. They are necessary as documentation generation relies on them as they form part of the code conventions for Sun Microsystems. The comment statistic in Table 1 includes both Javadoc and normal code comments.

## ***5.2 Classifying Code Smells***

The characterization of code smells and their associated refactorings in terms of complexity and value to the overall project can at best, be done at a notional, subjective level. There are no objective metrics available. Nevertheless, the view is taken in this dissertation that it is worthwhile and instructive to undertake such a classification, albeit somewhat subjective, as a guide in investigating the various refactoring facilities offered by the IDEs under consideration. However, one should consider that the more experience a person has with a code base, the less subjective his/her classifications become, because they have experienced where the design flaws in the system are and they can then determine what value will be brought from performing a refactoring on a focused part of the code base. In the this case, the author has little or no experience with the JDK code-base and is left with only the classification method as a means to decide which refactorings to use and which ones to avoid. To this end, consider Figure 4.



**Figure 4: A refactoring classification graph**

Picture (accessed 2006-04-01), from [www.silofx.com/migration](http://www.silofx.com/migration).

The four categories displayed in Figure 4: A refactoring classification graph are commonly used in financial circles to classify the value and complexity of something. The term “quick win” appears often in financial literature. Refer to this 4-quadrant graph throughout in order to classify the refactorings mentioned.

Complexity increases from left to right on the x-axis. Value increases from bottom to top on the y-axis. Four quadrants are identified in relation to complexity/value combinations. These are designated: Low Hanging Fruits (LHF); Quick Wins; Strategic; and Nice to have/Avoid.

Refactoring complexity (x-axis) can be related roughly to the amount of time taken to perform the refactoring. The term *time complexity* will be used throughout this dissertation to denote how long something will take to do. A task may be relatively simple, but if it appears in very large numbers then it will take a long time to complete and in this sense, it will be considered complex.

However, refactoring complexity is not merely limited to time considerations. The complexity is also determined by how much impact the refactoring has on the surrounding code base. If the refactoring change ripples through a large part of the code base then one can consider the refactoring to be complex. In addition, complex refactorings cannot be easily automated and complex code smells could require multiple refactorings in order to improve the design. Complexity is also high when the code is at risk of potentially breaking other

code when being refactored. For example, if an abstract method is changed then all of the abstract class' children will also be affected by this change.

The *value* (y-axis) of removing a code smell, though real, is difficult to measure in precise terms. It is generally recognised that there are significant financial rewards for code that is well structured and comprehensible and thus easily maintainable. However, in assigning value levels to various refactorings in this dissertation, no attempt is made to fully justify these levels in terms of financial benefits. Instead, value levels are generally chosen to reflect, to the authors best judgement, the extent to which the maintainability of the code is improved as a result of the refactoring.

Another measure of value would be to measure how often that piece of code is changed or maintained. Refactoring code that is used and changed more often will produce more value than refactoring code that no one uses. This is because refactored code is easier to maintain. This measure of value will not be used as there is no information telling us how often the source code is changed. If a versioning system would be able to provide this sort of information, then versioning information could be factored into the classification to be able to better focus on the most important parts of the code.

Note that this assessment of maintainability is not entirely subjective. In many cases, the value of a refactoring will be associated with a complimentary notion of code smell *severity*. This, in turn, is frequently tied to various severity settings that may be indicated in the IDE under investigation, these settings being used to identify and repair the various code smells.

To illustrate these notions, consider the presence of duplicate code as an example of a code smell. A change made to one version of the duplicated code generally has to be replicated on all the other versions of the duplicated code, a process that is notoriously error-prone. Refactoring this duplicate code smell often consists of gathering all of the duplicate code into one method. All instances of duplicate code are then deleted and the single new method is called at each point instead.

Now a refactoring that eliminates say 10 lines of duplicate code (a low severity code smell) would clearly have lower value than a refactoring, which eliminates 200 lines of code (a more severe code smell). To this extent, value derived from a refactoring varies with the severity of the associated code smell: if severity rises, then refactoring value also rises.

Of course, code smell severity may also be related to complexity of the associated refactoring. However, in general, this relationship is less clear-cut than the tie-in between value and severity. For example, in principle refactoring 200 lines of duplicated code into a method does not seem



significantly more complex than refactoring 10 lines of duplicated code into a method.

When referring back to Figure 4, one can focus on quick wins to get the most refactoring value in the shortest amount of time. The size of the JDK project (especially the later versions) makes it very difficult to refactor all of the code smells. It is therefore a better idea to be able to classify the code smells in such a way as to return the most refactoring value with the least amount of effort. Therefore, the code smells that can be fixed quickly and that hold high refactoring value should be addressed first.

In forthcoming chapters, various code smells will be classified into appropriate quadrants of Figure 4 based on severity settings in the IDE under investigation and/or on the informed but ultimately subjective judgement of the author. In Chapter 10, suggestions are provided for the classification of various code smells into one of the four quadrants in the above figure. These classifications should not be construed to be the final word on the matter. Rather, they are a starting point for further discussion and reflection.

### 5.3 Choosing Thresholds in IDEA

The best way to explain a threshold in IDEA is to give an example:

A good example is the generation of the code smell “method with too many exceptions” for each major version of the JDK. The threshold value determines the maximum value that is allowed to go undetected through the search, so if the threshold is set to five, then all values above five will be detected. The first threshold used was three, then four and then five. The following are the results:

Threshold Value	1.0.2	1.1.8	1.2.2	1.3.1	1.4.2	1.5.0
5	0	0	0	0	1	48
4	0	0	4	4	9	78
3	0	6	16	16	48	167

**Table 2: Code smell count for different thresholds**

Code smell thresholds will be used in IDEA to distinguish between severe and non-severe code smells. Note that most strategic refactorings will have thresholds available. Also, note that a threshold value is not always applicable to every code smell as this is application or IDE specific.

Each threshold generated a number of code smells. General heuristics were used to determine acceptable threshold values. When considering “the method with too many exceptions” code smell, the number of code smells generated

was used as a guide for selecting the thresholds for some of the code smell detection tools. If the number of code smells generated was close to zero, but not zero, then the threshold value was considered a high threshold value. All values lower than the highest threshold value will be considered lower higher threshold values. The threshold values used were first used on the 1.4.2 version of the JDK and were later reused for all of the other versions of the JDK.

The choice of thresholds depends entirely on the user of the IDE. It is recommended to choose a threshold that will filter out the least severe smells so that the most severe code smells will be displayed and fixed. It is the case that, in general, the more severe code smells occur less frequently than the less severe code smells. This can be seen by the frequency of code smells generated for each code smell using different thresholds.

The first column in the above table represents the threshold values. The other columns represent values falling under the specific JDK version. It should be clear that for each version, the greater the threshold value used, the smaller the number of code smells that is generated. For example, consider the highlighted number nine in the row with a threshold of four and JDK version of 1.4.2. When increasing the threshold to five, the frequency of code smells drops from nine to one. Decreasing the threshold to three causes the frequency of code smells to increase to 48.

One can therefore decide how strict one wants to be on the generation of code smells by manipulating the threshold values used during their detection. The number of code smells generated will change from project to project, so initially one should play around with a few random threshold values before the required output is generated.

It is difficult to determine the threshold values for each code smell, because the code quality differs with each project. In other words, it is possible to search for methods with over three exceptions in some projects and find numerous examples. In other projects, it might be the case that there are no methods with more than three exceptions and lower threshold values would need to be used in order to find bad code smells.

With the above being said, it should be clear that it is up to the user to determine what an acceptable threshold value is. The other code smells, which have configurable threshold values used for analysis, are ‘Nesting Depth’, ‘Long Method’, ‘Too many parameters’, ‘Large Class’ and ‘Inappropriate Intimacy’. In section 10.3 these code smells are discussed in detail. Low, normal and high threshold values were chosen for these code smells and the statistical details as well as a discussion are provided.

## ***5.4 Java As The Language Of Choice***

This dissertation views Java as the language with the greatest amount of refactoring support, mainly due to its open source nature, its popularity and the amount of refactoring support it has gained in the source of books, articles and automated tools available for it.

Refactorings have spread into many different languages, but Java remains the language with the most refactoring tool support, with C# being in a close second position. A list of recent refactoring tools is available for several languages [Fowler 2005].

## **5.5 Summary**

The analysis methodology includes the use of Eclipse and IDEA to find code smells in the Java Software Development Kit. A code smell classification method was introduced which will be used through the rest of the dissertation.

The following chapter will show how to capture code smells in the popular open source IDE from IBM, namely Eclipse. A code review will be performed on version 1.4.2 of the JAVA development kit.

## Chapter 6 Eclipse Code Review

This chapter describes how to reproduce the code review in Eclipse. All the checks for unnecessary code that can be carried out in Eclipse will be shown as well as its abilities to remove these code smells. The analysis techniques already present in Eclipse will be used to detect code smells.

If necessary, code segments are provided throughout this chapter to illustrate the general issue and its solution more clearly. Note that the results displayed are for the 1.4.2 version of the JDK. Please refer to Appendix A.1 and A.2 when looking for information on all of the code smells and refactorings in this review.

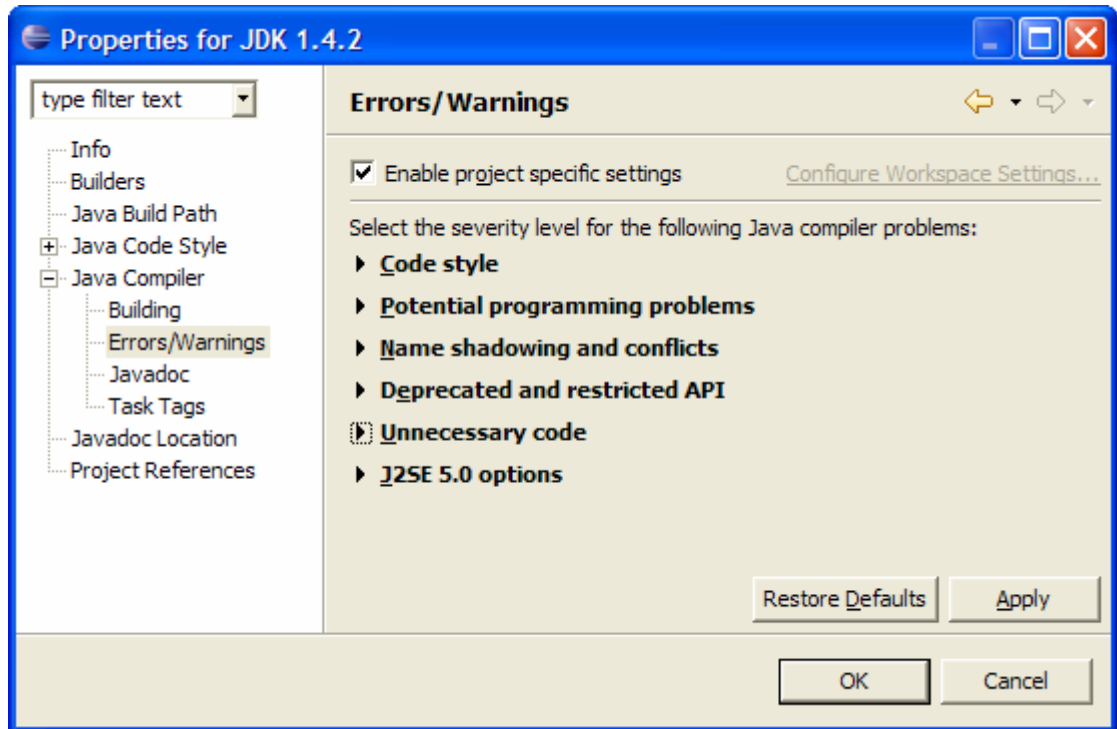
The only code smells considered for Eclipse are kinds of unnecessary code. While there are many other refactorings available in the tool, the only code smells detected are by the compiler, and involve unused code.

### 6.1 Unnecessary Code

Fowler [1999] does not refer to unnecessary code smells. Manyla [2006] uses a bad code smell taxonomy to categorise code smells. One of the code smell categories mentioned is the ‘Dispensables’ category, which contains ‘Dead code’ which translates to unnecessary code. This category can be found in Appendix A.3. The ‘Dispensables’ category contains many more examples of code, which can be removed. In this chapter, the focus will be on searching for dead code (code that is not being used) and using the facilities available in Eclipse to do this.

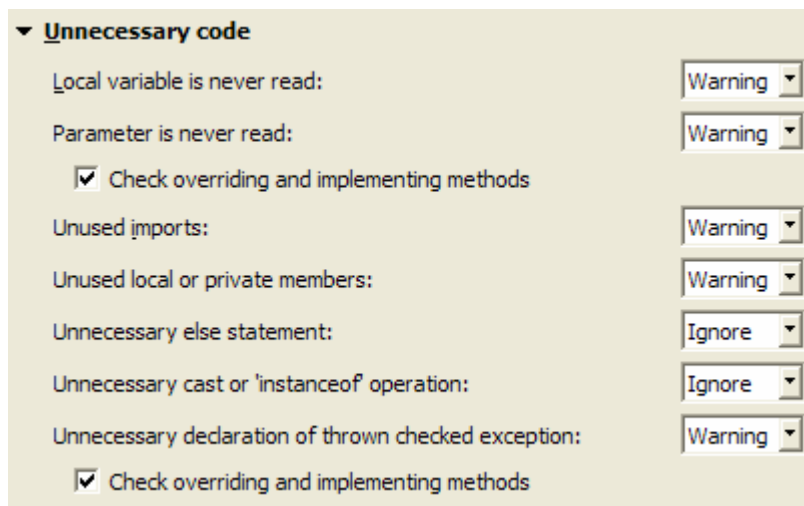
To be able to search for unnecessary code there are a few things that need to be setup under Eclipse. These setup issues and further advice will be given in this and the following chapter.

To access the compiler analysis tools of Eclipse, follow this menu flow:  
Project -> Properties -> Java Compiler -> Errors/Warnings -> Unnecessary code. This can be seen in Figure 5 on the following page.



**Figure 5: Error categories under Java compiler settings in Eclipse.**

Figure 6 below shows the settings for unnecessary code that have been used in this study. After code is built, (compiled) messages are received indicating what is wrong with the code, depending on how the Eclipse java compiler has been configured. Eclipse reacts to specific problems, by giving an error, warning or ignoring the problem. The developer in order of priority can configure these three options.



**Figure 6: Revealing unnecessary code in Eclipse.**

All unnecessary code warnings are caught during compilation and displayed under the problems tab, once the above settings are setup. Filtering the warnings returns statistics pertaining to a specific issue. The severity of this unnecessary code smell varies widely as it comes in many different forms.

Fowler [1999] does not discuss unnecessary code, although it can most definitely describe as a code smell. Unnecessary code can confuse programmers, as they will usually first try to figure out what the code does and then try to find how it connects to the rest of the software system. The result is that the code has no link to any other method i.e. it is not called from any method from outside the class. In some cases, the entire class in which unnecessary code exists could also be unnecessary.

Finding solutions to all of these problems follows a common pattern. First, one must enable the “Problems” filter to show all of the warnings (refer to Figure 8). Set filter configurations by clicking on the filter icon (refer to Figure 7) found on the right hand corner of the Problems tab window next to the window controls:



**Figure 7: Filter icon**

Tasks Problems Declaration Progress				
0 errors, 3,532 warnings, 0 infos (Filter matched 3,532 of 10,373 items)				
Description	Resource	In Folder	Location	
The import java.awt is never used	AbstractAction.java	JDK 1.4.2/javaw/swing	line 9	
The import java.awt.event is never used	AbstractAction.java	JDK 1.4.2/javaw/swing	line 10	
The import java.beans.PropertyChangeEvent i...	AbstractActionPro...	JDK 1.4.2/javaw/swing	line 9	
The import javax.swing.text.html is never used	AbstractButton.java	JDK 1.4.2/javaw/swing	line 23	
The import javax.swing.plaf.basic is never used	AbstractButton.java	JDK 1.4.2/javaw/swing	line 24	
The import javax.swing.border is never used	AbstractButton.java	JDK 1.4.2/javaw/swing	line 19	
The import java.util is never used	AbstractButton.java	JDK 1.4.2/javaw/swing	line 25	
The import java.util.Vector is never used	AbstractButton.java	JDK 1.4.2/javaw/swing	line 16	
The import java.awt.image is never used	AbstractButton.java	JDK 1.4.2/javaw/swing	line 11	
The import javax.swing.event.ChangeListener...	AbstractDocument...	JDK 1.4.2/javaw/swi...	line 16	
The import java.lang is never used	AbstractFilter.java	JDK 1.4.2/javaw/swi...	line 10	
The import java.awt.Dimension is never used	AbstractLayoutCac...	JDK 1.4.2/javaw/swi...	line 11	
The import java.util.Map.Entry is never used	AbstractMap.java	JDK 1.4.2/java/util	line 9	
The import javax.swing is never used	AbstractTableMod...	JDK 1.4.2/javaw/swi...	line 10	
The import java.util.Vector is never used	AccessibleBundle.j...	JDK 1.4.2/javaw/acc...	line 12	

**Figure 8: Warnings filtered by unused imports and sorted by Resource**

Each warning will be described by the description column (refer to Figure 8). One may filter by the description column to find one type of warning; otherwise, the problem tab will display every single warning.

String prefixes for all the Eclipse warnings are provided in the following sub-sections and are labelled as “Filter search prefix”. To illustrate this process more clearly, refer to Figure 9, which contains a filter search prefix containing

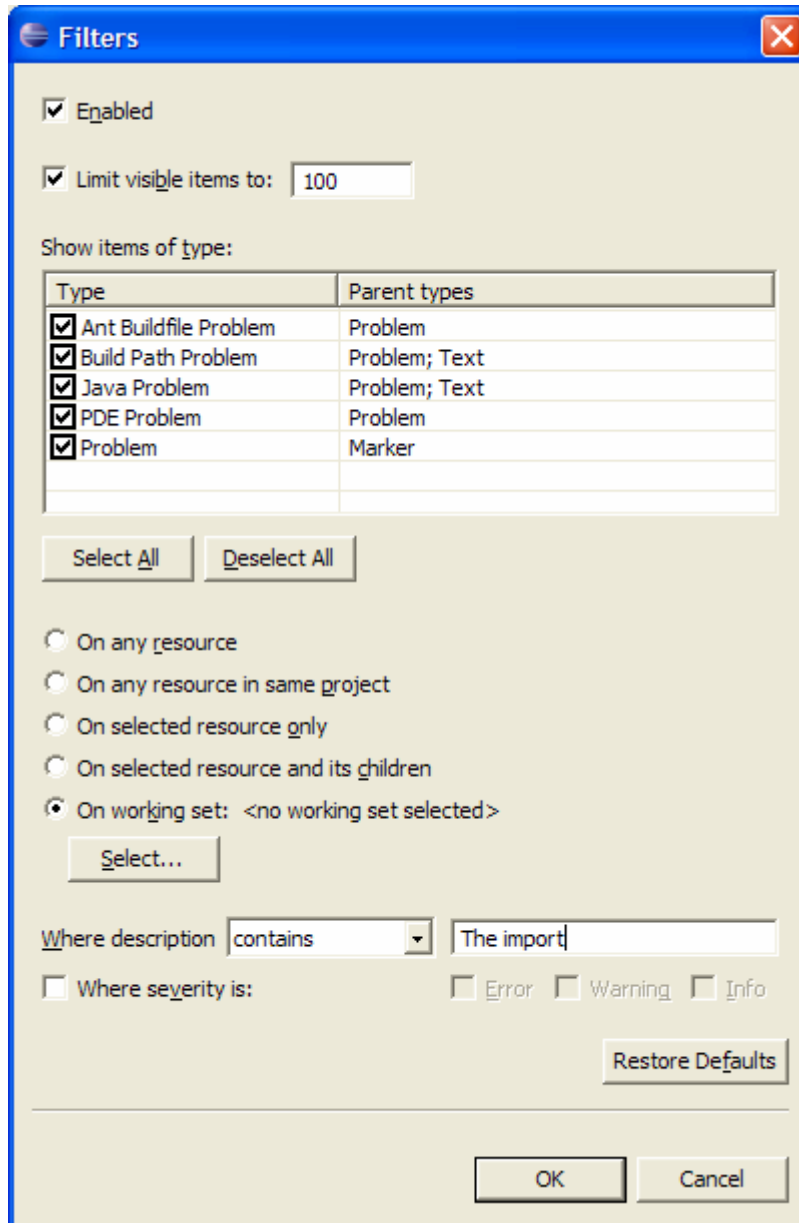
the string “The import”. This filter setting will search for strings in the description column that start with the letters “The import”. String prefixes; filter out the unwanted warnings by searching all the descriptions for an occurrence of the search prefix string. As an example, a (case-sensitive) string warning description for each warning is provided. Names used for demonstration purposes are denoted by the letter x.

After filtering, the warnings are sorted by ‘Resource’ and can apply a quick fix to each one. Figure 8 displays how a quick fix is about to be performed on the unused import in line 23 of the code. When right clicking on a resource (Java file) as shown in Figure 8, Eclipse comes up with a pop-up window, which allows a quick fix allowing import removal within the resource. This is quicker than having to open the physical resource and performing the refactoring.

The following sections discuss all of the code smells in JDK 1.4.2 that were detected using Eclipse. A means by which to replicate the analysis is provided and a discussion of the results is also given. If necessary, code segments are provided to illustrate the general issue and its solution more clearly.

Whenever a code smell appears in large numbers and the only possibility for removal is a manual one (i.e. removing each code smell, one by one) then one can say that the time complexity for the code smell is high and therefore the overall complexity is high.

It will be seen that most of the code smells were assigned a high complexity value, for this reason. For example, one cannot select all of the unused imports, right click on them and quick fix all of them. Instead, they have to be repaired one by one; therefore, a high complexity value was assigned to this code smell.



**Figure 9: Setting a filter search prefix**

### 6.1.1 Unused imports

Often programmers import a library to be used in a class, and may subsequently delete the class, but forget to delete the corresponding import. It is possible that an unused import points to a resource not existing in the production phase of the project, but present in the development phase. This will cause unwanted dependencies on resources that do not exist and compile time errors will result in the production phase.



If for example, if one codes `import java.sql.Timestamp`, but provides an import statement `import java.sql.*`, then the scope of the import statement is too large. When organising imports, Eclipse not only removes unneeded imports but will also remove wildcards and replace them with the actual classes that should be there. So after organising imports `import java.sql.*` becomes import `java.sql.Timestamp`, if this is the only class that one is using in the java.sql library.

