Detecting Inheritance Hierarchy Smells

IOANNIS ZIAMOS

This dissertation was submitted in part fulfilment of requirements for the degree of MSc Advanced Computer Science

DEPT. OF COMPUTER AND INFORMATION SCIENCES UNIVERSITY OF STRATHCLYDE

August 2017

(INTENTIONALLY BLANK PAGE)

DECLARATION

This dissertation is submitted in part fulfilment of the requirements for the degree of MSc of the University of Strathclyde.

I declare that this dissertation embodies the results of my own work and that it has been composed by myself. Following normal academic conventions, I have made due acknowledgement to the work of others.

I declare that I have sought, and received, ethics approval via the Departmental Ethics Committee as appropriate to my research.

I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to provide copies of the dissertation, at cost, to those who may in the future request a copy of the dissertation for private study or research.

I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to place a copy of the dissertation in a publicly available archive.
(please tick) Yes [ ] No [ ]

I declare that the word count for this dissertation (excluding title page, declaration, abstract, acknowledgements, table of contents, list of illustrations, references and appendices is:

I confirm that I wish this to be assessed as a Type 1 2 3 4 5
Dissertation (please circle)

Signature:

Date:

(INTENTIONALLY BLANK PAGE)

Abstract

Technical debt in software systems takes many forms, some of these are the various software smells that have been proposed and studied in the last 20 years. A recent software engineering textbook "Refactoring For Software Design Smells: Managing Technical Debt" (Suryanarayana,Samarthyam and Sharma, 2014) proposes a formal set of such software patterns. The aim of this work is to explore the hierarchical smell patterns proposed in chapter 6 of the textbook.

After conducting a review of the relative literature, a software tool was built to aid with inspection of software systems and specifically detect instances of these smells in Java projects. These Java systems came from the Qualitas Corpus (Tempero, 2010), a curated collection of software systems intended for empirical studies. A selection of these systems was analysed with the tool and inspected manually to detect a subset of the proposed smells and expose how common they are.

The findings of this study reveal that these smells do exist in software systems and identify examples of specific ways in which they are commonly introduced. These examples show that in many cases these smells are a result of violating object oriented software engineering guidelines and enabling techniques. The work proposes an expansion of this study to include more systems in different programming languages and some possible redefinitions of the smells so that they can produce more useful results in practice.

(INTENTIONALLY BLANK PAGE)

Acknowledgments

I thank Dr Murray Wood of the University of Strathclyde for supervising this work and Dr Ewan Tempero of the University of Auckland for providing access to the Qualitas Corpus.

(INTENTIONALLY BLANK PAGE)

Table of Contents

1 Introduction 1

2 Literature Review 4

2.1 Useful Definitions 4

2.2 Inheritance metrics 5

2.3 Inheritance guidelines 7

2.4 Software Smells 8

2.5 Refactoring 10

2.6 Hierarchy Smells 10

2.7 Apply Meaningful Classification 11

2.7.1 Missing Hierarchy 11

2.7.2 Unnecessary Hierarchy 12

2.8 Apply meaningful generalisation 13

2.8.1 Unfactored Hierarchy 13

2.8.2 Wide Hierarchy 14

2.8.3 Speculative Hierarchy 15

2.8.4 Deep Hierarchy 15

2.9 Ensure Substitutability 16

2.9.1 Rebellious Hierarchy 16

2.9.2 Broken Hierarchy 17

2.10 Avoid redundant paths 17

2.10.1 Multipath Hierarchy 17

2.11 Ensure proper ordering 18

2.11.1 Cyclic Hierarchy 18

3 Research Method 21

3.1 Qualitas Corpus 21

3.2 Selection Criteria 21

3.3 Detection Tool 22

3.4 Manual Examination 22

3.5 PMD's Copy/Paste Detector 23

3.6 Result Format 23

4 Detection Tool 25

4.1 Extracting information from Java source files 25

4.2 Internal Representation of the Hierarchy Tree 26

4.3 Detecting Hierarchical Smells 26
4.3.1 Deep Hierarchy 26
4.3.2 Wide Hierarchy 27
4.3.3 Multipath Hierarchy 27
4.3.4 Cyclic Hierarchy 27
4.4 Use of the tool 27
4.4.1 Usage 27
4.4.2 Examples 28

5 Results and Analysis 29
5.1 Java Runtime Environment 1.6.21 Class Libraries 29
5.1.1 Description 29
5.1.2 Summary 30
5.1.3 Findings 30
5.1.4 Conclusions 32
5.2 Eclipse SDK 4.3 32
5.2.1 Description 32
5.2.2 Summary 33
5.2.3 Findings 33
5.2.4 Conclusions 36
5.3 JHotDraw 7.5.1 36
5.3.1 Description 36
5.3.2 Summary 37
5.3.3 Findings 37
5.3.4 Conclusions 39
5.4 Weka 3.7.9 39
5.4.1 Description 39
5.4.2 Summary 39
5.4.3 Findings 39
5.4.4 Conclusion 41
5.5 Azureus 4.8.1.2 42
5.5.1 Description 42
5.5.2 Summary 42
5.5.3 Findings 42
5.5.4 Conclusions 43
5.6 FreeCol 0.10.7 43
5.6.1 Description 43
5.6.2 Summary 43

5.6.3 Findings 44
5.6.4 Conclusions 46
5.7 Marauroa 3.8.1 46
5.7.1 Description 46
5.7.2 Summary 47
5.7.3 Findings 47
5.7.4 Conclusions 47
5.8 ArgoUML 0.34 48
5.8.1 Description 48
5.8.2 Summary 48
5.8.3 Findings 48
5.8.4 Conclusions 50
5.9 FreeMind 0.9.0 50
5.9.1 Description 50
5.9.2 Summary 50
5.9.3 Findings 50