Fixing this problem in Eclipse is difficult (Figure 8) and time consuming. The problem is the sheer volume of unused imports.

Sorting the warnings by the resource column (Figure 8) allows us to view how many unused imports exist in one project. By performing the “organise imports” quick fix, one is able to remove all unused imports from a class.

The unused imports smell rarely results in an inability to compile in the case of the JDK project analysed and receives a low value. Eclipse does not allow the imports to be organised in more than one file at a time. This smell has a large numbers of occurrences, which increases the time complexity involved in refactoring. However, it is possible (by activating the hot key CTRL-SHIFT-O) to remove all unwanted imports from a single file, but this is still not good enough from a productivity point of view.

Description: “The import x is never used”

Filter search prefix: “The import”

Files affected: 30% (1234)

Occurrences: 3532

Refactoring Complexity - High

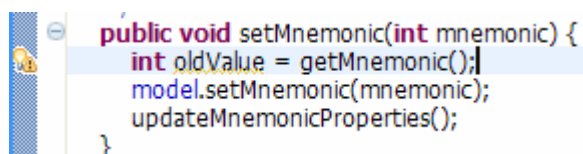
Value – Low

Classification – Avoid

Classification justification – Time complexity and number of occurrences is too high, therefore it is recommend to avoid this refactoring.

### 6.1.2 Unread local variables

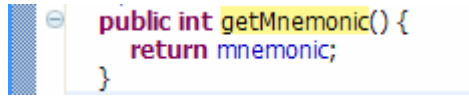
Proceed with caution when removing this smell in Eclipse. In Figure 10, notice *oldValue* in the following method from class *AbstractButton*:



```
public void setMnemonic(int mnemonic) {
    int oldValue = getMnemonic();
    model.setMnemonic(mnemonic);
    updateMnemonicProperties();
}
```

**Figure 10: Variable *oldValue* is local and unused**

Delete the *oldValue* variable without any behaviour change. Method *getMnemonic()* (see Figure 11) simply returns a value that is assigned to *oldValue*. It is safe to refactor this code by deleting the *oldValue* (assignee) variable and the assignor method *getMnemonic()*.

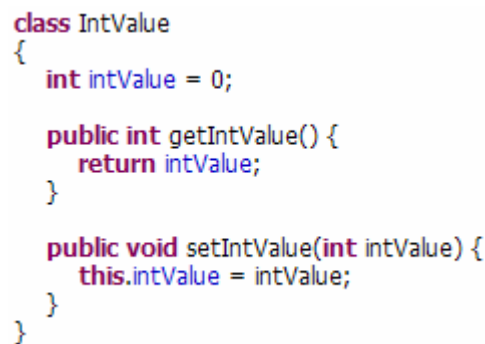


```
public int getMnemonic() {
    return mnemonic;
}
```

**Figure 11: Standard getter - getMnemonic()**

If an assignor method accepts an object as a parameter and changes the state of this object then one can no longer use this refactoring. If the parameter object state is needed in the operations following the assignor call, then deleting the assignor can change behaviour.

Now *getMnemonic(IntValue iv)* (Figure 13) (assignor) can change the value of variable *iv*. This can lead to different behaviour if one removes the assignor and *oldValue*. However, removing *oldValue* and still calling the assignor will be a behaviour preserving refactoring.

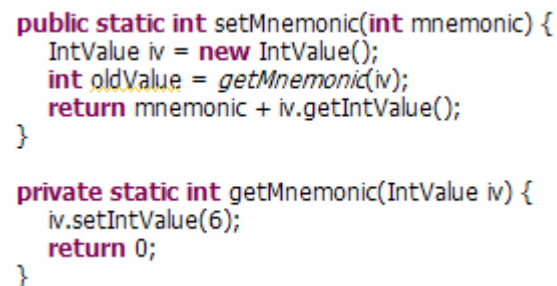


```
class IntValue
{
    int intValue = 0;

    public int getIntValue() {
        return intValue;
    }

    public void setIntValue(int intValue) {
        this.intValue = intValue;
    }
}
```

**Figure 12: A mutable int Wrapper class.**



```
public static int setMnemonic(int mnemonic) {
    IntValue iv = new IntValue();
    int oldValue = getMnemonic(iv);
    return mnemonic + iv.getIntValue();
}

private static int getMnemonic(IntValue iv) {
    iv.setIntValue(6);
    return 0;
}
```

**Figure 13: Side effects in getter method.**

As shown above getter, methods should not have side affects like that of the *getMnemonic(IntValue iv)*. Assignors containing side effects are not always getter methods. When clicked, a yellow triangle with an exclamation mark in the middle (not seen in Figure 13, but will appear just to the left of the

highlighted variable as in Figure 10) will produce recommendations from Eclipse on how to remove the unused local variable. Eclipse recommends removing the assignor and the assignee, which can prove troublesome.

Description: “The local variable x is never read”

Filter search prefix: “The local variable”

Files affected: 7% (295)

Occurrences: 517

Refactoring Complexity - High

Value – Low

Classification – Low Hanging Fruit. Avoid complex cases.

Classification justification – The value of these refactorings can be rather low. When taking into account the fact that the resulting refactoring can not preserve behaviour, one should rather leave this refactoring out. The complex cases mentioned here are those where an assignor could have a possible side effect. The resulting refactoring of such a complex case would give rise to a change in behaviour. The behavioural change would make the refactoring invalid.

### 6.1.3 Unread parameter

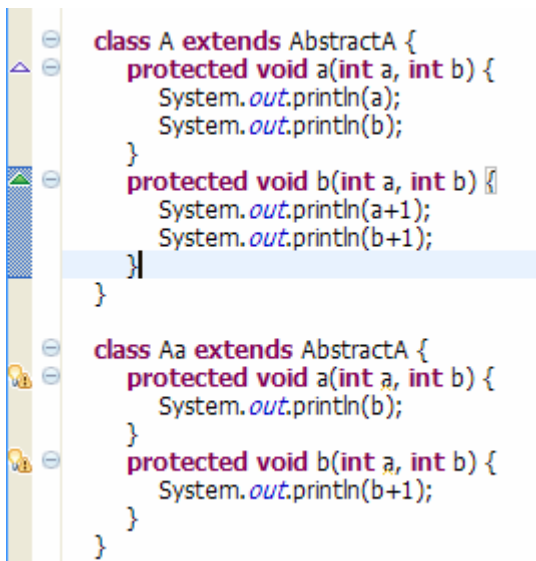
If the method concerned is *private* or *final* then the method may be refactored by removing the parameter (use the “change method signature” refactoring). The *final* keyword ensures that the method cannot be overloaded or inherited from a subclass.

It is common to give empty method implementations in an abstract class. There is a design pattern that advocates doing so, called Adapter [Gamma et al. 1995]. These methods may be overloaded in the subclass inheriting from the abstract class and may contain many parameters that are unused in the abstract class but used in the overloaded method present in the concrete class. Removing the unused parameters in the sub class will result in errors if both methods have an inheritance relationship. Eclipse does no checks for errors in hierarchies before highlighting a parameter as being unused.

```
public abstract class AbstractA {
    abstract void a(int a, int b);
    protected void b(int a, int b) {}

    public static void main(String[] args) {
        AbstractA a = new A();
        a.a(5, 0);
        a.b(5, 0);
        AbstractA aa = new Aa();
        aa.a(5, 0);
        aa.b(5, 0);
    }
}
```

**Figure 14: A mutable int Wrapper class**



```
class A extends AbstractA {
    protected void a(int a, int b) {
        System.out.println(a);
        System.out.println(b);
    }
    protected void b(int a, int b) {
        System.out.println(a+1);
        System.out.println(b+1);
    }
}

class Aa extends AbstractA {
    protected void a(int a, int b) {
        System.out.println(b);
    }
    protected void b(int a, int b) {
        System.out.println(b+1);
    }
}
```

**Figure 15: Children of AbstractA**

In Figure 14, method *a(...)* is defined to be abstract. Method *b(...)* is defined as an Adapter method. Classes inheriting from *AbstractA* need an implementation of method *a(...)* and can choose to overload method *b(...)*. In Figure 15, Class *A* and *Aa* both satisfy the contracts of *AbstractA*. Eclipse gives two warnings in the form of yellow triangles next to the two unread parameters in Class *Aa*. *AbstractA* is used in the main method to make use of the Adapter pattern and to make use of the abstract implementations. Eclipse does not pick this up and removing the two parameters has some serious behaviour changes.

Removing parameter *a* from *Aa:a(...)* results in Class *Aa* deviating from its inheritance contract with *AbstractA* and a compilation error. Similarly removing parameter *a* from *Aa:b(...)* creates this problem.

In the case where there is no direct inheritance relationship with the offending class, the class' offending method, may just have the unused parameter removed.

Description: “The parameter x is never read”

Filter search prefix: “The parameter”

Files affected: 20% (820)

Occurrences: 4890

Refactoring Complexity – High

Value – Low

Classification – Avoid Complex cases, all else are Low Hanging Fruit.

Classification justification – The complex cases mentioned here should be avoided, due to the way that Eclipse deals with the refactoring (essentially breaking the class hierarchies). All other simple cases where no class hierarchy relationship exists should be refactored. An unused parameter can be deleted without having any affect on the code.

### 6.1.4 Unnecessary throws clause (method or constructor)

This situation occurs when the declared exception x is not actually thrown by the method x() from type X.

In general, remove unused throws clauses in overloaded methods or methods implemented from abstract methods. Other methods have to catch or re-throw exceptions thrown from these methods. In projects where there are deep chains of method calls, this results in many unwanted catch and throws clauses. This smell adds unnecessary error handling complexity.

Figure 16 shows *AbstractA::b()* as having a unnecessary throws clause warning. The reason for this is that *AbstractA::b()* does not throw the *SomeException* exception. Removing this throws clause, causes *A::b()* in Figure 17, to no longer be compatible with the contract from *AbstractA::b()*. The same problem holds for abstract methods.

This issue is very similar to the one found when detecting unread parameters. Eclipse fails to recognise the deeper implications when removing an unused parameter or exception when the code smell appears in class hierarchies.

Most trivial cases of this code smell would include offending methods that do not necessarily occur in class hierarchies, but are called by many other methods throughout a project. Each of these methods needs to catch the exception, even if it is not thrown. This can result in large try-catch blocks scattered throughout a project with many unnecessary catch blocks and more confusing while performing error handling.

```
class SomeException extends Exception {
    private static final long serialVersionUID = 1L;
}

public abstract class AbstractA {
    abstract void a() throws SomeException;
    protected void b() throws SomeException {}

    public static void main(String[] args) {
        try {
            AbstractA a = new A();
            a.a();
            a.b();
            AbstractA aa = new Aa();
            aa.a();
            aa.b();
        } catch (SomeException e) { }
    }
}
```

**Figure 16: SomeException and AbstractA**

```
class A extends AbstractA {
    protected void a() { }
    protected void b() throws SomeException {
        throw new SomeException();
    }
}

class Aa extends AbstractA {
    protected void a() { }
    protected void b() { }
}
```

**Figure 17: A::b() throws SomeException**

If one is throwing more than three exceptions from a method, then whoever is calling that method must catch all of the exceptions. Dealing with too many exceptions can result in clumsy and bulky error handling code, especially in projects with long method call chains. An example call chain could be:

Layer 1	a()			
Layer 2	b()		c()	
Layer 3	d()	e()	f()	g()

**Table 3: Using layering to handle exceptions (Eclipse)**

It is common to divide a complex project into layers. These layers could also represent class hierarchies. A layer is assigned certain responsibilities. A parent layer would use its child layers to carry out its responsibilities. The call chain would include *a()* calling *b()* which calls *d()* and *e()*. After *b()* has completed, *a()* calls *c()* which calls *f()* and *g()*. Consider that all methods on Layer 3 throw two unique exceptions. This will mean that *a()* needs to deal with 8 exceptions.

Parent layers should wrap the child layer exceptions into more general exceptions in order to reduce too many catch statements at the top-most layers. A method declaring too many exceptions is dangerous when it is being used often and if it is deep in the call chain (like the methods in Layer 3). In such cases, create a superclass exception that can be extended by all of the offending exceptions. Only throw the superclass exception to the calling methods. If necessary, wrap the child exceptions manually by catching and wrapping them as the superclass exception. Perform any error handling in the catch statement used to wrap the exception. These special cases are Quick wins as their value is high.

Filter search prefix: “The declared exception”

Files affected: 6% (249)

Occurrences: 623

Refactoring Complexity – Medium to High

Value – Medium to High

Classification – Complex cases could lead to strategic design changes. Simple cases can also be seen as Strategic refactorings.

Classification justification – Simple cases would need to be cleaned up so that all code that is catching or re-throwing the exception is removed. Complex cases would require one to re-throw the bottom layer exceptions and wrap them into other exceptions.

## 6.1.5 Unused private members

This category is divided into 4 separate warnings. Their removal from the code base is very simple and involves a simple deletion performed automatically by Eclipse through clicking on a yellow triangle and following the recommended fix. A single filter search prefix “is never used locally” can be used to filter all of these warnings or alternatively search for them individually.

### 6.1.5.1 Unused private method

Unused private methods should be deleted. These methods cause confusion when they have names that lead us to believe that they do something meaningful. This happens when searching for specific verbs in a method name. Reading and understanding such methods is a waste of developer time.

Description: “The method x() from the type X is never used locally”

Filter search prefix: “The method”

Files affected: 2% (99)

Occurrences: 152

Refactoring Complexity - Low

Value – Medium

Classification – Quick Wins

Classification justification – This code smell can be easily removed. It has a small number of occurrences and reasonable value.

### 6.1.5.2 Unused private constructor

Unused constructors should be deleted. Unused constructors cause confusion and can lead us to believe that they do something meaningful.

Description: “The constructor X() is never used locally”

Filter search prefix: “The constructor”

Files affected: 0.1% (8)

Occurrences: 8

Refactoring Complexity - Low

Value – Medium

Classification – Quick Wins

Classification justification – This code smell can be easily removed. It has a small number of occurrences and reasonable value.

### 6.1.5.3 Unused private type/class

Unused classes should be deleted. They can lead us to believe that they do something meaningful.

Description: “The type X is never used locally”

Filter search prefix: “The type”

Files affected: 0.07% (3)

Occurrences: 5

Refactoring Complexity - Low

Value – High

Classification – Quick Wins

Classification justification – This code smell can be easily removed. It has a small number of occurrences and high value.



#### 6.1.5.4 Unused private member field

Unused private member variables should be deleted. They could cause unwanted confusion. Searching for their usages is a waste of time.

Description: “The field x is never read locally”

Filter search prefix: “The field”

Files affected: 5% (209)

Refactoring Complexity - Low

Occurrences: 362

Value – Medium

Classification – Quick Wins

Classification justification – Even though this refactoring does not hold a very high value, it has been decided that because there are not too many occurrences, that all of the smells will be quickly removed so as not to waste too much time and gain the most possible value.

## 6.2 Summary

This chapter provides the available code smell detection inside the Eclipse IDE. Each code smell and its corresponding refactoring is discussed. Possible problems with the automated refactorings are raised. All of the code smells detectable in Eclipse are related to unused code issues.

The above are not hard and fast rules that dictate what good code should look like. Under special conditions, there are many exceptions to the rules. Following these rules blindly under such special conditions can lead to mistakes. Experience helps us correct the compiler’s mistakes when detecting code smells. Eclipse is quick and requires little memory when detecting the code smells.

Our main concern is dealing with bulk quick fixes. A bulk quick fix entails the ability to fix a large number of code smells with just one action, instead of attending to each code smell separately. Eclipse is unable to process more than one file simultaneously. For some smells, this is good, as it requires the developer to think in case he makes a bad decision. Trivial smells in large numbers occurring in multiple files become tedious to remove and a bulk quick fix would make more sense to implement.

There is risk of breaking code (causing compile errors) when following the refactorings Eclipse suggests for some of the code smells. This is true for unread local variables, unread parameters and unused throws clauses.

Below are the summarised results of the Eclipse code review.

Name	Occurrences	Value	Classification
Unused imports	3532	Low	Low Hanging Fruit
Local variable is never read	517	Low	LHF to Avoid
Unread parameter	4890	Low	LHF to Strategic
Unnecessary throws clause	623	Low to High	LHF to Strategic
Unread private member field	362	Medium	Quick Wins
Unused local or private members	165	Medium to High	Quick Wins
Total	10089		

**Table 4: Eclipse code smell summary**

The following chapter will include a detailed code review that was done in IDEA. The code review spans version 1.02 to version 1.5.0 of the JDK.

## Chapter 7 IDEA Code Review – Part 1

### 7.1 Introduction

This chapter describes how to reproduce the code review in IDEA. Section 7.2 gives the IDEA settings needed to do the review. The remaining sections then consider various categories of code smells. Thus, section 7.3 considers code smells that relate to *duplicate variables*; section 7.4 considers code smells that relate to *inheritance*; section 7.5 considers code smells associated with *types*; section 7.6 looks at *abstraction*-related code smells; and section 7.7 addresses code smells related to *encapsulation*. Then, in sections, 7.8 and 7.9 *method metrics* and *class metrics* are considered, respectively.

IDEA's vast collection of code smell detection facilities is demonstrated, along with its abilities to remove these code smells in an automated fashion.

It should be noted that the code smells detected here are mostly from Fowler [1999]. The exceptions are:

- Chains using `instanceof`.
- Methods with too many exceptions declared, discussed in section 7.8.3.

The above exceptions are code smells identified by the author.

Please refer to Appendix A.1 and A.2 when looking for information on all of the code smells and refactorings in this review.

### 7.2 Code Review Methodology

IDEA 5 was used for this review as it dynamically detects a large number of code smells upon entering a source file. However, it was found that earlier builds of IDEA 5 have many bugs. It is therefore recommended that the newest available build be identified before using this IDE.

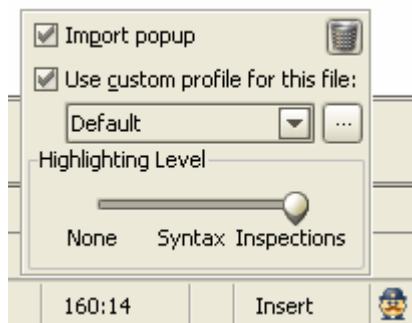
In the succeeding sections, various analysis techniques already present in IDEA are mapped to specific code smells. In each case, the analysis techniques are motivated for each code smell and advice is provided on how best to use these refactoring tools to one's advantage. Where necessary, Java code segments are provided to illustrate general issues more clearly.

Note that the code smells detected are for the 1.4.2 version of the JDK. High threshold values were used in order to generate a conservative number of code

smells. (Refer to section 5.3 in order to see more information concerning threshold values within IDEA.)

To configure the global analysis settings follow this menu flow:  
 File -> Settings -> Errors

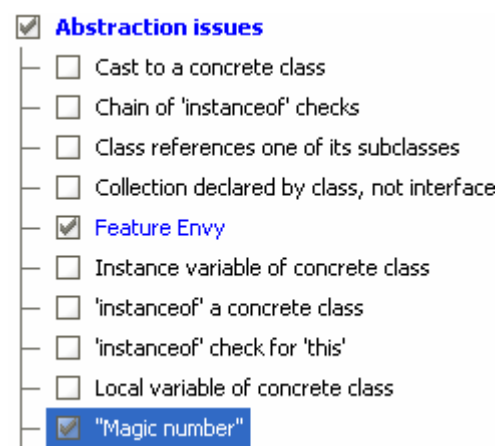
This leads to figure 18, which provides the screen for IDEA's file analysis settings. At this stage, the custom profile needs to be set. This provides the user with the ability to setup which code smells he wishes to have automatically detected. To set this profile, click on the man with the hat icon (bottom right hand corner). Configure the custom profile by clicking the ellipsis (...).



**Figure 18: IDEA file analysis settings**

Most menu flows can be accessed by right clicking on a package or class. Some code smells and refactorings mentioned here in this dissertation can be found in [Fowler 1999] & [Kerievsky 2004]. The rest are inspections that where mapped to code smells. The smells and their refactorings are discussed in detail in the remainder of this chapter and in Chapter 8.

The following figure shows how one would be able to select the 'Feature Envy' and 'Magic Number' code smells from IDEA's static file analysis tool.

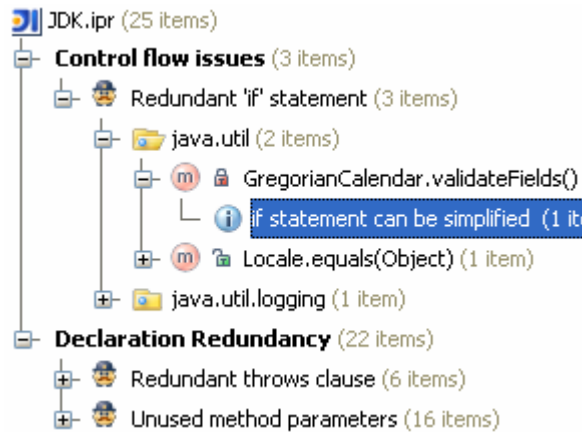


**Figure 19: IDEA code smell selection.**

When considering how to handle and refactor code smells please refer to

Figure 20. As shown in the figure below, clicking on an item will provide a problem description and show the offending code.

In some cases, right clicking on the man in the hat reveals a menu option, which allows one to fix all smells under that category with one click. This is useful for fixing many trivial smells without having to inspect each source file.



**Figure 20: IDEA Analysis Results**

The following section describes one of the most well known code smells (duplicate code). IDEA provides rather sophisticated mechanisms to detect this smell.

## 7.3 Locating Duplicates

*“Number one in the stink parade is duplicated code. If one see the same code structure in more than one place, one can be sure that oner program will be better if one find a way to unify them.”*

**Martin Fowler.**

### 7.3.1 Introduction

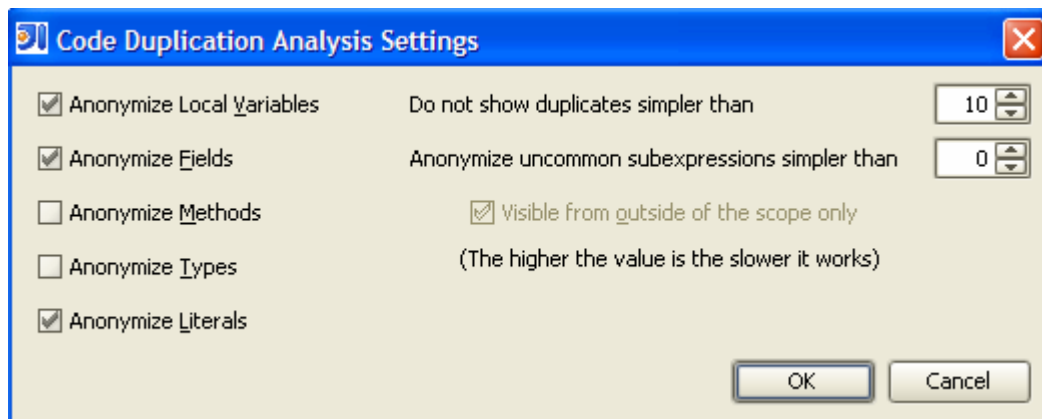
IDEA is able to find duplicates in code. Duplicate code is normally a result of copying and pasting code from one place to another. The duplicate code could be in the same class or in a different class entirely. The copy and paste pattern is usually an anti-pattern.

Although the precise definition of cost is not given here, it may generally be taken to mean the length of the string of text in the code that is to be matched with the rest of the text.

One can thus filter duplicates according to cost. The value 10 is the default cost threshold (see Figure 21). This value is input next to the literal “Do not show

duplicates simpler than”. This value is a cost representation that IDEA uses to calculate cost. Code sections with less code duplication will have lower cost and longer duplicate sections have higher cost.

Throughout this section, this “cost metric” will be used to measure the value of refactoring a duplicate section of code. High value will equate to high cost and low value to low cost.



**Figure 21: Duplicate location settings**

### 7.3.2 Anonymity

Anonymity allows duplicate code sections to differ by their variable names and string literals (refer to class names highlighted in blue in Figure 22). Anonymity is used to find more duplicate code sections. These variables can be member (field) or local variables. The trade-off is that more anonymity in duplicates results in slighter higher refactoring complexity, as refactoring this code requires the variable names to be changed and string literals to be parameterised. Changing variable names is however trivial for both local and member variables.

Duplicate code sections can differ by one or more string literals and can be refactored by extracting a method that accepts the string literals as parameters. Too many anonymous string literals become impractical to handle. Other refactorings like “Introduce Parameter Object” are used to handle excess parameters. These refactorings can become complex. This was found to be the case in some of the inspections that performed. Default anonymity settings are shown in Figure 21.

Selecting no anonymity options results in:

1. lower refactoring complexity and fewer duplicates and
2. high refactoring value and reduced search times.

No anonymity options were selected in the final analysis in order to focus on Quick Wins (refer to Figure 4). The number of duplicates returned with the default anonymity settings is a too large. For example, under these circumstances, the *java.util* package returned over 400 duplicate groups and the *java.awt* package, over 1000. Below, a duplicate search is shown with anonymous variable names in the class, *java.util.regex.Pattern*.

<code>int minL = info.minLength;</code>	1	1	<code>int minL = info.minLength;</code>
<code>int maxL = info.maxLength;</code>	2	2	<code>int maxL = info.maxLength;</code>
<code>boolean maxV = info.maxValid;</code>	3	3	<code>boolean maxV = info.maxValid;</code>
<code>info.reset();</code>	4	4	<code>info.reset();</code>
<code>prev.study(info);</code>	» 5	5 «	<code>yes.study(info);</code>
<code>int minL2 = info.minLength;</code>	6	6	<code>int minL2 = info.minLength;</code>
<code>int maxL2 = info.maxLength;</code>	7	7	<code>int maxL2 = info.maxLength;</code>
<code>boolean maxV2 = info.maxValid;</code>	8	8	<code>boolean maxV2 = info.maxValid;</code>
<code>info.reset();</code>	9	9	<code>info.reset();</code>
<code>next.study(info);</code>	» 10	10 «	<code>not.study(info);</code>
	11	11	

**Figure 22: Duplicate code found in *java.util.regex.Pattern***

When considering Figure 22, then a variable name could be made to be anonymous if *prev.study()* and *yes.study()* had both been changed to *genericname.study()*. The “generic name” in this example would be changed in order to make the duplication clearer and the resulting refactoring easier to perform.

In general, it was rather surprising to discover that a large amount of code duplication existed in the JDK as this code smell undoubtedly can lead to many maintenance problems. However, a detailed study into the results of the duplicates found in the entire code base is beyond the scope of this dissertation.

### 7.3.3 Solutions

Depending on context, code duplication can be repaired by the following refactorings:

1. Extract method – This refactoring will extract a method from a duplicated piece of code. After a method is extracted, the method can replace every piece of duplicated code, with a call to that method.
2. Parameterize method – This refactoring can be used in order to consolidate multiple methods that do the same thing and only differ by certain values.

3. Extract Class – This is when a new class is extracted from another class, by moving the relevant methods and fields from the old class into the new one.
4. Extract Superclass – Is when one has two classes that share similar features. The similar features are consolidated into one class i.e. the superclass.
5. Form Template method – This is when one has a process that performs similar steps in a specific order, but the steps are different.
6. Introduce parameter object. – This is when one has a number of parameters that appear inside a method and one chooses to reduce the parameters by putting them into an object. This will simplify the method, by reducing the number of parameters.

Please refer to Fowler [1999] for more precise definitions of the above refactorings. 'Replace Method with method object' may also be used when it is difficult to use the 'Extract Method' refactoring.

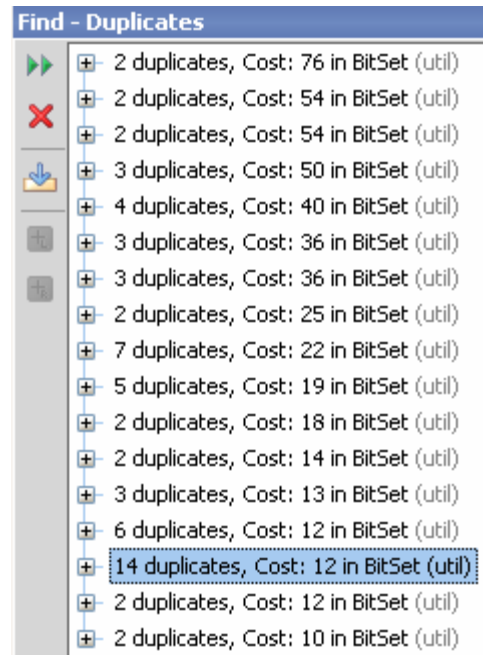
An effort should be made to refactor the duplicate groups that carry a high cost in favour of low cost duplicates in order to add as much value to the refactorings. This is especially the case if one is not considering removing all of the code smells due to their overwhelming size.

Figure 23 shows the IDEA listing of duplicate code groups found in the *java.util.BitSET* class. In each case, the number of duplicates in a group as well as the cost for that group is indicated. A duplicate group can contain two or more duplicate code sections. In the example provided, the number of duplicates a group had varied between 2 and 14.

Note that after an analysis on duplicates, IDEA does not supply a detailed statistical summary of the number of duplicate groups identified. Manual counting was performed to find out that there were 17 duplicate groups in this particular run.

As noted in Section 7.3.1, one can set a cost threshold to filter out duplicates. Increasing the cost threshold will decrease search times. However, the example provided here illustrates that such filtering could miss a group with a large number of low cost duplicates (blue highlighted line in Figure 23). Seventeen duplicate groups in *java.util.BitSet*.). Large duplicate groups are excellent candidates for the 'extract method' refactoring, because IDEA can find all of the duplicated code sections and extract them into a single method.





**Figure 23: Seventeen duplicates groups in java.util.BitSet.**

In general, the more a piece of code is duplicated, the more severe the code smells, especially if the code section is large. Consider a developer (inexperienced with a code base) changing one method in order to add the desired behaviour. This developer could be unaware that there are, say, 13 other similar such methods which also need to be changed. If all of these methods were extracted into one single method, then the developer need not have to search for the duplicate code.

Another thing to note is that the four duplicates found in one group with a cost of 40, together have a cost of 160. This total cost is the highest total cost out of all of the duplicate groups. Due to the automatic way that IDEA handles the refactoring of duplicates, the number of duplicates in a group can become transparent when considering the complexity of dealing with too many similar duplicate code sections.

### 7.3.4 Dealing with Anonymity

When one avoids anonymity, far fewer results are obtained. The results are also far easier to refactor. IDEA can only sort duplicate groups by cost (as is shown in Figure 23 **Error! Reference source not found.**). It does not provide detailed results of its duplication analysis. For large projects, one needs to manage the code duplication information to be more productive. Customised methods where used to organise the analysis results IDEA provided for code duplication. These methods are described in more detail in the following paragraphs.

In general, if a code base is allowed to grow without any refactoring, code duplicates will be more common in larger packages and classes than in smaller ones. The “Physical source lines of code” (SLOC) metric can be easily used to measure package size accurately. For this code review, a physical source code analyser was used to group and sort the JDK packages by their SLOCs.

In total, there are over 230 packages in the JDK. When performing the duplication analysis by a package-by-package basis, it was found that 60% of the version 1.4.2 of the JDK code could be covered by only analysing 25 packages. This information can be read from the output of a good source code analyser. It makes more sense to only refactor the larger packages in order to cover as much of the code as possible without having to analyse all of the packages. Thus grouping the JDK by packages was a useful abstraction.

As a proof of concept, the duplicate code in the class *java.awt.image.DirectColorModel* was refactored. The focus was on the high cost or large duplicate groups. For example, when considering Figure 23, consider the first few items with the highest cost. In terms of larger groups, one may only consider the 14 duplicates (highlighted in blue) due to the large number of groups. The ‘Extract Method’ refactoring was applied for roughly 20 minutes. As a result, this reduced the 715 physical source lines of code by 150 lines. In essence, twenty-one percent of the code was removed from this class.

IDEA allows the user to selectively specify which duplicated sections are to be abstracted into a single method. Consider a group that contains 10 duplicate sections, 10 sections each with the same code. If the ‘Extract Method’ refactoring operation is performed on one of these sections, then IDEA first highlights each of the remaining nine duplicate sections sequentially, asking the user whether to refactor the particular duplicate code section into the method. This results in higher productivity when performing this refactoring.

When dealing with a large project, one must be able to decide which classes are worth refactoring. From experience, it can be shown that one saves time by analysing the largest classes in a package first. Larger classes will have a greater likelihood of containing code duplication. If time permits, duplicates over multiple packages may be located to check if any smaller classes with a lot of code duplication have been missed. However, cross package duplication is discovered at the expense of longer analysis times.

### 7.3.5 Final Analysis and Conclusion

Due to the long analysis time, a package-by-package analysis was done— i.e. no cross-package analysis was performed. No anonymity options were used and a cost threshold of 40 was selected. All of the located duplicate groups can be classified as Quick Wins, due to their high value and low refactoring

complexity. In total, the JDK contained 68 duplicate groups either that had a cost of at least 40 or that had a reasonable group size (like the group of 14 highlighted in blue in Figure 23). The counts for the respective packages are given below.

java: 15	javax: 19
com: 25	org: 9

**Table 5: Duplicate group occurrences**

As pointed out above, repair of these duplicates by one of the various refactoring methods was considered to be beyond the scope of this present limited study.

Packages affected: **JDK wide**

Occurrences: **68**

Refactoring Complexity – Low

Value – High

Classification – Quick Win Group

Classification Justification – The number of duplicates found here was highly dependent on the input parameters, which only filtered out the duplicates with the highest cost, and therefore the highest value. No anonymity was used in order to make the refactoring of the duplicates as simple as possible. Due to the ease with which this code smell is refactored, the refactoring complexity can be termed as being low.

## 7.4 Inheritance Issues

### 7.4.1 Refused Bequest

*“Subclasses get to inherit the methods and data of their parents. However, what if they do not want or need what they are given? They are given all these great gifts and pick just a few to play with.”*

**Martin Fowler.**

Choosing to inherit from a superclass that does not truly reflect the operations or data needed in the subclass is generally bad design. The refused bequest smell shows subclass methods that override (and thereby ignore) the implementations in their superclass.

If many Refused Bequest smells occur in a class, then one should consider refactoring it with the ‘Replace inheritance with delegation’ refactoring.

Fowler [1999] uses the *java.util.Stack* class in the JDK as an example of when to use ‘Replace inheritance with delegation’. The motivation is that *Stack* should only implement *push()*, *pop()*, *size()* and *isEmpty()*. Instead *Stack*,

inherits everything from Vector and reuses only a little, ignoring the rest. Vector should be made a member and be used for delegation. The inheritance from Vector can now be removed.

In simple hierarchies, one can use the refactoring ‘Push Down Field/Method’ to fix one or two “Refused bequest” smells. This will ensure that only common code stays in the superclass and that the subclass specific code is pushed down from the superclass to the sub-class. In complex class hierarchies, it is difficult to perform the ‘Push Down’ refactoring, as there could be multiple dependencies on the data or method to consider. It is important to design the hierarchy properly and catch any such smells as early as possible before the hierarchy gets too complex and inflexible to refactor.

Packages affected: 110

Occurrences: 5485

Refactoring Complexity – High (due to too many occurrences)

Value – Low to Medium

Classification – Avoid

Classification Justification – Due to the high complexity of refactoring this code smell, it is classified as being a code smell to avoid.

## 7.5 Type Code

When talking about “type code” one talks about code that is executed based on decisions taken relating to an object’s type. These decisions are taken within conditional statements, such as *if* and *switch* blocks. It will be indicated why the presence of type code is often considered a code smell. An explanation will also be given in order to explain a new type of way to encode type code inside a class and will show what the best way is to encode type code information using the ‘Type Safe Enum’ pattern.

The detection of type code is discussed along with a number of methods that will be used to remove type code. The major subsections involve the replace with class, subclass, state, strategy and command pattern refactorings in order to refactor this code smell.

### 7.5.1 Detection of Type Code

Quite a few instances of type code were found. To search for instances of type code, one needs to find *switch* or *if* statements with too many branches in the control flow category of IDEA’s analysis section. High threshold values need to be set for the number of branches in order to get high refactoring value. A default value of ten was used for both *if* and *switch* statements.

## 7.5.2 Replace with Class

This section is used as an introduction to dealing with type information. Use this refactoring if a class is encountered that contains type information that does not affect the behaviour of the class. IDEA cannot detect this code smell. The following subsection explains this in further detail.

```
public class Car
{
    public static final int AUDI = 0;
    public static final int BMW = 1;
    public static final int MERC = 2;

    int type;

    public Car (int type)
    {
        this.type = type;
    }
}
```

Figure 24: Embedded type information

```
class CarType
{
    public static final CarType AUDI = new CarType(0);
    public static final CarType BMW = new CarType(1);
    public static final CarType MERC = new CarType(2);

    public static final CarType values[] = {AUDI, BMW, MERC};

    int carType;

    private CarType(int carType)
    {
        this.carType = carType;
    }
}
```

Figure 25: Typesafe Enum pattern

Figure 24 shows a typical way for storing type information. Figure 25 shows a type safe solution. Note that the private constructor is used so that no instance of the class can be made. This will allow the class to be used for the purposes of accessing the public static variables exclusively. This is exactly the case with the CarType class. This is the “Type Safe Enum” pattern [Tiger 2004], which mimics C style enumerations (see *values[]*). Figure 26 shows a new *TypeSafeCar* class which can only be created when it receives an object instance of class *CarType*. Compile time type safety is now implemented.

```
class TypeSafeCar
{
    CarType carType;

    public TypeSafeCar(CarType carType)
    {
        this.carType = carType;
    }
}
```

**Figure 26: A type-safe class**

```
public class TestCars
{
    public static void main(String[] args)
    {
        Car car = new Car(Car.BMW);
        TypeSafeCar typeSafeCar = new TypeSafeCar(CarType.AUDI);
    }
}
```

**Figure 27: Demonstration of Car and TypeSafeCar**

If one will be moving to the JDK 1.5 version of Java in the future, rather wait before attempting this refactoring. JDK 1.5 allows easier handling of enumerations through its new language features. It would be simpler to refactor the code so that it will support the new type features that are supported in the new version of the JDK rather than the older one shown in the figures above. The result would be far more maintainable code. Tiger [2004] explains a better approach for dealing with enumerations in the 1.5 version of the JDK and sample code is provided below:

```
enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }
enum Rank { DEUCE, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN,
JACK, QUEEN, KING, ACE }
```

```
List<Card> deck = new ArrayList<Card>();
```

```
for (Suit suit : Suit.values())
    for (Rank rank : Rank.values())
        deck.add(new Card(suit, rank));
```

```
Collections.shuffle(deck);
```

The above code demonstrates how much easier it is to deal with types when using enumerations in the 1.5 version of the JDK. The same code in an earlier version of the JDK would take many more classes and work to achieve the same affect.

### 7.5.3 Replace with Subclasses

Use the ‘Replace with Subclasses’ refactoring when the type information influences class behaviour. If it does not then use the ‘Replace with Class’ refactoring.

```
public static Car createCar(int carType)
{
    if (carType == Car.AUDI)    return new Car(Car.AUDI);
    else if (carType == Car.BMW) return new Car(Car.BMW);
    else if (carType == Car.MERC) return new Car(Car.MERC);

    throw new IllegalArgumentException("Type not supported.");
}
```

Figure 28: Factory method for the Car class

```
public static TypeSafeCar createTypeSafeCar(CarType carType)
{
    if (CarType.AUDI.getCarType() == carType.getCarType())
        return new TypeSafeCar(CarType.AUDI);
    else if (CarType.BMW.getCarType() == carType.getCarType())
        return new TypeSafeCar(CarType.BMW);
    else if (CarType.MERC.getCarType() == carType.getCarType())
        return new TypeSafeCar(CarType.MERC);

    throw new IllegalArgumentException("Type not supported.");
}
```

Figure 29: Factory method for the TypeSafeCar class

This refactoring can be used as soon as conditional logic determines object creation based on type information. All type information that is encoded in the if statements is replaced with subclasses.

There are many possible ways to create the object with conditionals (including switch statements). The name of the ‘Replace Conditional with Polymorphism’ refactoring is confusing when dealing with creational logic that needs type information. The solution is to move the creational logic to the inheritance structure. The child instances will be used to customise behaviour.

### 7.5.4 Replace with State, Strategy or Command Pattern

Use this refactoring when the behaviour or state can change during an objects lifetime. This can make its difficult or impossible to use ‘Replace with Subclass’. State and Strategy are very similar and as a result, the refactorings will be very similar. As with “Chains of instanceof”, the solution is to ‘Replace Conditional with Polymorphism’.

Interestingly enough Kerievsky [2004] includes three refactorings that depend on checks for conditional logic. All of the following refactorings are based on refactoring to patterns.

- 1) Replace Conditional Logic with Strategy
- 2) Replace State-Altering Conditionals with State
- 3) Replace Conditional Dispatcher with Command

1) and 2) have already been discussed in this section. Use 3) to replace conditional logic deciding what actions to execute (based on an action name or type) with polymorphism (Command Pattern).

It is interesting to note that the GUI related classes in the *com.java.sun.swing.plaf.gtk* and *javax.swing.plaf.basic* packages implement the *PropertyChangeListener* interface. All of the implementations of this interface contained the following general code structure, where *propertyChange(...)* is the method to implement:

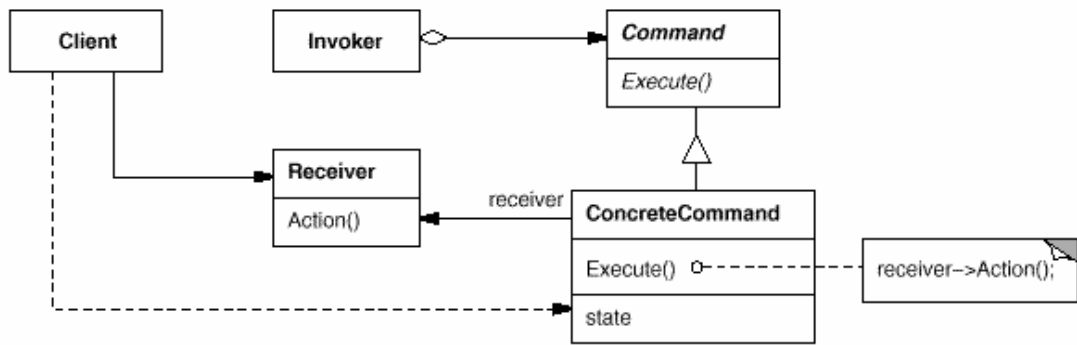
```
public void propertyChange(PropertyChangeEvent evt)
{
    String propertyName = evt.getPropertyName();

    if (propertyName.equals("Action 1"))
    {
        // action code
    }
    else if (propertyName.equals("Action 2"))
    {
        // action code
    }
    // more actions handled by more else-if statements
}
```

**Figure 30: Replacing a conditional dispatcher with Command**

A project may follow certain anti-patterns (as show in Figure 30), which would benefit from the ‘Replace Conditional Dispatcher with Command’. So in effect, each *if* and *else if* statement will be handled by the Command Pattern. The command Pattern is used to replace each action with a class that inherits from a subclass as show in Figure 31 below. The result is that the Command Pattern decouples the object that invokes the operation from the one that knows how to perform it.





**Figure 31: Command pattern structure diagram**

### 7.5.5 Chains using Instanceof

IDEA can check for chains of *instanceof*, *if* statements. It searches for chain lengths with a minimum size of two, i.e. an *if* statement with one or more *else* branches. The check ensures that all the conditionals contain the *instanceof* keyword. A chain length parameter cannot be set, but it still helps us to identify cases, which could benefit more from polymorphism. The reasoning behind this is that the conditional logic can be replaced by one polymorphic method call, which will delegate the behaviour to the correct subclass or instance.

An *if-else* chain that chooses behaviour, based on object type, is usually a sign of poor design. If practical, one should remove the *if* checks and implement the behaviour into subclasses using polymorphism. Polymorphism decides which object one is going to call the method on and removes the need for the *if-else* structure. In general, larger *if-else* chains containing large amounts of behaviour will contain more refactoring value and complexity.

The solutions depend on whether or not one is working in a creational or a behavioural context as discussed in the above section. Use the above sections to identify the type code contained in the conditionals and refactor accordingly if necessary.

Packages affected: 46

Occurrences: 173

Refactoring Complexity –High

Value – Low to High

Classification – Strategic.

Classification Justification – The act of adding polymorphism to substitute the *instanceof* chains is not a simple task. Thus, the complexity of this refactoring is high. However, the value of the refactoring puts the code into a more user-friendly object-orientated style, which can be maintained much easier than normal *if-else* branches.

## 7.5.6 Conclusion

Fowler [1999] covers refactorings on all of the type code smells. The only new refactoring is that of ‘Replace Conditional Dispatcher with Command’ from Kerievsky [2004]. A definition of the type code smell is given in order to group all of the refactorings that solve type code smells.

All the type code refactorings (except two) can be generalized with one solution, namely ‘Replace Conditional with Polymorphism’. The only exceptions are when refactoring creating logic with ‘Replace with Class’ and ‘Replace with Subclasses’. The low-level refactorings needed are identical across all the other high-level refactorings. The only differences are the contexts across which these refactorings are applied.

## 7.6 Abstraction Issues

This section will describe abstraction issues. These are issues pertaining to the object-orientated principle of having an abstract or superclass used to sub-class other classes, which share common data and behaviour.

### 7.6.1 Feature Envy

This smell occurs when an object method, say `X.m()` is too “interested” in the data of another object, say `Y`. For example, object `Y` can be passed via a parameter to `X.m()`, or object `Y` can be instantiated locally in `X.m()` and then many methods of `Y` are called from `X.m()`. This is an indication that functionality is in the wrong class. It is as if object `X` is overly interested in the methods (features) of object `Y`, and `X` is therefore deemed the “offending” object.

IDEA’s approach to detecting this feature envy code smell is to indicate that the smell occurs when `Y`’s methods are called more than 2 times from methods of the offending object `X`. IDEA does not allow for any value other than two to be used in this context.

When running IDEA against the JDK1.4.2, the results were as follows:

Packages affected: 26

Occurrences: 58, 219

Refactoring Complexity - High

Value – Low to Medium

Classification – Avoid

Classification Justification – The value of these refactoring is low and the large number of occurrences justifies putting this code smell in the avoid category.

Intuitively, one could say that the greater the number of external method calls from the offending object, the more severe the smell. It would have been useful to be able to configure the threshold value used to identify feature envy in IDEA, but this was not possible.

Feature envy may be eliminated by relying on the 'Extract Method' refactoring to isolate access to the offending object. The selected code will therefore be extracted into a method. After performing the 'Extract Method' refactoring use the 'Move Method' refactoring to move the extracted method to the offending class. This will cause the behaviour, to be moved to the calling class.

## 7.6.2 Magic Numbers

Magic numbers are literal numeric constants used without declaration. They can result in code whose intention is extremely unclear. Errors may result if a magic number is changed in one code location but not another. Good coding practice therefore traditionally requires that meaningful constants are assigned to magic numbers. One should focus on selecting meaningful names for magic numbers, especially in instances where the same number is to be used several times.

The numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 0L, 1L, 0.0, 1.0, 0.0F and 1.0F are not reported by IDEA. The results obtained when running IDEA against JDK1.4.2 were as follows:

Packages affected: 125

Occurrences: 13003

Refactoring Complexity - Medium

Value – Low to Medium

Classification – Avoid, due to too many occurrences.

Classification Justification – The value of these refactorings is low and the large number of occurrences justifies putting this code smell in the avoid category.

IDEA has a refactoring named 'Introduce Constant' which can solve this problem. Folwer [1999] has a 'Replace Magic Number with Symbolic Constant' refactoring to solve this problem.

As an example, consider the magic numbers that IDEA found in the *java.math.MutableBigInteger* class method *binaryGcd(int a, int b)* :

Magic number '0xff' at line 1060
Magic number '0xff' at line 1069
Magic number '0xff' at line 1083
Magic number '0xff' at line 1089
Magic number '0x80000000' at line 1080
Magic number '0x80000000' at line 1080

**Table 6: Magic Number example**

The solution to this magic number problem is to replace 0xff and 0x80000000 with descriptive constants. In the first case, IDEA will ask if one wants to replace the matching four magic numbers in the same method automatically, and similarly in the second case.

## 7.7 Encapsulation

Encapsulation is an important part of the object-orientated style. It was designed to limit access to class data. All data should be private and only accessible from get and set methods. There are however a few exceptions, such as constants, which are discussed later in the following section.

### 7.7.1 Public Field

If a field is not a constant and is, declared public then anyone can access this field from a class. This will violate the encapsulation law, which is designed for object-orientated programs.

In general, public fields should only be used when defining constants. A constant should be static and final, otherwise IDEA will report it as a smell. It must be static to represent the fact that it must hold one value across each class instance. It must be final so that nothing can write to it. It is a Java convention that the constant name should be written in higher-case letters.

To refactor this code smell, use the 'Encapsulate Field' refactoring.

Packages affected: 48

Occurrences: 700

Refactoring Complexity – Trivial

Value – Low

Classification – Low Hanging Fruit.

Classification Justification – This smell occurs a fair deal. However, it is very simple to refactor. This is why it is classified as low hanging fruit.

## 7.8 Method Metrics

This section describes code smells relating to methods. Methods are the building blocks of classes and are used to encapsulate behaviour. The code smells, which arise from methods, are therefore important, albeit more difficult to fix, since one has to rely on manual refactorings in order to fix them. Manual refactorings are by their nature, more error prone to fix when compared to automatic refactorings. In order to detect these smells methods metrics are used.

### 7.8.1 Long Method

*“The object programs that live best and longest are those with short methods”*  
**Martin Fowler.**

One should decide what a long method is. Usually a method hides a lot of code that does many different things in a sequence. A long method should be broken down into more methods, which do less and are easier to understand. It is difficult to find a piece of functionality in a long method, than it is to find it in a short method; therefore, short methods are more maintainable.

If one only considers strategic refactorings, then one should attempt to refactor methods longer than 74 lines. To keep things in perspective, a 30-line method without any braces and comments can fit onto a 19” screen in IDEA. Analysis does not count comments or blank lines.

The ‘Extract Method’ refactoring can be used to shorten long methods. Long methods can contain complex nested conditional statements. Organise the code for easy understandability and navigation. Extracting short methods from the long ones helps us do this. The “Poor Method Composition” section gives more information on proper method composition.

To refactor this smell use the ‘Replace Nested Conditional with Guard Clauses’ or ‘Decompose Conditional’ refactorings. Fowler [1999] has an entire chapter on simplifying conditional statements. He also recommends ‘Replace Temp with Query’, ‘Introduce Parameter Object’, ‘Preserve Whole Object’ and ‘Replace Method with Method Object’ for this smell.

Packages affected: 104

Occurrences: 1264 (default setting of 30 physical source lines of code)

Method Length	30	>= 50	>= 75	>= 100
Occurrences	1264	450	195	92
Complexity	Low	Medium	High	High
Value	Low	Medium	High	High
Classification	LHF	LHF	Strategic	Strategic

**Table 7: Long Method analysis results**

Classification Justifications – Methods shorter than 75 lines were seen as low hanging fruit due to their low complexity and value. Methods over this limit were seen as strategic refactorings due to the high complexity and value involved in the refactorings.

## 7.8.2 Long Parameter List

A method may have too many parameters in it. The result will be that this method will be difficult to use and understand. The method can also potentially be a long method due to the large amount of parameters that it holds.

Methods with long parameter lists are difficult to understand. Due to its complexity, this smell has no automatic refactoring suggested. Introducing a parameter object takes work. Unless the object already exists, one needs to build the object from the calling side to use it in the method.

Sending a large Serializable object across the network as a parameter could be uneconomical if all of its fields were not needed. This is not a problem if one is only sending it by reference. If one decides to construct smaller versions of objects to send them over the network efficiently as parameters, then one must also consider the class explosion problem.

To refactor this smell use the ‘Replace parameter with method’, ‘Preserve Whole Object’ and/or ‘Introduce Parameter Object’ refactorings.

Packages affected: 76

Occurrences: 681 (default setting of five parameters lines)

Parameters	>= 5	>= 8	>= 11
Occurrences	681	108	35
Complexity	Medium	Medium	High
Value	Low	Medium	High
Classification	Nice to have	LHF	Strategic

**Table 8: Long Parameter List analysis results**

Classification Justifications – One can see that as the number of unused parameters increase, the higher the complexity and value becomes. Threshold

values were chosen to filter out items so as to ensure that only a few high valued items could be classified as being strategic.

### 7.8.3 Too Many Exceptions

Dealing with too many exceptions can result in clumsy and bulky error handling code, especially in projects with long method call chains. The issues and solutions with this code smell are discussed below.

If one is throwing more than three exceptions from a method, then whoever is calling that method must catch all of the exceptions. An example call chain could be:

Layer 1	a()			
Layer 2	b()		c()	
Layer 3	d()	e()	f()	g()

**Table 9: Using layering to handle exceptions**

It is common to divide a complex project into layers. These layers could also represent class hierarchies. A layer is assigned certain responsibilities. A parent layer would use its child layers to carry out its responsibilities. The call chain would include *a()* calling *b()* which calls *d()* and *e()*. After *b()* has completed, *a()* calls *c()* which calls *f()* and *g()*. Consider that all methods on Layer 3 throw two unique exceptions. This will mean that *a()* needs to deal with 8 exceptions.

Parent layers should wrap the child layer exceptions into more general exceptions in order to reduce too many catch statements at the top-most layers. A method declaring too many exceptions is dangerous when it is being used often and if it is deep in the call chain (like the methods in Layer 3). In such cases, create a superclass exception that can be extended by all of the offending exceptions. Only throw the superclass exception to the calling methods. If necessary, wrap the child exceptions manually by catching and wrapping them as the superclass exception. Perform any error handling in the catch statement used to wrap the exception. These special cases are Quick wins as their value is high.

Packages affected: 15 (for more than three exceptions)

Occurrences: 48

Exceptions	$\geq 4$	$\geq 5$	$\geq 6$
Occurrences	48	9	1
Complexity	Medium	Medium	Medium
Value	Medium	Medium	High
Classification	Strategic	Strategic	Strategic

**Table 10: Method with too many exceptions analysis results**

Classification Justifications – One can see that as the number of unused exceptions increases, the higher the complexity and value becomes. Threshold values were chosen to filter out items to ensure that all of the refactorings were classified as being strategic.

## 7.9 Class Metrics

Class metrics are used to pick up code smells pertaining to classes. Inappropriate intimacy deals with how highly coupled a class is and large classes deal with classes that have too many instance variables. Refactorings and discussions are given around these two code smells.

### 7.9.1 Inappropriate Intimacy

*“Sometimes classes become far too intimate and spend too much time delving in each others' private parts. We may not be prudes when it comes to people, but we think our classes should follow strict, puritan rules.”*

**Martin Fowler.**

To measure inappropriate intimacy, highly coupled classes were identified. Highly coupled classes lead to one change needing to be propagated through to all dependent classes. This makes maintenance difficult when an automatic refactoring is not available.

The ‘Move Method’ and ‘Move Field’ refactorings can be used to solve this issue. Use the ‘Change Bidirectional Association to Unidirectional’ refactoring if possible. ‘Extract Class’ or ‘Hide Delegate’ could also be appropriate refactorings depending on the context.

Packages affected: 140

Occurrences: 1277



Dependencies	$\geq 10$	$\geq 30$	$\geq 50$
Occurrences	1277	155	29
Complexity	Low	Medium	High
Value	Low	Medium	High
Classification	Avoid	Strategic	Strategic

**Table 11: Inappropriate Intimacy analysis results**

Classification Justifications – One can see that as the number of dependencies increases, the higher the complexity and value becomes. Threshold values were chosen to filter out items to ensure that only a few high valued items could be classified as being strategic.

## 7.9.2 Large Class

*“As with a class with too many instance variables, a class with too much code is prime breeding ground for duplicated code, chaos, and death.”*

**Martin Fowler.**

The number of member fields was used as an indication of a class’ size. A class with a large number of member fields could be suggestive of a class that is trying to do too much. However, constant fields were ignored in the analysis. Note that using the number of methods only (i.e. ignoring member instance fields) as measure of whether or not a class is too big can be misleading, since a class may have only a few methods, but these methods could be rather long.

To refactor this code smell use the ‘Extract Class’ or ‘Extract Subclass’ refactoring to group the local variables.

Packages affected: **68**

Occurrences: **278 (for 10 local variables)**

Variables	$\geq 10$	$\geq 20$	$\geq 30$
Occurrences	278	105	44
Complexity	Low	Medium	High
Value	Low	Medium	High
Classification	LHF	Strategic	Strategic

**Table 12: Large Class analysis results**

Classification Justifications – One can see that as the number of instance variables increases, the higher the complexity and value becomes. Threshold values were chosen to filter out items to ensure that only a few high valued items could be classified as being strategic.

## **7.10 Summary**

A large number of code smells were classified and their appropriate refactorings were given, which are defined by Fowler [1999]. This was done using IDEA in the context of the Java SDK code base.

The next chapter deals with refactoring to pattern techniques [Kerievsky 2004]. Kerievsky [2004] is cited where applicable while the author of this dissertation identifies the rest of the code smells. The author provides new complementing code smells. The respective solutions or refactorings are also provided for the newly identified code smells. Note that not all the refactorings implement a pattern as the result of the refactoring. Mention is also given on how coupling and cohesion can influence what should be refactored.

## Chapter 8 IDEA Code Review – Part 2

### 8.1 Introduction

This chapter continues the IDEA Code Review. In contrast to the previous chapter, however, there are almost no references to code smells mentioned by Fowler [1999]. Instead, some concepts have been taken from [Kerievsky 2004] and this work is cited appropriately. The author has provided the rest of the contribution in terms of unused code smells and the discussion on coupling and cohesion and its resulting relationship with refactoring.

This chapter deals with a number of issues such as poor method composition, creational issues, control flow issues and unused code. Please refer to Appendix A.1 and A.2 when looking for information on all of the code smells and refactorings in this review.

### 8.2 Poor Method Composition

*“You can't rapidly understand a method's logic”*

**Joshua Kerievsky**

Method composition refers to method structure. An analysis and discussion of issues mentioned in [Kerievsky 2004] and [Fowler 1999] on method composition is given. In Section 8.2.1 a discussion into the breakdown of methods is given. Section 8.2.2 deals with coupling and cohesion. Finally, 8.2.3 discusses IDEA.

Current developers should write maintainable code for future developers. Methods must have appropriate names, which clearly describe their purpose. The result may be more code, but the benefits will be that the code is easier to navigate.

#### 8.2.1 Method breakdown

A developer working on an existing code base needs to go through a lot of code in order to find functionality that is being sought. A project with well-composed methods will allow one to do this quickly, because the developer will focus on methods describing the functionality that he is looking for.

Consider that one wants to find some functionality in a 1000 line method. This illustrates Fowler's “Long Method” code smell. Searching through all the 1000 lines and finding the functionality at the very bottom is not very productive for

the programmer. The entire 1000 line method needed to be understood. Consider that this method is now composed into two methods with descriptive names, each with 500 hundred lines. The developer is saved from reading 500 lines of code provided it is known which method to search. Each 500-line method may then be further broken down. This may continue until the methods are of reasonable size. This concept is similar to finding a number in an unsorted array (1000 line method) as opposed to a sorted binary tree (composed methods).

In section 8.2.2 below, facilities available in IDEA are discussed for detecting poorly composed methods that could be further broken down in the way discussed above.

## 8.2.2 Coupling and Cohesion

Method size should not be the only consideration in dividing long methods. Gamma et al. [1995] emphasise the importance of low coupling and they define coupling to be “The degree to which software components depend on each other”. Methods depending heavily on other classes are highly coupled. High coupling leads to fragile code as one change in a method can propagate changes to all referenced classes.

Although low coupling is desirable, an aspect that Gamma et al. [1995] do not mention is cohesion [Constantine 1979]. Cohesion was first defined by an IBM researcher (Larry Constantine) trying to identify the characteristics of good programming practises through source code analysis. High cohesion often correlates with low coupling, and vice versa. Cohesion is a measure of how strongly related and focused the responsibilities of a single class are.

Cohesion levels are more informative than coupling and give the developer a better idea of how to deal with method composition. Cohesion is categorised into seven levels, ranked according to their level of cohesion. Cohesion can help decide how to divide long and complicated methods into classes or among other classes if appropriate. Good method composition using high cohesion will therefore influence positively on the class.

Removing high coupling levels of a method; by using the ‘Extract Method’ refactoring to divide, the method into smaller methods is futile. This will not change the class coupling level, as the class will still reference the same number of external classes, albeit in more methods now. Instead, the aim should be to reduce the total number of external classes referenced in the class as a whole. Instead, methods from the highly coupled methods should be extracted in such a way as to separate the code containing the largest parts of the “Feature Envy” smell. In other words, code segments with the highest amount of external references should be extracted. This can be achieved by

using the *move method* to move the unwanted functionality into the correct class.

The removal of high coupling within classes creates communicational cohesion. Communicational cohesion ensures that methods are working on data that is co-located in a class. Functional cohesion is preferable to communicational cohesion, but is harder and sometimes impossible to create. Functional cohesion will group methods together in a class to contribute only to the single responsibility of the class.

Functional cohesion will automatically enable a developer to understand a class quicker. This ensures that the class and its methods perform no other tasks, other than the ones required to fulfil the responsibility of the class. It is therefore a very desirable attribute to strive for when designing a new class or refactoring an existing class through method composition. If functional cohesion is not attainable or impractical to achieve in context, then communicational cohesion is sufficient. Classes with cohesions levels lower than that of communication cohesion (as defined in [Constantine 1979]) should be refactored through proper method composition so they adhere to the rules of communication cohesion.

### 8.2.3 More informative smell detections

IDEA has several inspections, which can be used to indicate poor method composition. These inspections reveal method characteristics such as cyclomatic complexity, level of nesting, method coupling, and method length.

Complex methods are shown by measuring cyclomatic complexity. Nested methods indicate the level of nesting that is made with *if*, *while*, *try*, *for* statements etc. Coupled methods show methods referencing too many classes. Long methods can also be an indication of poor method composition.

Each IDEA inspection in relation to these characteristics accepts a threshold parameter to control the inspection results. Highly coupled methods have already discussed them in section 7.9.1 under the heading of “Inappropriate intimacy” and will not be covered here. “Long methods” were also discussed in section 7.8.1.

A method nesting depth is measured by the maximum depth at which statements are found in a method. The statements that contribute to the nesting depth include *if*, *while*, *switch* and *for* statements. Take for example an *if* statement containing a *for* loop and that *for* loop containing another *for* loop. In this case, the maximum nesting depth is three.

Higher nesting depths are often found in longer methods. However, it is generally more profitable to base smell detection on nesting depth, rather than on a search governed by method size alone. This is because long methods may embody trivial code, despite their length.

The ‘Compose Method’ refactoring shown by [Kerievsky 2004] tells us to transform the logic into a small number of intention-revealing steps at the same level of detail. In other words, the small number of intention-revealing steps would comprise of methods extracted from the original method. This improvement is powerful in its simplicity, but has no regard for coupling or cohesion. For improvements taking cohesion and coupling into account, then research into how the method is fulfilling its responsibilities to the class, should be conducted. Further refactorings considering cohesion and coupling issues will be of a higher value as they will contribute more to the overall design of the classes.

One may use all the refactorings mentioned by Fowler to simplify conditional statements, as these will be part of the nesting. Extract method may also be used in this case, but care should be taken with deeply nested structures as they are tricky to refactor.

Large nesting depths in methods can make even a short and lowly coupled method a maintenance hazard. Using IDEA to detect smells that exceed a nesting depth threshold of four yielded the following results for the JDK1.4.2.

Packages affected: 73

Occurrences: 435 (for a nesting depth threshold of 4)

Nesting depth	$\geq 5$	$\geq 6$	$\geq 7$
Occurrences	435	154	53
Complexity	Medium	Medium	High
Value	High	High	High
Classification	Strategic	Strategic	Strategic

**Table 13: Method nesting depth analysis results**

Classification Justifications – One can see that as the nesting depth increases, the higher the complexity and value becomes. Threshold values were chosen to filter out items, which displayed the highest complexity and value.

## 8.3 Creational Issues

In this section, code smells related to the creation of objects are considered. In the first three subsections, the focus is on constructors. In the last subsection, subsection 8.3.4, the theme is on scenarios in which data and code that is used to instantiate a class spans across several classes. That way one is able to detect more smells quicker, instead of just focusing on one particular smell.

### 8.3.1 Non-private Utility Class constructors

This code smell is detected through IDEA and is not found in [Fowler 1999] or [Kerievsky 2004]. Utility classes only contain static methods and have no state. Utility classes should have all public constructors removed. In addition, since utility classes with no constructors can still be constructed with the default constructor provided by Java, the default constructor should be made private. These measures ensure that access to utility classes is always static. IDEA provides smell detection for utility classes with public constructors or no constructors at all.

IDEA has global automatic removal of these two smells built in already. It can detect utility classes with no private or public constructors. It automatically creates private default constructors in all offending utility classes with the click of a button if none existed beforehand.

Packages affected: 50

Occurrences: 101

Refactoring Complexity – Trivial

Value – Medium (Due to low time complexity)

Classification – Quick Wins

Classification Justifications – Due to low complexity and relatively high value, this smell is classified as a quick win.

### 8.3.2 Confusing or too many constructors

Unless only static methods are called on a class, then one can be sure that a constructor needs to be called to create the object before it can be used.

Kerievsky [2004] replaces the constructors with intention-revealing “Creation Methods” that return object instances. These methods may be seen as static or non-static factory methods. The confusing constructors themselves are made private as a result.

Detection: IDEA can be used to search for classes with too many constructors.

Classes with too many constructors will result in confusion. Just how many constructors are too many will vary from class to class. If one finds that one cannot understand the creation process fast enough, then it is a sure sign that the construction process is too complicated and needs refactoring.

One should focus on classes that are referenced often. This arises in more complexity, but greater refactoring value. Classes needed to exceed a constructor threshold of six in order to be seen as potential problems. Use lower thresholds to find more potential refactoring candidates.

Use the 'Introduce Factory' refactoring in IDEA to replace the constructor with a creation (factory) method. The 'Replace Constructors with Creation Methods' refactoring from to pattern technique Kerievsky [2004] is already available in both Eclipse and IDEA.

Packages affected: 17

Occurrences: 31

Refactoring Complexity – Trivial

Value – Medium (Due to low time complexity)

Classification – Quick Wins

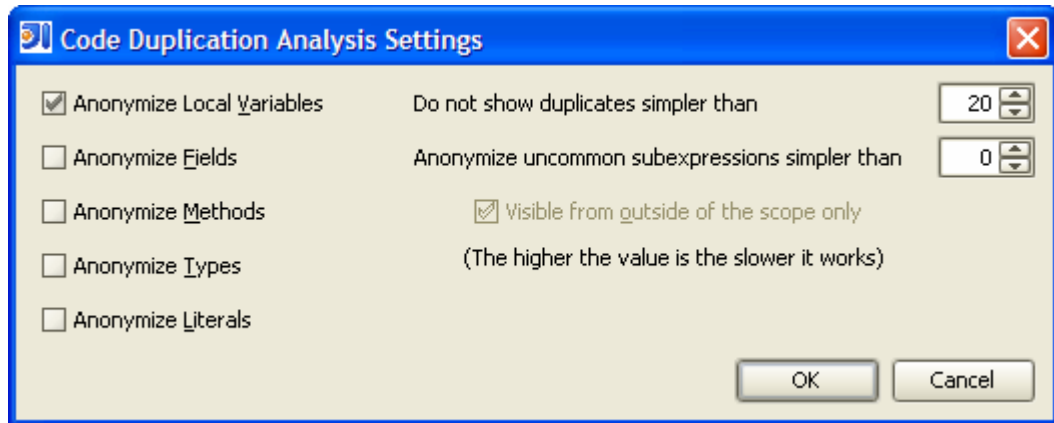
Classification Justifications – Due to low complexity and relatively high value, this smell is classified as a quick win.

### 8.3.3 Constructors with duplicate code

It has already been pointed out that code duplication can be seen as a bad smell. In general, more code duplication is found in long classes. This rule can be extrapolated to constructors. The more constructors a class has, the greater the chance that they will contain code duplication.

As a proof of concept, the java.util package was searched for classes with too many constructors. To guarantee some duplication, a constructor threshold limit of six was used (refer to section on confusing constructors). The GregorianCalendar class had seven constructors. A duplicate search was performed on this class to search for duplication in constructors. The settings used are displayed in figure 31.





**Figure 32: Constructor code duplication settings**

Only local variables are anonymized. Fields do not need to be anonymized as one is only searching for duplicates in one class. Literals are left out in order to return fewer duplicates.

In *java.util.GregorianCalendar*, five duplicate groups were spotted, two of which are groups where constructors contained duplicate code. One group had three duplicates; the other, had two.

The ‘Chain Constructors’ refactoring discussed by Kerievsky [2004] is one possible solution to this smell. Chaining constructors involves calling one constructor from the other. However, it was not possible to chain constructors in the example mentioned above. Instead, the duplicate code was removed by extracting a private constructor from the duplicate code. This private constructor can then be called in place of the duplicated code. This refactoring is called “Extract Constructor”.

The first class with a high number of constructors, turned out to have duplicated code in the constructors. Subsequent random attempts to find duplicates resulted in only a handful of constructors containing duplicate code.

Note that the Java language limits this refactoring, such that each constructor can at most call one other constructor. All other initialisation code must follow the first constructor call. The amount of code duplication is usually low for this code smell when compared to searching for normal code duplication. It is also very difficult to locate this smell. The value added is therefore minimal and there is more complexity to deal with. Searching for this smell should therefore be avoided in favour duplicate detection searches that have a higher value (mentioned in the Locating Duplicates section).

### 8.3.4 Distributed creation information

This smell occurs when data and code used to instantiate a class, starts to span across numerous classes.

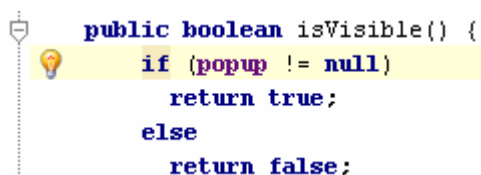
Detection: IDEA has no way to detect high coupling inside constructors.

The factory method is usually used when a class requires many other classes in order to be created and the creational logic can be dispersed to the point that it starts to exist in different classes. Take for example a Maze class, which requires room, door and decoration information. The refactoring will make the offending constructors private, replacing them with a public *create* methods and replacing all uses of the now private constructors with the new factory method. This refactoring employs a design pattern and ensures that the creation logic is placed in one class only.

Use the ‘Move Creation Knowledge to Factory’ refactoring to pattern technique to solve this issue. IDEA has the ‘Introduce Factory’ refactoring for this code smell.

## 8.4 Redundant if Statements

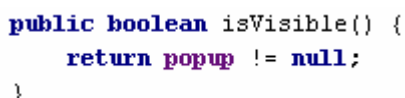
Replacing redundant if statements with better logic is good programming practice. This issue can also be one of coding style. One can argue that, brevity saves lines of source code. IDEA provides a global quick fix to eradicate all of these smells with one click. This is why one can consider the smells as Quick Wins. See Fig 3.3 and the text to the right of it to see how to do this.



```
public boolean isVisible() {
    if (popup != null)
        return true;
    else
        return false;
}
```

**Figure 33: Redundant if - before refactoring**

Clicking on the light bulb (see Figure 33 above) results in IDEA suggesting to us to simplify the *if* statement.



```
public boolean isVisible() {
    return popup != null;
}
```

**Figure 34: Redundant if - after before refactoring**

The result (see Figure 34) of simplifying the *if* statement is cleaner code.

The following results were obtained when running IDEA on the JDK 1.4.2.

Packages affected: 48

Occurrences: 161

Refactoring Complexity – Trivial

Value – Medium (Due to low time complexity)

Classification – Quick Wins

Classification Justifications – Due to low complexity and relatively high value, this smell is classified as a quick win.

## 8.5 Unused Code

One can carry out smell detection for unused code in IDEA as indicated in Figure 34. Navigate to the inspection profile settings and supply a string prefix for the filter. Click the Funnel on the top right and IDEA will return all categories that correspond to the keyword that was requested. The string “unused” is used to search for unused code.

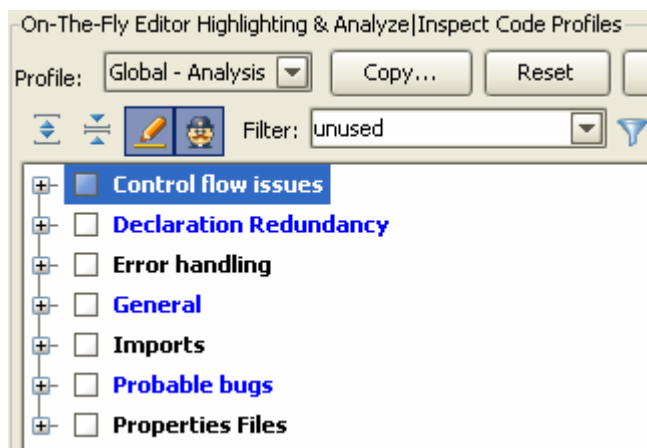


Figure 35: Searching for unused code in IDEA

### 8.5.1 Redundant local variables

A local variable that only gets used once is redundant. It should be replaced by the value that is assigned to it. Fixing this smell has little value.

```
Entry e = new Entry(hash, key, value, tab[index]);
    tab[index] = e;
```

is replaced by: `tab[index] = new Entry(hash, key, value, tab[index]);`

IDEA does this refactoring automatically. Fowler [1999] calls it Inline Temp.

Packages affected: 125

Occurrences: 335

Refactoring Complexity - Trivial

Value – Low

Classification - Low Hanging Fruit.

Classification Justifications – Due to low complexity and low value, this smell is classified as low hanging fruit.

## 8.5.2 Unused method parameters

This smell is discussed in section 6.1.3 under the Eclipse Review.

IDEA provides a global quick fix to eradicate all of these smells with one click. This is why one can consider the smells as Quick Wins. IDEA also checks if these parameters are not used in methods implemented or overridden from the class. This additional check explains why the number of occurrences found in Eclipse (4890) is greater. IDEA therefore only detects less complex versions of this smell and as a result, the solution is less complex.

Packages affected: 94

Occurrences: 1122

Refactoring Complexity - Trivial

Value – Medium (Due to low time complexity)

Classification – Quick Wins

Classification Justifications – Due to low complexity and relatively high value, this smell is classified as a quick win.

## 8.5.3 Redundant throws clause

Read about the “Method with too many exceptions declared” smell detected by IDEA for more background information.

IDEA provides a global quick fix to eradicate all of these smells with one click. This is why one can consider the smells as Quick Wins. IDEA also checks if this method does not use the *throws* clause in methods implemented or overridden from the class. This additional check explains why the number of occurrences found in Eclipse (623) is greater. IDEA therefore only detects less complex versions of this smell and as a result, the solution is less complex.

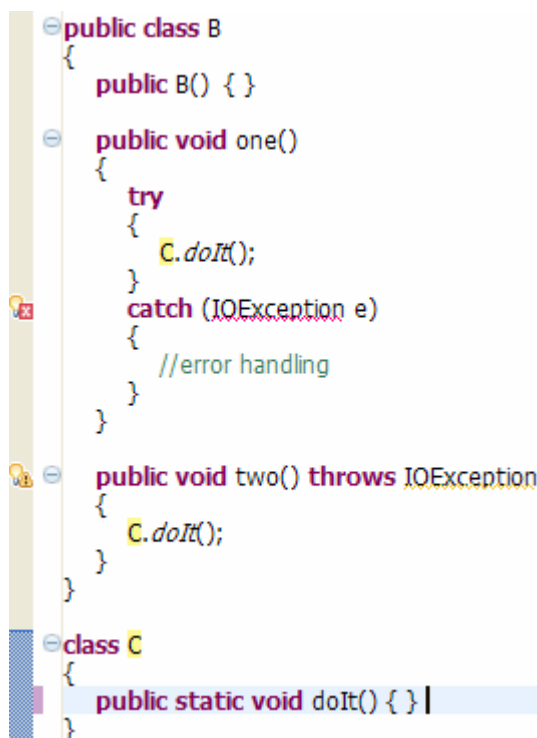
Side Effects: This smell produces another potential smell after refactoring. Once removed, redundant throws clauses must have had a place where they were caught or re-thrown. All catch statements used to catch these clauses are

redundant. All throw clauses used to re-throw the exceptions are also redundant. This means that this refactoring needs to be run as many times as new redundant throws clauses are detected. This could require many iterations of smell detection, but could also reduce the code base considerably.

To illustrate the side effects: *C.doIt()* previously threw a redundant *IOException*. *B.one()* and *B.two()* both called *C.doIt()* and had to handle the *IOException* somehow. This was done by catching or re-throwing the exception.

Now both methods need to be refactored. *B.one()* shows a compile error as the catch statement is now unreachable. *B.two()* shows another redundant throws clause.

This could be a problem depending on how many times *C.doIt()* was called throughout a project and how its exception was handled by the callers and their callers and so on. For an idea on how this smell can become serious, please see the section on “Methods with too many exceptions declared”.



```
public class B
{
    public B() { }

    public void one()
    {
        try
        {
            C.doIt();
        }
        catch (IOException e)
        {
            //error handling
        }
    }

    public void two() throws IOException
    {
        C.doIt();
    }
}

class C
{
    public static void doIt() { }
}
```

Figure 36: Exception side effects.

Note that although Eclipse was used to display the code, the behaviour is similar in IDEA. IDEA will need some initial setup to display all warnings, though.

Warning: This detection also finds exceptions that are thrown in interface methods. Applying a global quick fix in this case can have potentially disastrous effects. One only needs to go as far as the *java.sql.Array* interface to see what would happen if the exceptions were seen as redundant and removed from the interface.

Packages affected: 68

Occurrences: 491

Refactoring Complexity - Trivial

Value – Medium (Due to low time complexity)

Classification – Quick Wins and Strategic refactorings

Classification Justifications – Due to low complexity and relatively high value, this smell is classified as a quick win.

## 8.5.4 Unused imports

Unused imports seem like a trivial code smell. However, over a long time, unused imports can litter a large class. This is especially the case when new classes are added and removed, without cleaning up any imports. Over time the imports can take up many lines of code, especially in heavily used classes.

IDEA can organise imports in every class simultaneously. Checking to see that everything still compiles should provide sufficient testing. This smell is further discussed in section 6.1.1 under the Eclipse Review. The redundant import smell is also fixed when one organises the imports.

Select a package or file and press Ctrl-Alt-O.

Packages affected: 37

Occurrences: 3427

Refactoring Complexity - Trivial

Value – Medium (Due to low time complexity)

Classification – Quick Wins

Classification Justifications – Due to low complexity and relatively high value, this smell is classified as a quick win.

## 8.5.5 Field can be local

This would be an important inspection for reducing the amount of data in a class. Network traffic e.g. could be reduced in web-applications if the classes passed through the network are made smaller, by removing redundant fields. Consider first how many local fields need to be created, before removing the field. It is counter-intuitive to declare a member variable that is never returned from the class and only used in one method.

IDEA will provide a refactoring for this smell, named ‘Convert to local’ after analysis. IDEA provides a global quick fix to eradicate all of these smells with one click. This is why one can consider the smells as Quick Wins.

Side Effects: Say somebody is relying on the size of class to always stay constant and uses it to identify that class (an in-experienced programmer may do this). If a field is removed from the class, the programmer’s code does not recognize the class anymore, since its size has changed!

Packages affected: 48

Occurrences: 179

Refactoring Complexity – Trivial

Value – Medium (Due to low time complexity)

Classification – Quick Wins

Classification Justifications – Due to low complexity and relatively high value, this smell is classified as a quick win.

## **8.6 IDEA Code Review Conclusion**

This conclusion provides a consolidated view in table 14 of this chapter and the previous one. The table groups code smells detected in the 1.4.2 version of the JDK by the various classification groups previously identified: Quick Wins, Strategic, LHF and Avoid. Recall that relatively high threshold values were used for the above in order to produce a conservative amount of code smells. The table includes the number of code smells detected in each case.

Large numbers of trivial code smells that do not have global quick fixes should rather be detected by dynamic inspection in IDEA. This is because it will take too long to refactor the code smells manually. IDEA should be configured to pick up emerging smells while developers start to create or change functionality. As the developer navigates the code base, an attempt should be made to refactor code smells located in the current areas of focus. The developer will understand the code he/she is creating or editing and will therefore be in a better position to refactor. This method of dynamic inspection, which can be switched on in IDEA, is far better than having to do a manual code inspection for a large amount of files.

Name	Occurrences	Value	Classification
<b>Quick Win Group</b>			
Duplicates	68	High	Quick Wins
Confusing Constructors	31	Medium	Quick Wins
Non-private Utility Class	101	Medium	Quick Wins
Unused parameters*	1122	Medium	Quick Wins
Redundant throws clause*	491	Medium	Quick Wins
Unused imports *	3427	Medium	Quick Wins
Field can be local*	179	Medium	Quick Wins
Redundant if statement*	161	Medium	Quick Wins
<b>Strategic Group</b>			
Method with too many exceptions declared	48 to 1	Medium to High	Strategic
Nesting Depth	453 to 35	Medium to High	Strategic
Long Method	1264 to 92	Low to High	LHF to Strategic
Large Class	278 to 44	Low to High	LHF to Strategic
Too many parameters	681 to 35	Low to High	Avoid to Strategic
Inappropriate Intimacy	1277 to 29	Low to High	Avoid to Strategic
Feature Envy	58	Low to High	LHF to Strategic
Instance of Chains	173	Low to High	Strategic
<b>Low Hanging Fruit</b>			
Public field	700	Low	Low Hanging Fruit
Redundant local variables	333	Low	Low Hanging Fruit
<b>Avoid</b>			
Refused Bequest	5485	Low to Medium	Avoid
Magic Numbers	13003	Low to Medium	Avoid

**Table 14: IDEA code smell summary**

\* - represents smells having a global quick fix. This fix can be applied to all offending files with one click. Value was increased in these cases due to the massively reduced time complexity.

IDEA can educate developers about code smells by its dynamic detection of code smells. This will prevent developers from re-creating the code smells. A large team of developers practising refactoring will slowly remove all of the code smells over time.



Another good way to categorise code smells is to use the taxonomy created by [Mantyla 2006]. This taxonomy is presented in the appendix at the end of the dissertation (refer to Appendix A.3).

## **8.7 Summary**

This concludes the full IDEA code review. An analysis of the JDK is given from a code smell and refactoring perspective. This analysis includes work from the author, who has mapped new and existing code smells and refactorings by investigating IDEA.

The next chapter gives a comparison between IDEA and Eclipse from a productivity and feature point of view. The discussion also touches on how well each IDE (Integrated Development Environment) handles refactoring complexity.

## Chapter 9 An IDE Comparison

Table 15 is used to compare the aspects thought to be most important when considering good refactoring tool support in a modern Java IDE. This chapter provides a comparison between IDEA and Eclipse from a productivity and feature point of view. Note that even though only 19 code smell inspections were found in IDEA, it is possible to find a few more code smells, which are suitable targets for refactorings.

It is important to define the comparison criteria before the results are produced:

1. Code smell inspections – Here the physical number of code smell inspections available is shown for each IDE. Note that these are the code smell detections that were identified by the author and that there could possibly still be several more of them available in IDEA.
2. Dynamic Analysis – This refers to the ability of the IDE to automatically detect code smells upon opening of a specific resource (such as a Java source file). The type of code smells detected would depend on user settings. This feature is much like that of Words ability to dynamically pick up spelling mistakes in a document.
3. Refactorings available – This tells us whether or not there are a large number of refactorings available in the IDE. Both IDEs had a comparable number of refactorings available.
4. Smell detection speed – this tells us how quick it is to locate the code smells. Note that large projects were used to measure detection speed. Eclipse uses a compiler-based method to perform analysis. IDEA uses a static analysis detection method, which is far slower than that of Eclipse.
5. Assisted smell removal – Eclipse allows one to fix each code smell that it detects. In most of the cases, the removal of the code smells is rather simple. There are problems with the automated removal as discussed for example in Section 6.1.3. IDEA has very good smell removal techniques, but not all of them are automated.
6. Global smell removal – This criterion refers to being able to remove all of the detected code smells in a project, as opposed to having to fix the code smells separately in each file through a manual process.
7. Filter smells by severity – This idea is based on the threshold concept, present in IDEA when one is doing a static code analysis.

8. Detection of complex smells – Eclipse can only detect code smells pertaining to unused code. IDEA can detect somewhat more complex smells, one of the most complex being the location of duplicate sections of code.

	<b>IDEA</b>	<b>Eclipse</b>
Code smell inspections	19	8
Dynamic Analysis	Partial	Yes
Refactorings available	Yes	Yes
Smell detection speed	Slow	Fast
Assisted smell removal	Partial	Full
Global smell removal	Partial	No
Filter smells by severity	Partial	No
Detection of complex smells	Yes	No

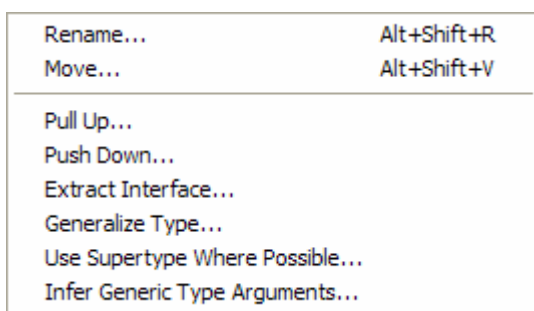
**Table 15: An IDE Comparison**

This chapter provides insight into how Eclipse filters refactoring by context. A discussion is provided on the various productivity issues that arise when one starts refactoring with Eclipse and IDEA. The discussion follows on how refactoring complexity can be dealt with in the other IDE's.

## ***9.1 Filtering Refactorings By Context In Eclipse***

This section is related to productivity issues present in Eclipse. A demonstration will be given on how only certain refactorings appear available to the user, depending on the code element selected.

Examples of available refactorings for different programming attributes:



**Figure 37: Class refactorings**

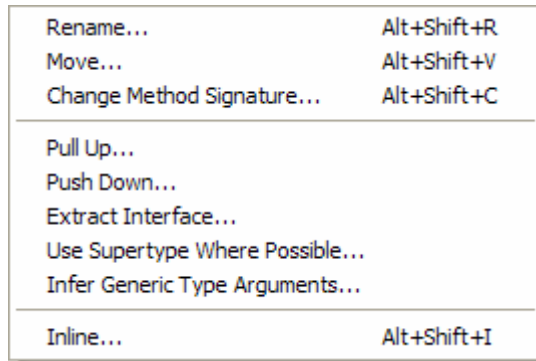


Figure 38: Method refactorings

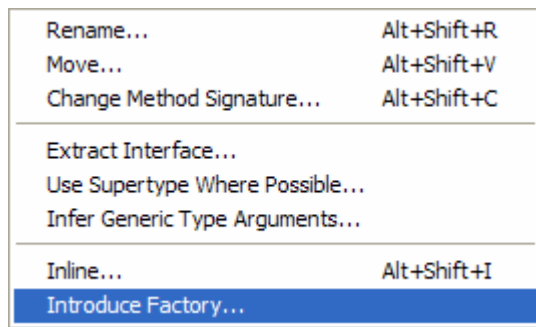


Figure 39: Constructor refactorings

Programmers inexperienced with refactoring will be less prone to pick incorrect refactorings by mistake in Eclipse, due to its selective filtering. This is still not perfect, e.g., Eclipse shows refactorings which rely on classes belonging to class hierarchies even if they don't (such as Pull Up and Push Down).

Eclipse shows refactorings that are applicable in the current context. For example if one right clicks a class, then only class applicable refactorings will be displayed (Figure 37). Note how 'Introduce Factory' is only available for constructors (Figure 39). Although not perfect, it is less confusing to see these filtered menus compared to the 27 or so available refactorings shown by IDEA.

## 9.2 Productivity Issues

There is no doubt that semantic preserving refactorings found in Eclipse and IDEA make the refactoring exercise more productive. The same can be said for finding targets for refactorings.

Eclipse quickly performs checks by building the source. To build the code, first configure the type of warnings emitted from the compiler. Thereafter, incremental builds are performed and code is checked for smells as one saves. Eclipse has a small amount of smells to analyse, so this approach is feasible.

IDEA can do over 400 code inspections, of which 16 mapped to bad code smells. IDEA is pre-configured once to perform analysis transparently; each time a source file is accessed or changed. IDEA can extract a method from multiple code duplicate groups in one operation. This feature only works for low cost duplicate groups in Eclipse.

Eclipse can perform the compilation of the JDK source in the background. Compiling code in the background allows one to carry on coding and using the IDE. In comparison, IDEA consumes large amounts of CPU time and memory while blocking the IDE during analysis. Eclipse performs eight code inspections in the background while compiling the JDK code in under 2 minutes. The java memory settings for IDEA need to be tweaked to configure the system memory (heap size).

IDEA is written in Java. Therefore it uses garbage collection in order to manage its memory allocation and de-allocation. IDEA documentation claims that the garbage collector automatically performs garbage collection before it starts to run out of memory. This is not the case while running a complex analysis and will crash the IDE. The solution is to do this manually by clicking on the bin in the lower left hand corner of the IDE. To ensure optimal performance, garbage collection should be performed before analysis and/or the physical memory used should be increased.

### ***9.3 Handling Refactoring Complexity***

It is easy to make mistakes when fixing unnecessary code in Eclipse. These mistakes occur in complex smells where parameters or exceptions are not used in methods implemented or overridden from a super class. For an example of these issues, please refer back to section 6.1.3.

IDEA simplifies its code smell detection for certain types of unnecessary code, by ignoring class hierarchies altogether. This results in refactorings, which can cause bugs, because of the ignorance of class hierarchy relationships. The advantage is that the refactoring can be automated for many classes. This is quite clear in section 6.1.3.

Code smells in large class hierarchies can identify potential design flaws. This is particularly true of code smells that are more complex and that can lead to strategic refactorings. The over-simplification for the “unread parameter” code smell search in IDEA has the drawback of not being able to identify such issues. This is discussed in section 8.5.2.

## **9.4 Summary**

This chapter provided a comparison between IDEA and Eclipse from a productivity and feature point of view. The previous chapters explore the different features present in Eclipse and IDEA and this chapter provides the results. Discussion centred on how well each IDE (Integrated Development Environment) handled refactoring complexity. It is important to note that no IDE is better than the other. It is up to the user to decide which IDE will best suit their needs based on the advice given in this thesis.

The following chapter will give an analysis of all previous results from a code evolution perspective.

## Chapter 10 Code evolution

*"Complex systems that work, evolved from simple systems that worked."*  
**G.Booch**

Usually if a system is designed well enough then it will have few code smells. Having a design that is more flexible usually leads to code that is more complex. This added flexibility will not always be needed in the future and will eventually clutter up the code base. An example is to use a strategy pattern to allow for multiple algorithms when there is only one algorithm needed at present. The strategy pattern is redundant in this case and breaks the 'YAGNI' rule. YAGNI stands for "One Ain't Gonna Need IT" [Beck 2000] and is used in XP circles to show that one should not build code merely because it might fulfil some future requirements. One should only build code for current requirements and future requirements may change. It is easy to change requirements, but much harder to change code.

A well-built system eventually loses out to growth. Good design cannot foresee all of the possible future requirements and problems. During the evolution of code, there are certain warning signs that can show us that there may be trouble ahead. These warning signs are code smells. If these warnings are ignored, then all sorts of problems tend to emerge, and these eventually can become unmanageable.

The JDK source code (available from the software development kits), has been used below to show how the code has evolved from version 1.02 to version 1.5.0. An investigation was performed on how such a code base grows in number of files, source lines of code and code smells. The classification system from Figure 4 is used to describe the changes and provide graphs, which identify the evolutionary aspects of the code.

### 10.1 A Statistical View

As with the first code review, statistics were gathered on the number of files, number of source lines of code (SLOC, means no commented or blanks lines), number of commented lines of code, number of blank lines and the total number of lines of code in each code base. Thus  $SLOC + Comments + Blanks = \text{Total number of lines of code}$ .

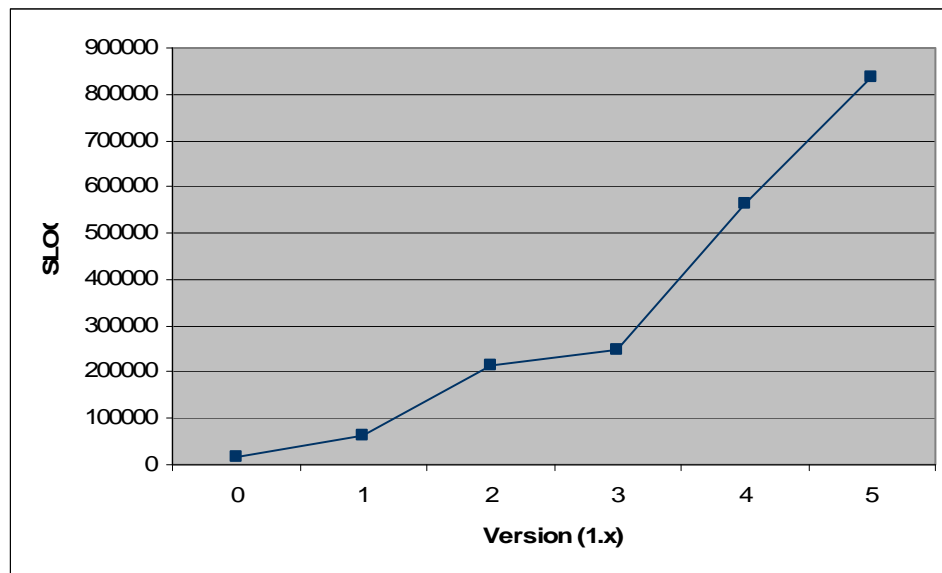
Version	Files	SLOC	Comments	Blanks	Total
1.0.2	217	15,350	17,401	3,333	36,084
1.1.8_10	685	64,795	72,399	11,526	148,720
1.2.2_17	1,654	215,806	227,664	50,938	494,408
1.3.1_17	1,882	246,385	263,834	56,240	566,459
1.4.2_03	4,141	563,073	569,285	161,504	1,293,862
1.5.0_02	6,565	836,453	827,428	223,830	1,887,711

**Table 16: All JDK Code Statistics**

Version	SLOC %	Comments %	Blanks %	Total
1.0.2	43%	48%	9%	36084
1.1.8_10	44%	49%	8%	148720
1.2.2_17	44%	46%	10%	494408
1.3.1_17	43%	47%	10%	566459
1.4.2_03	44%	44%	12%	1293862
1.5.0_02	44%	44%	12%	1887711

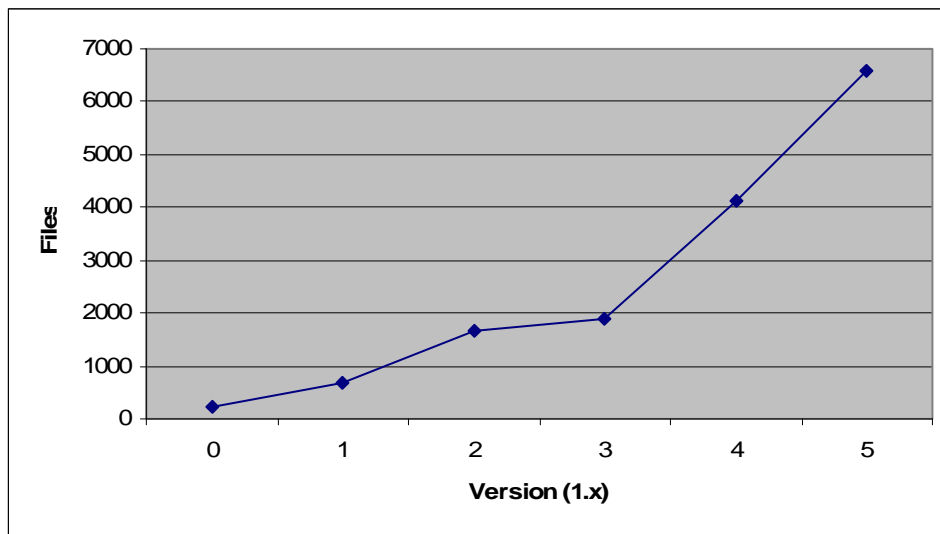
**Table 17: JDK Code Statistics (in percentages)**

The next three graphs are used to identify any possible relationships between the number of physical source lines of code, number of files and code smell across the JDK versions.

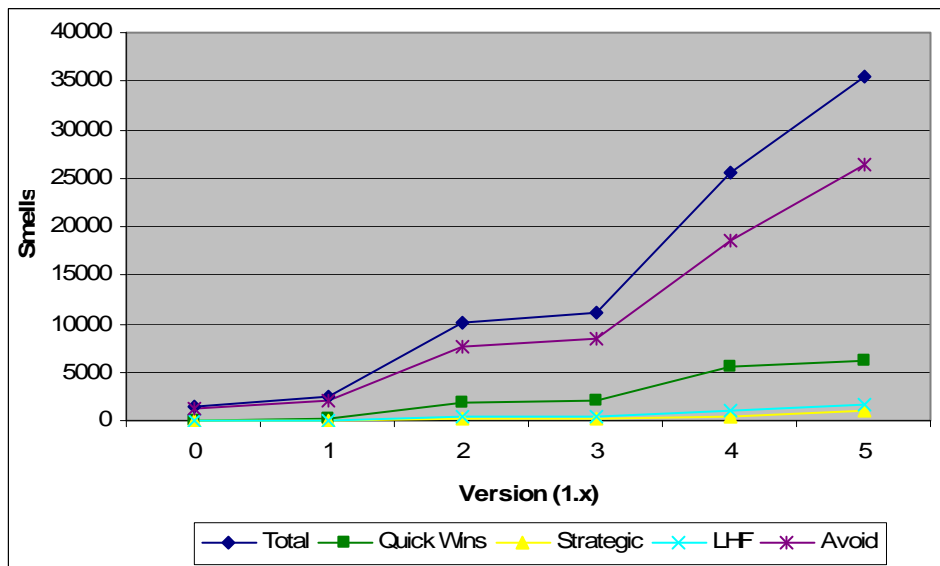


**Figure 40: Source lines of code per JDK version**





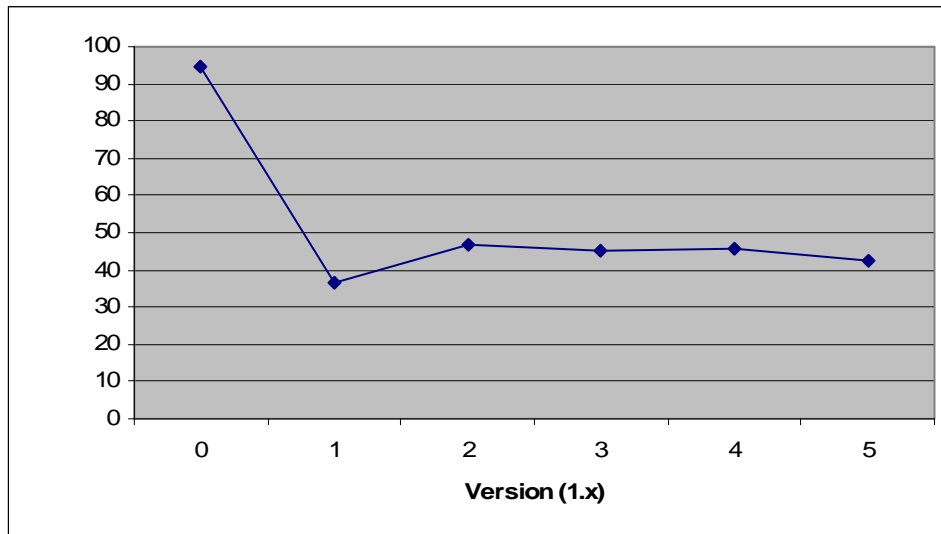
**Figure 41: Files per JDK version**



**Figure 42: Code smells per JDK version (High thresholds)**

From the statistical view of the physical source lines of code (SLOC), number of files and code smell groups, it's obvious that the growth rates for each of these three attributes are very similar. The similar shape of all three graphs (Figure 40, Figure 41, and Figure 42) reveals this trend.

The following figure shows that there is no relationship between the numbers of smells per 1000 lines of physical source code. The data shows that version 1.0 had a high ratio of code smells to source lines of code when compared to the later versions. The ratio in the later versions seems to remain rather constant.



**Figure 43: Smells per 1000 SLOC**

The next few subsections will show the frequency distribution of the various code smells as they appear in each JDK version. Each group will be shown in order of importance. These statistics were used to compile the information in Figure 42.

## 10.2 Quick Win Group

This is the most important group in the code smells. If the global quick fixes (indicated by the \* symbol in Table 18) are proven to be 100% semantically correct then one has a chance to refactor a vast portion of the code without much manual intervention.

	1.02	1.1.8	1.2.2	1.3.1	1.4.2	1.5.0
<b>Duplicates</b>	3	12	29	38	68	86
<b>Confusing Constructors</b>	2	6	19	21	31	40
<b>Non-private Utility Class*</b>	0	7	23	42	101	182
<b>Unused parameters*</b>	40	66	472	538	1122	1448
<b>Redundant throws clause*</b>	13	38	223	225	491	554
<b>Unused imports *</b>	34	108	1045	1131	3427	3509
<b>Field can be local*</b>	9	26	54	63	179	258
<b>Redundant if statement*</b>	0	26	60	69	161	210

**Table 18: JDK - Quick Win Group**

## 10.3 Strategic Group

Section 5.3 showed how thresholds were generated for strategic groups.

Strategic refactoring holds the most value, but at the cost of increased complexity. Table 21 shows the input thresholds (to the right of the names of the code smells) that were used to retrieve the desired code smells. A very conservative set of threshold values was selected, which yielded a low number of code smells. Table 19 shows the number of code smells that were retrieved after using low threshold values. Table 20 shows the number of code smells retrieved with a medium threshold values.

Brackets next to the code smell name surround the actual threshold value. The same threshold values were used across all of the JDK versions to illustrate the growth of the code smells from version to version. Only the highest threshold values were used in order to reduce the total number of code smells reported and to show the code smells with the highest refactoring value. Refactoring does not stop once all of the code smells are refactored. The lower threshold values can be used to report new code smells, until acceptable threshold values are reached.

These strategic refactorings are characterised by their higher complexity levels and lack of automated global quick fixes. They will normally require many low-level refactorings. All of the code smells reported in Table 20 will be of high value due to the high threshold values used during inspection. Note that the inspection tool must exceed the threshold value in order to report the code. The strategic group arises from the ability to detect code smells based on the value that they carry.

Before showing the number of code smells generated for each of the differing code smells. Thresholds for different code smell are explained:

1. Too many method exceptions – Each method searched contains a number of thrown exceptions. If the threshold value is, say, three then all methods with four or more exceptions will be found.
2. Nesting Depth – A nesting depth is determined by the amount of nested statements such as *if*, *switch*, *for* and *while* loops. If the nesting threshold is 4 then all statements with a depth greater than 4 are found.
3. Long Method – If the threshold value is 49, then all methods with more than 49 lines of code will be found.
4. Too many parameters – If the threshold value is four then all methods with more than 4 parameters will be found.
5. Large Class – If the threshold value is nine then all classes with more than nine variables (excluding constants) will be found.
6. Inappropriate Intimacy – This threshold measures the amount of coupling by the number of classes referenced by the class. If the

threshold is nine then any class with more than 9 referenced classes is counted.

	1.02	1.1.8	1.2.2	1.3.1	1.4.2	1.5.0
<b>Too many method exceptions (3)</b>	0	6	16	16	48	167
<b>Nesting Depth (4)</b>	0	35	156	190	435	650
<b>Long Method (49)</b>	29	116	544	623	1264	1909
<b>Too many parameters (4)</b>	24	83	481	526	681	1562
<b>Large Class (9)</b>	11	30	140	159	278	407
<b>Inappropriate Intimacy (9)</b>	0	30	554	615	1277	2100

**Table 19: JDK - Strategic Group (low thresholds)**

As can be seen above, low thresholds produce large amounts of code smells when compared with the tables below, which use higher threshold values.

	1.02	1.1.8	1.2.2	1.3.1	1.4.2	1.5.0
<b>Too many method exceptions (4)</b>	0	0	4	4	9	78
<b>Nesting Depth (5)</b>	0	11	54	66	154	233
<b>Long Method (74)</b>	9	27	69	75	195	321
<b>Too many parameters (7)</b>	5	12	67	73	108	278
<b>Large Class (19)</b>	2	7	42	46	105	159
<b>Inappropriate Intimacy (29)</b>	0	4	41	55	155	271

**Table 20: JDK - Strategic Group (medium thresholds)**

As the threshold values increase, so the amount of generated code smells decreases.

	1.02	1.1.8	1.2.2	1.3.1	1.4.2	1.5.0
<b>Too many method exceptions (5)</b>	0	0	0	0	1	48
<b>Nesting Depth (6)</b>	0	5	23	24	53	80
<b>Long Method (99)</b>	7	15	32	37	92	155
<b>Too many parameters (10)</b>	0	2	14	18	35	74
<b>Large Class (29)</b>	0	1	16	16	44	70
<b>Inappropriate Intimacy (49)</b>	0	2	6	8	29	62

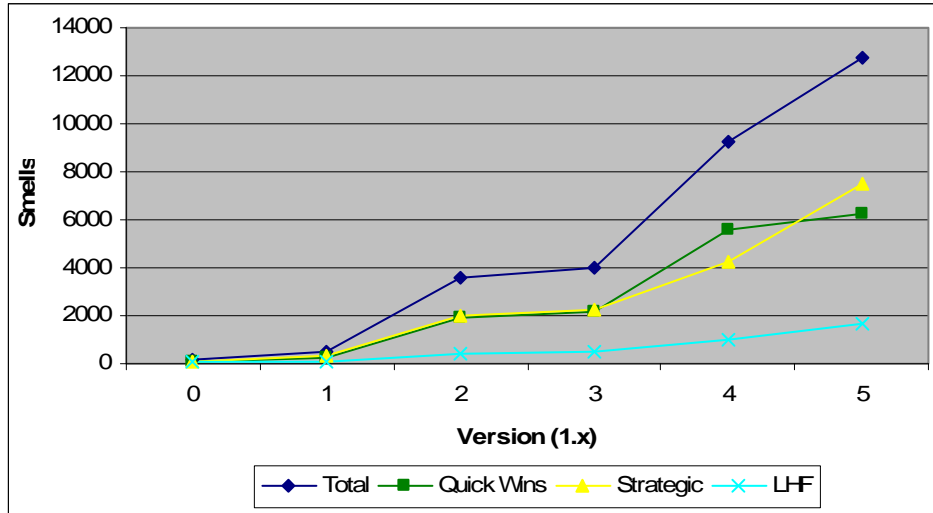
**Table 21: JDK - Strategic Group (high thresholds)**

The values in Table 21 are the original values that were used for the IDEA review. They are the most conservative set of values that were chosen as can be seen by the small amount of generated code smells.

	1.02	1.1.8	1.2.2	1.3.1	1.4.2	1.5.0
<b>Feature Envy</b>	2	8	28	33	58	219
<b>Instance of Chains</b>	0	10	55	65	173	278

**Table 22: JDK - Strategic Group (Threshold independent smells)**

The code smells in Table 22 require no thresholds for generation and therefore only have one set of results. Even though no threshold values are present these code smells are here because they were classified as being strategic targets for refactoring.



**Figure 44: Code smells group per JDK version (low threshold values)**

Compare Figure 42, which displays code smell information with low threshold values with that of Figure 44 where high threshold values apply. There is a marked increase between the two yellow lines representing the count of strategic code smells across all of the JDK versions. This clearly illustrates the large impact that the threshold values can have on the number of code smells produced.

## 10.4 Low Hanging Fruit Group

The LHF group contains a large number of trivial code smells. They are easy to refactor, but their number of occurrences has grown to such a level that it would be impractical to refactor all of the code smells. Note that if these code smells had been tackled in the earlier JDK versions, then it might have been easier to handle them in current versions. Had the developers been aware of the code smell and starting fixing them early on, then they might also have been encouraged to prevent these code smells from re-appearing.

	1.02	1.1.8	1.2.2	1.3.1	1.4.2	1.5.0
Public field	47	63	284	304	700	1135
Redundant local variables	3	13	123	163	333	547

**Table 23: JDK – Low Hanging Fruit Group**

## 10.5 *Avoid Group*

Due to the relatively high number of occurrences, these code smells should be avoided. The reasoning behind this is that the sheer number of the code smells, makes them impractical to remove, since it would take a very long time to remove them. The resulting value of removing these code smells would not warrant the large amount of time expended on removing them. There is no way to sort the code smells into high or low value groups, except by manual inspection. The nature of these code smells is such that some of them could be of little or no value. As a result, much time will be lost in trying to refactor these smells, due to the manual filtering required. As with the “Low Hanging Fruit Group”, these code smells should have been caught earlier on in the development process, where their removal would have been much less of a problem.

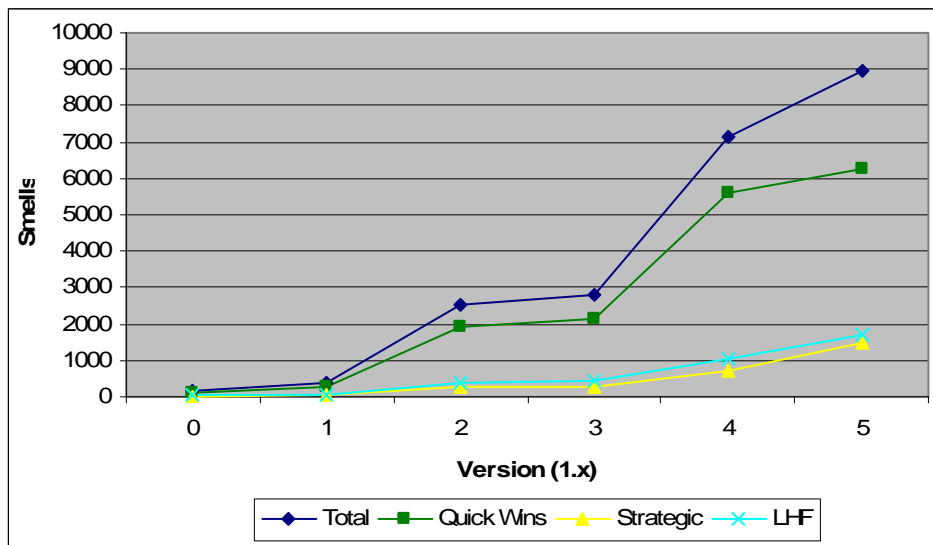
	1.02	1.1.8	1.2.2	1.3.1	1.4.2	1.5.0
<b>Refused Bequest</b>	97	350	2414	2680	5485	7870
<b>Magic Numbers</b>	1196	1616	5198	5699	13003	18622

**Table 24: JDK - Avoid Group**

## 10.6 *Removing the Avoid Group*

The “Avoid Group” was removed in order to get a better statistical view of the other groups. The results (see Figure 45) again showed a graph that resembled those of Figure 40 and Figure 41 (SLOC and number of files in the JDK version). Of course, it cannot be claimed that this similarity phenomenon would be reproduced across all projects. However, it makes sense that if code smells are not refactored, then over time they will increase proportionally in line with the size of the code base. The growth rates between the different JDK version were almost identical.

The results also show that the “Quick Win” group closely follows the total number of code smells across the JDK versions. This is good news as most of the code smells in the “Quick Win Group” are easily refactored.



**Figure 45: Code smells group per JDK version (- avoid group)**

## 10.7 Summary

This chapter gave an analysis of all previous results from a code evolution perspective. All classification groups were represented statistically and it was determined that there was a distinct linear relationship between the size of the JDK code base and the number of code smells as the code base grew between different versions.

The last chapter describes the contributions made to the field of refactoring, analysis results are presented, future work is proposed and the conclusions derived from this work.

## Chapter 11 Conclusion

“Any damn fool can write code that a computer can understand the trick is to write code that humans can understand.”

**Martin Fowler.**

### 11.1 *Refactoring Contributions*

This dissertation shows how to efficiently identify and refactor common and new code smells, by using a popular IDE on a large code base. Common code smells are those presented by Martin Fowler. New code smells defined by the author, gave rise to suggestions for new refactorings that, to the best of his knowledge, have not been previously proposed.

Detailed discussions based on the authors experience were given on all code smells. The most significant new code smell mentioned is that of unnecessary code. The many different variants of unnecessary code were discussed as well as the refactorings used to eliminate these smells. By proposing a way to classify code smells, the research conducted gave more insight into when to refactor code smells and the possible pitfalls one may find along the way. The code smells and refactorings proposed by Fowler [1999] were used as a rough guide and additional comments made by the author of this dissertation also made the purpose behind the refactorings more clear.

A new way to classify code smells was proposed. The classification is used to show that refactoring is not always practical in every context. The aim was to detect code smells that brought the most value to a refactoring exercise. IDEA can successfully be used to attain code smell classifications. A comparison was given between Eclipse and IDEA from a number of perspectives in Chapter 9.

Mens et al. [2004] point out that although commercial refactoring tools have begun to proliferate, research into software restructuring and refactoring continues to be very active, and remains essential to reveal and address the shortcomings of these tools. This dissertation clearly identifies any such shortcomings as well as advantages in the tools selected.

The code evolution of a large project is discussed and classification is used to determine what should and should not be refactored. There are many discussions on each code smell based on the authors experience in general. These discussions include observations concerning shortfalls identified in the two IDEs when considering automated refactoring tools. A number of comments are made on how the design of a program can be improved by each refactoring described.



## 11.2 *Analysis Results*

Refactoring is not always beneficial. Sometimes code smells are put there for very good reasons, e.g. to increase performance. It pays to be familiar in the problem domain in which one is working before starting to refactor. Knowing the difference between good and bad practices in a domain can mean the difference between good and bad refactorings. No one should be allowed to attempt complex refactorings without decent exposure to the system or problem domain. Code reviews are another line of defence to ensure that refactorings are improving the design of existing code. Knowledge of good design is essential, before one starts to refactor.

For refactoring to succeed, one needs to refactor code smells pointing to design flaws. In general, strategic code smells pointed to bad design. Not every code smell found has the same refactoring value e.g. finding duplicates will be more valuable to refactor than unused imports. Not every instance of one specific code smell type will necessarily have the same refactoring value e.g. when finding duplicates, not all of the duplicates found will be equally valuable.

Being able to classify code smell types, can lead to greater refactoring productivity. This is often only possible with the right refactoring tools, such as IDEA which can (in some cases) offer analysis threshold parameters in order to filter out code smells by the refactoring value and complexity that they hold.

The resulting analysis performed with IDEA and Eclipse, gave a list of code smells with the number of times each code smell occurred in each version of the JDK. This list may be used to see the quality of the JDK code, through a code smell perspective. When looking at the design of the JDK at very detailed granularity, one can see how the design is affected through the many code smells present. The more code smells present the harder the JDK will be to maintain.

Importance is placed on the need for continuous design, which constantly integrates refactoring into the development process to ensure good code maintainability. It is shown that IDEA is one of the best tools to use when one needs to employ a continuous design technique. It is also shown that tools such as Eclipse have drawbacks, which can cause refactoring to break down if proper care is not taken.

Different refactorings require different amounts of effort, and provide different returns in terms of the improvement of the system as a whole. This means that it is necessary to characterize code smells and their refactorings to prioritize their application. This is achieved through employing a four-quadrant model, which contrasts complexity against value.

In the study of the evolution of the JDK, the analysis results conclude that code smells generally increased at a linear or constant rate throughout all of the different versions. The only exception to this rule was that of the average number of code smells per 1000 lines of code, decreased sharply from the first to second version of the JDK.

In principle, poor code smell detection in Eclipse can be remedied with plug-ins such as PMD (<http://pmd.sourceforge.net>). However, this plug-in does not integrate very well into Eclipse and produces output in HTML format. This raises productivity issues when one compares the abilities of IDEA, which has fully integrated inspections, which provide links to the offending code as well as providing solutions. PMD offers around 160 inspections. IDEA offers over 400. Although PMD can be used as a plug-in in many IDEs, it falls short of the abilities of IDEA.

## **11.3      *Future Work***

A lot of information was gained from the research done; it would be valuable to go further than having statistical measurements of a software system. The code smell classification system provided was very useful and it was clear from the research that as code evolves then the number of code smells increases. To be able to pinpoint the density of code smells in relation to classes would prove very useful.

When classifying code smells, another measure of value would be to measure how often that piece of code is changed or maintained. Refactoring code that is used and changed more often will produce more value than refactoring code that no one uses. This is because refactored code is easier to maintain and it makes more sense to maintain code that is used more often.

The more a class is maintained and used to add new features into, the larger it will grow and therefore the greater the likelihood becomes of more errors and code smells occurring in that class. In effect, finding such hotspots in a software system would alert developers as to where the most refactoring work is needed or in other words, where the greatest density of code smells is likely to exist. It would be interesting to be able to persist the code smell information into a database and to be able to mine that data more meaningfully through statistical queries and even to derive patterns in that data.

To be able to rank each class according to the number and type of code smells it contains would tell us straight away where to start refactoring. If versioning information were also incorporated then this would further enrich the ranking of the classes. One might be tempted to say that in order to find hot spots, one should only need to look for the largest classes in the system as these would normally point us to the God classes, which are seemingly getting out of control and are starting to suffer from code rot or excessive code smells. These

classes could be the ones that are constantly changed and have enhancements added unto them, causing them to become God classes over time.

More work could be done in order to derive such knowledge from a code base. In essence, one would be mining source code for smells as if it was a database. The obvious problem is where to find the source for all of this code smell data.

During the code review with IDEA, an HTML report was generated for code smells that were detected in the system. The HTML report informs us of each class that contained the code smell, how many times it occurred in that class, in which method and on, which line it, occurred. This is clearly enough information from which one could populate a database

Depending on the project chosen for analysis, most of the work would go into parsing the HTML, retrieving the necessary information and storing it into a database. The information retrieved would be more useful than what any refactoring tool can provide now.

There has been work into building a fact extraction tool that extracts facts from a code base and uses a visualization method in order to organise the data in a meaningful way. Visualizations tools [Lommerse, Nossin, Voinea, Telea 2005] can be used to visualise large code bases, which normally take a very long time to understand. Many different kinds of facts may be extracted. The fact extraction tool can be customised so that the user can focus on facts that are most important to him/her. Such feature extraction tools could also benefit from having code smells identified as the features to be extracted.

The main views identified for visualisation tools from [Lommerse, Nossin, Voinea, Telea 2005] are: The syntactic view, showing the syntactic constructs in the source code. The symbol view, showing the objects a file makes available after compilation, such as function signatures, variables, and namespaces. The evolution view looks at different versions of the same source file during a project lifetime.

There has been some work into the analysis of code evolution by Voinea, Telea and van Wijk [2004]. CVSgrab [Voinea, Telea and van Wijk 2004] is a tool aimed at developers involved in maintenance projects. It acquires the information about artefact evolution of entire projects and it visualizes it down to file level. It enables correlations based on activity and contributors. It may be used as a CVS data acquisition tool for the CVSScan application. CVSScan is an integrated multiview environment, using a line-oriented display of the changing code, where a column represents each version, and where the horizontal direction is used for time. Separate linked displays show various metrics, as well as the source code itself. A large variety of options is provided to visualize a number of different aspects. These two applications are for the C/C++ market.

Tools such as CVSSgrab can possibly be incorporated into an application, to see which class files are being maintained the most and are therefore potential breeding grounds for bad code smells. This will further help prioritize the code smells found in a project.

## **11.4      *Final Thoughts***

Traditional agile practises such as test-driven development recommend the use of refactoring. Unit testing is used to ensure behaviour preservation. Refactoring in this type of scenario is not always perfect. Time constraints and many other issues can result in code being left with code smells. Not all programmers are equally experienced and the less experienced ones may produce many code smells. In some scenarios, refactoring is seen as a luxury and not a necessity, which can lead to various levels of un-maintainable code in the long term.

Refactoring code that has not been refactored over many years is a great maintenance challenge. Continuously refactoring early on in a projects life cycle will help to ease the difficulty of such a challenge. If refactoring is done early, small problems can be prevented from turning into serious problems. Flaws in design can be caught quickly and stopped from spreading. Code smells are therefore a good indication of the state of a project and should be taken seriously. The best refactoring tool is still the experienced developer with his suite of unit tests, practising continuous design.

When considering design heuristics, design patterns, software maintenance, software evolution and software reengineering, this dissertation links each of these software disciplines with the practice of refactoring.

## References

1. [Bandi, Vaishnavi, Turk 2003] BANDI, R.K, VAISHNAVI, V.K AND TURK, D.E: *Predicting maintenance performance using object orientated design complexity metrics*, IEEE Transaction on Software Engineering, vol. 29, no 1, pp 77-87.
2. [Beck 2000], BECK, K, FOWLER, M: *Planning Extreme Programming*, Addison Wesley.
3. [Cinnéide 2000] CINNÉIDE, MEL Ó: *Automated application of design patterns - A refactoring approach*, University of Dublin, PhD dissertation.
4. [Foote 1997] FOOTE, B, OPDYKE, W: *Life Cycle and Refactoring Patterns that Support Evolution and Reuse*.
5. [Fowler 1999] FOWLER, MARTIN. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999.
6. [Fowler 2005] FOWLER, MARTIN: *A list of refactoring tools for several languages*, <http://www.refactoring.com/tools.html>, accessed 2005-08-13.
7. [Gamma et al. 1995] GAMMA, E, HELM, R, JOHNSON, R and VLISSIDES, J: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley.
8. [Garlan 1994] GARLAND, D, SHAW, M: *An Introduction to Software Architecture*, Carnegie Mellon University.
9. [Griswold 1991] GRISWOLD, WILLIAM, E: *Program restructuring as an aid to program maintenance*, University of Washington, PhD dissertation.
10. [Grotehen & Dittrich 1997] GROTEHEN, T, DITTRICH, R: *The MeTHOOD Approach, Transformation Rules, and Heuristics for Object Oriented Design*, Technical report.
11. [IEEE 1998], THE INSTITUTE FOR ELECTRICAL AND ELECTRONIC ENGINEERS, New York: *IEEE Standard for Software Maintenance*.

12. [JUnit 2005] JUNIT HOME PAGE, <http://www.junit.org/>, accessed 2005-08-28.
13. [Kataoka, Ernst, Griswold, Notkin 2001] KATAOKA, YOSHIO, ERNST, MICHAEL GRISWOLD, WILLIAM, NOTKIN, DAVID: *Automated Support for Program Refactoring using Invariants, ICSM 2001*.
14. [Kerievsky 2004] KERIEVSKY, JOSHUA: *Refactoring to Patterns*, Addison Wesley, <http://www.industriallogic.com/xp/refactoring/index.html>, accessed 2005-08-15.
15. [Korman 1998] KORMAN, WALTER, F: *Elbereth: Tool support for refactoring Java programs*, University of California, MSc dissertation.
16. [Lehman & Belady 1985] LEHMAN, M.M, BELDAY, L.A: *Program Evolution*, Academic Press.
17. [Li & Henry 1993a] LI, W and HENRY, S.M: Maintenance metrics for the object oriented paradigm, in *Proceedings of the First International Software Metrics Symposium, IEEE*, pp. 52-60.
18. [Li & Henry 1993b] LI, W and HENRY, S.M: *Object-Oriented metrics that predict maintainability*, *Journal of System and Software*, vol 23, no2, pp 111-122.
19. [Mantyla 2003], MANTYLA, MIKA: *Bad smells in Code: A taxonomy and an empirical study*, MSc Dissertation.
20. [Mantyla 2006] MANTYLA, MIKA: *A Taxonomy for "Bad Code Smells"*, <http://www.soberit.hut.fi/mmantyla/BadCodeSmellsTaxonomy.htm>, accessed 2006-01-02.
21. [McCabe 1976] MCCABE, T.J: *A complexity measure*, *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308-320.
22. [Mens et al. 2004], MENS ET AL., T and TOURWÉ, T: *A Survey of Software Refactoring*, *IEEE Transactions on software engineering*, VOL. XX, NO. Y, MONTH 2004.
23. [Molla 2005] MUHAMMAD K. BASHAR MOLLA: *An overview of object orientated design heuristics*, Department of Computer Science, Umeå University, Sweden, MSc Dissertation.

24. [Müller. Lembeck, Kuchen 2004] R. MÜLLER, C. LEMBECK, H. KUCHEN: *A Symbolic Java Virtual Machine for Test-Case Generation*, Proceedings IASTED, 2004.
25. [Opdyke 1992] OPDYKE, WILLIAM, F: *Refactoring: Object-Oriented Frameworks*, University of Illinois, PhD dissertation.
26. [Refactorit 2006] REFACTORIT: *Refactoring Comparison*, <http://www.refactorit.com/failid/refactorMatrix.pdf>, accessed 2005-03-02.
27. [Riel 1996] RIEL, ARTHUR J: *Object-Oriented Design Heuristics*. Addison-Wesley.
28. [Roberts 1999a] ROBERTS, DONALD, BRADLEY: *Practical analysis for refactoring*, University of Illinois, PhD dissertation.
29. [Roberts 1999b] FOWLER, MARTIN: *Refactoring: Improving the Design of Existing Programs*. (Chapter 14 Contributed by Don Roberts and John Brant) Addison-Wesley, 1999.
30. [Shore 2004], JIM, SHORE: *Continuous Design*, <http://martinfowler.com/ieeeSoftware/continuousDesign.pdf>, accessed 2005-08-11.
31. [Sommerville 2004] SOMMERVILLE, IAN: *Software evolution and reengineering*, Software Engineering 7<sup>th</sup> Edition (Chp 21) Addison-Wesley, retrieved from <http://www.ifi.unizh.ch/req/ftp/kvse/kap06-evolution.pdf>, accessed 2006-09-09.
32. [Sun Microsystems 1999] SUN MICROSYSTEMS, *Code Conventions for the Java Programming Language*.
33. [Tiger 2004] JOSHUA BLOCH AND NEAL GAFTER; *Forthcoming Java Programming Language Features*. Sun Microsystems, 2004. <http://java.sun.com/j2se/1.5/pdf/Tiger-lang.pdf>, accessed 2005-12-28.
34. [Tokuda 1999] TOKUDA, LANCE, A: *Evolving Object-Oriented Designs with Refactorings*, University of Texas at Austin, PhD dissertation.
35. [Van Kempen 2005] VAN KEMPEN, MARC: *Studies into Refactoring of Software Architectures*, Technische Universiteit Eindhoven, MSc Dissertation.

36. [Voinea, Telea, van Wijk 2004] Lucian Voinea, Alex Telea, Jarke J. van Wijk: *CVSscan: Visualization of Code Evolution*, Technische Universiteit Eindhoven.
37. [Lommerse, Nossin, Voinea, Telea 2005] : Gerard Lommerse, Freek Nossin, Lucian Voinea, Alexandru Telea: *The Visual Code Navigator: An Interactive Toolset for Source Code Investigation*, Technische Universiteit Eindhoven.



# Appendix

## A.1 Refactorings

Here refactorings will be categorized and explained. There are many other refactorings than the ones mentioned here. There is specific focus only on those refactorings that have been mentioned in the dissertation. The intention is to be able to look up the refactorings when searching for their meanings in the dissertation.

The Eclipse and IDEA code review will heavily reference this section in order to describe the many solutions to code smells. In some cases, it will be possible to have more than one refactoring as a solution to a code smell.

### Moving Features

#### **Move Method**

This refactoring moves the method to another class. The reasoning would normally be that the method gets used more in the target class.

#### **Move Field**

This refactoring moves the field to another class. The reasoning would normally be that the field gets used more in the target class.

#### **Extract Class**

Split a class into two or more classes when it does the job of two or more classes.

The new class is extracted from another class, by moving the relevant methods and fields from the old class into the new one.

#### **Hide Delegate**

Create delegate methods to hide the delegate for encapsulation reasons. An example would be having class X calling class Y and then Z. Class Y information would be needed to access class Z information. Class Z can be hidden, by creating a method in class Y that calls Class Z and thereby negating the direct call to Class Z.

### Composing Methods

#### **Extract Method**

This refactoring will extract a method from a section of code. This will normally be used to give a name to a section of code so that a large body of code can be abstracted to more methods and therefore be more clearly understood.

### **Replace Method with method object**

One has a long method that uses local variables in such a way that one cannot apply the 'Extract Method' refactoring. This refactoring makes an object out of method and it helps to use the 'Extract Method' refactoring.

### **Replace Temp with Query**

This refactoring is used when one is using a temporary variable to hold the result of an expression. Extract the expression into a method. Replace all references to the temp with the expression. The new method can then be used in other methods.

## **Organizing Data**

### **Replace Bidirectional with Unidirectional**

This refactoring needs to be used whenever one has a two-way relationship between two classes that is no longer needed. The reference to the other class will need to be removed.

### **Magic Number with Symbolic Constant**

Create a constant with a meaningful name in order to avoid mentioning the same constant value in different places.

### **Encapsulate Field**

Make the data item private, add accessors (get/set) in order to enforce encapsulation.

### **Replace Type Code with Class**

Replace type number with class. This will entail having a class that uses a constructor to encode the different types, which can be encoded. The class will normally have a private constructor and will have public variables, which are instances of the same class. These public class instances can be used to store the type information. This will ensure that there is a common place from which one can access and store type code information.

### **Replace Type Code with Subclasses**

This approach is similar to the one above and introduces subclasses instead of type code. The idea is to return a common super class, which could have any child class associated with it. Integers are still used to store type information in.

### **Replace Type Code with State/Strategy**

This approach is very similar to the two above methods, but will normally be used when one cannot use sub classing. The approach is best utilized whenever the type code needs to change throughout the object's lifetime.

## **Conditionals**

### **Decompose Conditionals**

Whenever a complicated invariant exists, it will prove useful to extract parts of the conditional into methods, by using the ‘Extract Method’ refactoring.

### **Replace Conditional with Polymorphism**

This is used whenever we have different behaviors being implemented, depending on the conditions of if statements. The refactoring removes the conditionals and replaces it with polymorphism. This enables the code to be simplified as only one method call needs to be called and the conditionals are removed.

### **Replace Nested Conditional with Guard Clause**

This refactoring removes a particular complicated nesting of “if” and “else if” statements, by introducing guard clauses. The guard clauses negate the need for the “else if” statements and this results in no nesting of if statements.

## **Method Calls**

### **Parameterize Method**

This refactoring can be used in order to consolidate multiple methods that do the same thing and only differ by certain values.

### **Preserve Whole Object**

If one is passing several values from an object into a method, then it is better to rather pass the entire object into the method. This refactoring simplifies the method, by removing unnecessary parameters. This makes the method easier to understand.

### **Parameter Object**

This is when one has a number of parameters that appear inside a method and one chooses to reduce the parameters by grouping them into an object. This will simplify the method, by reducing the number of parameters. This is especially useful if the method is called in multiple places.

### **Replace Constructor with Factory Method**

This allows more complex creation of classes through the use of a factory create method.

### **Replace parameter with method**

An object can invoke a method x() and use the value from this method as a parameter to another method y(). What should in fact happen is that the method to which the parameter is passed (method y) should invoke the original method x().

## **Generalization**

### **Pull Up Field**

In order avoid duplication of a field in subclasses; it would be useful to move the field to the superclass.

### **Pull Up Method**

In order avoid duplication of a method in subclasses; it would be useful to move the field to the superclass.

### **Push Down Method**

If a method is not used in all subclasses, or is only used in one sub-class, it would be useful to move the method to that sub-class. In this way the developer is not lead to think that the method is actually needed in all of the subclasses.

### **Push Down Field**

If a field is not used in all subclasses, or is only used in one sub-class, it would be useful to move the field to that sub-class. In this way the developer is not lead to think that the field is actually needed in all of the subclasses.

### **Extract Subclass**

If there were a subset of features that is not used by a number of subclasses then it would be useful for the subset of features to be extracted into a separate subclass.

### **Form Template Method**

If one has a process that performs similar steps in a specific order, but the steps are different then one can apply the form template method refactoring. One has two methods in subclasses that perform similar but different steps in the same order - get the steps into methods, then pull original methods. Inheritance helps eliminate duplicate behaviour

### **Replace Inheritance with Delegation**

It is possible that a subclass does not match the relationship that it should have with the parent. The ideal relationship would need to use all of the data from the superclass and all of the behaviour. If a subclass does not need the data from the super class, then it might not necessarily need to inherit from it. This is also true of the interface. If the subclass does not necessarily need all of the methods from the super class then it would be a better idea to replace the inheritance and delegate the needed superclass functionality to the superclass.

## **Design Patterns**

### **Compose Method**

This refactoring will be used when one cannot rapidly understand a method's logic. The idea is to refactor the method (using the 'Extract Method' refactoring) into smaller, intention-revealing steps. This will make the method easier to understand and maintain.

### **Replace Conditional Dispatcher with Command**

This will employ the "Command pattern" in order to remove complicated "*if else*" statements. This will also decouple the behaviour from the decision logic.

### **Introduce Factory**

This refactoring will call a Factory create method instead of the public constructor. This employs the "Factory method" design pattern. The idea is to have a single place where the decision and creational logic pertaining to the construction of multiple objects can be housed. These factories will be used for objects, which have an inheritance relationship.

### **Chain Constructors**

It is possible to have a lot of duplicated constructor code inside of a class. Instead of repeating the constructor logic, one can make calls from a constructor to another constructor. This will eliminate code duplication.

## A.2 Code smells

### Unnecessary code

The following code smells are found in Eclipse and IDEA. The refactorings needed in all cases are that of pure deletion of the code. Deletion is reasonable if one has previous versions of the code base stored in a code repository. The only exception is that of the redundant local variables. This code smell uses the 'Inline Temp' refactoring to remove the unnecessary temporary variable. Therefore, if any code is accidentally deleted, then there will be a place from which one could restore. The following is a list of unnecessary code:

#### **Redundant local variables**

This smell happens whenever a temporary variable is used in one place. The temporary variable should rather be replaced directly with the expression, by using the 'Inline Temp' refactoring.

#### **Redundant throws clause**

Some methods may throw exceptions for no reason. This will result in calling methods having to catch the exceptions coming from these methods. This will result in a lot of unnecessary code. The redundant throws clause and the accompanying catch clauses should be deleted.

#### **Unused imports**

Unused imports usually start to build in large classes where new classes are added and removed on a continuous basis. Developers will often forget to remove the unused imports once they delete some code.

#### **Unread local variables**

Some variables may be used during development or testing, but then forgotten.

#### **Unread parameter**

Some variables may be used during development or testing, but then forgotten.

#### **Unread private member field**

Unread fields that are private have no use whatsoever and can be deleted.

#### **Unused private method**

Unread methods that are private have no use whatsoever and can be deleted.

#### **Unused private constructor**

Unread constructors that are private have no use whatsoever and can be deleted.

#### **Unused private type/class**

Unread classes that are private have no use whatsoever and can be deleted.

### **Code Duplicates**

This is one of the worst code smells around and the following refactorings are needed in order to resolve the many variations of the smell:

1. Extract method – This refactoring will extract a method from a section of code. This will normally used to give a name to a section of code so that a large body of code can be abstracted to more methods and therefore be more clearly understood.
2. Parameterize method – This refactoring can be used in order to consolidate multiple methods that do the same thing and only differ by certain values.
3. Extract Class – This is when a new class is extracted from another class, by moving the relevant methods and fields from the old class into the new one.
4. Extract Superclass – Is when one has two classes that share similar features. The similar features are consolidated into one class i.e. the superclass.
5. Form Template method – This is when one has a process that performs similar steps in a specific order, but the steps are different.
6. Introduce parameter object. – This is when one has a number of parameters that appear inside a method and one chooses to reduce the parameters by putting them into an object. This will simplify the method, by reducing the number of parameters.

### **Inheritance Issues**

#### **Refused Bequest**

Some subclasses may not need all of the behaviour and data from their parent classes. This is normally a faint smell and is solved by creating a new subclass, which will have all of the unwanted methods moved into it by using the ‘Push Down Field’ and ‘Push Down Method’ refactorings.

### **Type Code**

#### **Chains using Instanceof**

IDEA can check for chains of *instanceof*, *if* statements. It searches for chain lengths with a minimum size of two, i.e. an “*if*” statement with one or more *else* branches. The check ensures that all the conditionals contain the *instanceof* keyword. A chain length parameter cannot be set, but it still helps us to identify cases, which could benefit more from polymorphism.

## **Abstraction Issues**

### **Feature Envy**

If a method is too interested in another class, extract the offending code using the ‘Extract Method’ refactoring and use the ‘Move Method’ refactoring to move the extracted method to the class that is being called.

## **Encapsulation**

### **Public Field**

If a field is not a constant and part of the data of a class then it should be made private. The field should also have get and set methods provided for it.

## **Method Metrics**

### **Long Method**

A method that is too long cannot easily be understood and can sometimes be a maintenance nightmare. Sections of the code should be grouped into more intention-revealing methods so that the code will be easier to understand. The ‘Extract Method’ refactoring can be used to do this.

### **Long Parameter List**

A method that uses too many parameters is hard to use and to understand. The refactoring ‘Introduce Parameter Object’ can be used to group related parameter fields into an object. This will reduce the number of method parameters.

### **Too Many Exceptions**

A method that throws too many exceptions can be difficult to maintain. Methods that have to catch exceptions from other methods can have many catch statements if a method throws too many exceptions. This makes the code less maintainable. One of the solutions is to create generic exceptions, which have subclasses inheriting from them. This can reduce the number of total thrown exceptions.

## **Class Metrics**

### **Inappropriate Intimacy**

This smell happens when one class gets too involved with another class. This results in high coupling.



## **Large Class**

This smell indicates too many fields inside a class. The refactoring would be 'Extract Class'. If the class is doing too many things it would make sense to extract classes from it in order to make the behaviour more maintainable.

## **Creational Issues**

### **Non-private Utility Class constructors**

Utility classes are meant to have no data and should only expose public static methods, which could be used by any other class. An utility class should have a private default constructor and no public constructors so that it can only be used statically.

### **Confusing or too many constructors**

If one finds that one cannot understand the creational process of a class fast enough, then it would be beneficial to refactor it. The solution would be to replace the constructors with creational methods that describe the creational process more clearly. The 'Replace Constructors with Creation Methods' refactoring can be used.

### **Constructors with duplicate code**

It is possible to chain constructors if they contain duplicate code. This is done by having one constructor calling another and resuming with its own creational code. The 'Extract Method' or in this case 'Extract Constructor' refactoring may be used.

### **Distributed creation information**

This problem occurs whenever code used to instantiate a class starts to span across multiple classes. The solution is to use the 'Move Creation Knowledge to Factory' refactoring.

## **Redundant if Statements**

This code smell occurs when an "*if statement*" is made redundant. For example, assigning a value to a boolean variable. If the condition is true the boolean is true, otherwise it is false. It is easier to assign the condition directly to the boolean variable, which makes the *if else statement* redundant.

## **Magic Numbers**

This smell occurs whenever a constant value is reused multiple times over a project and it has no link to a physical constant. The refactoring is to replace the value with a constant. This allows the value to be changed in one place, as well to assign a meaningful name to it.

## **A.3 Riel Heuristics**

Heuristic #2.1: All data should be hidden within its class.

Heuristic #2.2: Users of a class must be dependent on its public interface, but a class should not be dependent on its users.

Heuristic #2.3: Minimize the number of messages in the protocol of a class.

Heuristic #2.4: Implement a minimal public interface, which all classes understand (e.g. operations such as copy (deep versus shallow), equality testing, pretty printing, parsing from a ASCII description, etc.).

Heuristic #2.5: Do not put implementation details such as common-code private functions into the public interface of a class.

Heuristic #2.6: Do not clutter the public interface of a class with things that users of that class are not able to use or are not interested in using.

Heuristic #2.7: Classes should only exhibit nil or export coupling with other classes, i.e. a class should only use operations in the public interface of another class or have nothing to do with that class.

Heuristic #2.8: A class should capture one and only one key abstraction.

Heuristic #2.9: Keep related data and behavior in one place.

Heuristic #2.10: Spin off non-related information into another class (i.e. non-communicating behavior).

Heuristic #2.11: Be sure the abstractions that one models are classes and not simply the roles objects play.

Heuristic #3.1: Distribute system intelligence horizontally as uniformly as possible, i.e. the top level classes in a design should share the work uniformly.

Heuristic #3.2: Do not create god classes/objects in ones system. Be very suspicious of an abstraction whose name contains Driver, Manager, System, or Subsystem.

Heuristic #3.3: Beware of classes that have many accessor methods defined in their public interface, many of them imply that related data and behavior are not being kept in one place.

Heuristic #3.4: Beware of classes, which have too much non-communicating behavior, i.e. methods which operate on a proper subset of the data members of a class. God classes often exhibit lots of non-communicating behavior.

Heuristic #3.5: In applications, which consist of an object-oriented model interacting with a user interface, the model should never be dependent on the interface. The interface should be dependent on the model.

Heuristic #3.6: Model the real world whenever possible. (This heuristic is often violated for reasons of system intelligence distribution, avoidance of god classes, and the keeping of related data and behavior in one place).

Heuristic #3.7: Eliminate irrelevant classes from ones design.

Heuristic #3.8: Eliminate classes that are outside the system.

Heuristic #3.9: Do not turn an operation into a class. Be suspicious of any class whose name is a verb or derived from a verb. Especially those which have only one piece of meaningful behavior (i.e. do not count sets, gets, and prints). Ask if that piece of meaningful behavior needs to be migrated to some existing or undiscovered class.

Heuristic #3.10: Agent classes are often placed in the analysis model of an application. During design time, many agents are found to be irrelevant and should be removed.

Heuristic #4.1: Minimize the number of classes with which another class collaborates.

Heuristic #4.2: Minimize the number of message sends between a class and its collaborator.

Heuristic #4.3: Minimize the amount of collaboration between a class and its collaborator, i.e. the number of different messages sent.

Heuristic #4.4: Minimize fanout in a class, i.e. the product of the number of messages defined by the class and the messages they send.

Heuristic #4.5: If a class contains objects of another class then the containing class should be sending messages to the contained objects, i.e. the containment relationship should always imply a uses relationship.

Heuristic #4.6: Most of the methods defined on a class should be using most of the data members most of the time.

Heuristic #4.7: Classes should not contain more objects than a developer can fit in his or her short term memory. A favorite value for this number is six.

Heuristic #4.8: Distribute system intelligence vertically down narrow and deep containment hierarchies.

Heuristic #4.9: When implementing semantic constraints, it is best to implement them in terms of the class definition. Often this will lead to a proliferation of classes in which case the constraint must be implemented in the behavior of the class, usually, but not necessarily, in the constructor.

Heuristic #4.10: When implementing semantic constraints in the constructor of a class, place the constraint test in the constructor as far down a containment hierarchy as the domain allows.

Heuristic #4.11: The semantic information on which a constraint is based is best placed in a central third-party object when that information is volatile.

Heuristic #4.12: The semantic information on which a constraint is based is best decentralized among the classes involved in the constraint when that information is stable.

Heuristic #4.13: A class must know what it contains, but it should never know who contains it.

Heuristic #4.14: Objects which share lexical scope, i.e. those contained in the same containing class, should not have uses relationships between them.

Heuristic #5.1: Inheritance should only be used to model a specialization hierarchy.

Heuristic #5.2: Derived classes must have knowledge of their base class by definition, but base classes should not know anything about their derived classes.

Heuristic #5.3: All data in a base class should be private, i.e. do not use protected data.

Heuristic #5.4: Theoretically, inheritance hierarchies should be deep, i.e. the deeper the better.

Heuristic #5.5: Pragmatically, inheritance hierarchies should be no deeper than an average person can keep in their short term memory. A popular value for this depth is six.

Heuristic #5.6: All abstract classes must be base classes.

Heuristic #5.7: All base classes should be abstract classes.

Heuristic #5.8: Factor the commonality of data, behavior, and/or interface as high as possible in the inheritance hierarchy.

Heuristic #5.9: If two or more classes only share common data (no common behavior) then that common data should be placed in a class which will be contained by each sharing class.

Heuristic #5.10: If two or more classes have common data and behavior (i.e. methods) then those classes should each inherit from a common base class which captures those data and methods.

Heuristic #5.11: If two or more classes only share common interface (i.e. messages, not methods) then they should inherit from a common base class only if they will be used polymorphically.

Heuristic #5.12: Explicit case analysis on the type of an object is usually an error, the designer should use polymorphism in most of these cases.

Heuristic #5.13: Explicit case analysis on the value of an attribute is often an error. The class should be decomposed into an inheritance hierarchy where each value of the attribute is transformed into a derived class.

Heuristic #5.14: Do not model the dynamic semantics of a class using the inheritance relationship. An attempt to model dynamic semantics with a static semantic relationship will lead to a toggling of types at runtime.

Heuristic #5.15: Do not turn objects of a class into derived classes of the class. Be very suspicious of any derived class for which there is only one instance.

Heuristic #5.16: If one thinks one needs to create new classes at runtime, take a step back and realize that what you are trying to create are objects. Now generalize these objects into a class.

Heuristic #5.17: It should be illegal for a derived class to override a base class method with a NOP method, i.e. a method which does nothing.

Heuristic #5.18: Do not confuse optional containment with the need for inheritance, modeling optional containment with inheritance will lead to a proliferation of classes.

Heuristic #5.19: When building an inheritance hierarchy try to construct reusable frameworks rather than reusable components.

Heuristic #6.1: If one has an example of multiple inheritance in ones design, assume one has made a mistake and prove otherwise.

Heuristic #6.2: Whenever there is inheritance in an object-oriented design ask oneself two questions: 1) Am I a special type of the thing I'm inheriting from? and 2) Is the thing I'm inheriting from part of me?

Heuristic #6.3: Whenever one has found a multiple inheritance relationship in an object oriented design be sure that no base class is actually a derived class of another base class, i.e. accidental multiple inheritance.

Heuristic #7.1: When given a choice in an object-oriented design between containment relationship and an association relationship, choose the containment relationship.

Heuristic #8.1: Do not use global data or functions to perform bookkeeping information on the objects of a class, class variables or methods should be used instead.

Heuristic #9.1: Object-oriented designers should never allow physical design criteria to corrupt their logical designs. However, very often physical design criteria is used in the decision making process at logical design time.

Heuristic #9.2: Do not change the state of an object without going through its public interface.

## ***A.4 MeTHOOD Heuristics***

Heuristic #1: A class in a containment hierarchy should only depend from its child classes.

Heuristic #2: Every attribute should be hidden within its class.

Heuristic #3: A client-server dependency between two classes should not lead to dependencies from the server to the client.

Heuristic #4: Avoid dependencies from database classes to their clients.

Heuristic #5: A class should capture one and only one key abstraction with all its information and all its behavior.

Heuristic #6: Do not create unnecessary classes to model roles.

Heuristic #7: Avoid pure accessor methods.

Heuristic #8: Avoid additional relationships from base classes to their derived classes.

Heuristic #9: Avoid classes with properties implying redundancies.

Heuristic #10: Avoid multivalued dependencies.

Heuristic #11: Convert associations, and uses relationships in the strongest containment relationship wherever possible.

Heuristic #12: Avoid contained instances that have to be modified concurrently.

Heuristic #13: All properties of the base class interface must be usable in instances of its derived classes in every location where a base class instance is expected.

Heuristic #14: Common properties of instances should be defined in a single location.

Heuristic #15: Instable classes should not be base classes.

Heuristic #16: Do not misuse inheritance for sharing attributes.

Heuristic #17: The overloading should define only differences to the overloaded method.

Heuristic #18: Avoid case analysis on properties of instances.

Heuristic #19: Prefer typing by attribute before typing by inheritance.

Heuristic #20: A method should use only classes of attributes of its class, classes of its parameters, or classes of instances locally created.

## A.5 Code Smell Taxonomy

The following taxonomy was assembled by [Mantyla 2006].

It can be used to understand how the different code smells impact on different parts of a system's design.

Group name	Smells in group	
<b>The Bloaters</b>	<ul style="list-style-type: none"> <li>-Long Method</li> <li>-Large Class</li> <li>-Primitive Obsession</li> <li>-Long Parameter List</li> <li>-DataClumps</li> </ul>	
<p>Bloater smells represent something that has grown so large that it cannot be effectively handled.</p> <p>Primitive Obsession is actually more of a symptom that causes bloats than a bloat itself. The same holds for Data Clumps. When a Primitive Obsession exists, there are no small classes for small entities (e.g. phone numbers). Thus, the functionality is added to some other class, which increases the class and method size in the software.</p> <p>With Data Clumps there exists a set of primitives that always appear together (e.g. 3 integers for RGB colors). Since the data items are not encapsulated in a class, this increases the sizes of methods and classes.</p>		
<b>The Object-Orientation Abusers</b>	<ul style="list-style-type: none"> <li>-Switch Statements</li> <li>-Temporary Field</li> <li>-Refused Bequest</li> <li>-Alternative Classes with Different Interfaces</li> </ul>	
<p>The common denominator for the smells in the Object-Orientation Abuser category is that they represent cases where the solution does not fully exploit the possibilities of object-oriented design.</p> <p>For example, a Switch Statement might be considered acceptable or even good design in procedural programming, but is something that should be avoided in object-oriented programming. The situation where switch statements or type codes are needed should be handled by creating subclasses. Parallel Inheritance Hierarchies and Refused Bequest smells lack proper inheritance design, which is one of the key elements in object-oriented programming.</p>		

The Alternative Classes with Different Interfaces smell lacks a common interface for closely related classes, so it can also be considered a certain type of inheritance misuse. The Temporary Field smell means a case in which a variable is in the class scope, when it should be in method scope. This violates the information hiding principle.

**The Change Preventers**

- Divergent Change
- Shotgun Surgery
- Parallel Inheritance Hierarchies

Change Preventers are smells is that hinder changing or further developing the software

These smells violate the rule suggested by Fowler and Beck, which says that classes and possible changes should have a one-to-one relationship. For example, changes to the database only affect one class, while changes to calculation formulas only affect the other class.

The Divergent Change smell means that we have a single class that needs to be modified by many different types of changes. With the Shotgun Surgery smell the situation is the opposite, we need to modify many classes when making a single change to a system (change several classes when changing database from one vendor to another)

Parallel Inheritance Hierarchies, which means a duplicated class hierarchy, was originally placed in OO-abusers. One could also place it inside of The Dispensables since there is redundant logic that should be replaced.

**The Dispensables**

- Lazy class
- Data class
- Duplicate Code
- Dead Code,
- Speculative Generality

The common thing for the Dispensable smells is that they all represent something unnecessary that should be removed from the source code.



This group contains two types of smells (dispensable classes and dispensable code), but since they violate the same principle, we will look at them together. If a class is not doing enough, it needs to be removed or its responsibility needs to be increased. This is the case with the Lazy class and the Data class smells. Code that is not used or is redundant needs to be removed. This is the case with Duplicate Code, Speculative Generality and Dead Code smells.



<b>The Couplers</b>	<ul style="list-style-type: none"> <li>-Feature Envy</li> <li>-Inappropriate Intimacy</li> <li>-Message Chains</li> <li>-Middle Man</li> </ul>	
<p>This group has four coupling-related smells.</p> <p>One design principle that has been around for decades is low coupling (Stevens et al. 1974). This group has 3 smells that represent high coupling. The Middle Man smell on the other hand represents a problem that might be created when trying to avoid high coupling with constant delegation. Middle Man is a class that is doing too much simple delegation instead of really contributing to the application.</p> <p>The Feature Envy smell means a case where one method is too interested in other classes, and the Inappropriate Intimacy smell means that two classes are coupled tightly to each other. Message Chains is a smell where class A needs data from class D. To access this data, class A needs to retrieve object C from object B (A and B have a direct reference). When class A gets object C it then asks C to get object D. When class A finally has a reference to class D, A asks D for the data it needs. The problem here is that A becomes unnecessarily coupled to classes B, C, and D, when it only needs some piece of data from class D. The following example illustrates the message chain smell: A.getB().getC().getD().getTheNeededData()</p> <p>Of course, I could make an argument that these smells should belong to the Object-Orientation abusers group, but since they all focus strictly on coupling, I think it makes the taxonomy more understandable if they are introduced in a group of their own.</p>		

## A.6 Refactoring Tool Support Comparison

### Refactoring Comparison

Product			IDEA	Eclipse	Jbuilder	Jbuilder	CodePro	Jrefactory	CodeGuide	Jfactor
Edition	Personal 2.0	Developer (1) 2.0	4.5	3.0	Foundation & Developer 2005	Enterprise 2005	Studio			VisualAge for Java
Company	Refactorit	Refactorit	Intelli	Freeware	Borland	Borland	Instantiations	Freeware	OmniCore	Instantiations
<b>Refactoring</b>										
Add delegate methods		✓	✓	✓						
Add throw clause and surround with try-catch exceptions					✓	✓				
Automatic conversion to generic Java 1.5 code			✓		✓				✓	
Change Method Signature		✓	✓	✓	✓	✓	✓			
Clean Imports		✓	✓	✓		✓			✓	
Convert Anonymous Class to Inner			✓	✓						
Convert autoboxing and auto-unboxing primitive and OO counterparts					✓	✓				
Convert existing loops to enhanced loops					✓	✓				
Copy/Clone class			✓	✓						
Create factory method		✓	✓	✓						
Delegate to Instance		✓		✓	✓	✓				
Encapsulate Field		✓	✓	✓						✓
Extract Interface		✓	✓	✓	✓	✓	✓	✓		✓
Extract Method	✓	✓	✓		✓	✓	✓	✓	✓	✓
Find definition for a symbol	✓	✓			✓	✓				
Inline Method		✓	✓	✓						✓
Inline Variable and Constant		✓	✓	✓						
Introduce Constant		✓ (2)	✓	✓						
Introduce field			✓	✓	✓	✓				
Introduce Parameter			✓	✓						
Introduce Superclass		✓	✓		✓	✓	✓			✓
Introduce Variable (3)		✓	✓	✓	✓	✓			✓	✓
Make method static		✓	✓							
Minimize Access Rights		✓	✓							
Move non-static method / field		✓	✓	✓						
Move static method / field		✓	✓	✓	✓	✓		✓	✓	
Moving classes and packages with reference correction	✓	✓	✓	✓						
Parameter add, remove, or reorder		✓		✓	✓	✓				
Promote temporary to field		✓		✓						
Pull down member-method-field		✓	✓	✓	✓	✓	✓	✓		✓
Push up member-method-field		✓	✓	✓	✓	✓	✓	✓		✓
Refactor references not referenced in compiler	✓	✓			✓	✓				
Refactoring - cancel, undo, redo	✓	✓	✓	✓	✓	✓			✓	
Refactoring in text files		✓	✓	✓						
Remove Parameter		✓		✓	✓	✓				
Rename	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Safe delete			✓						✓	✓
Search all references	✓	✓		✓	✓	✓				
Use interface where possible		✓	✓	✓						

Comparison compiled by [Refactorit 2006].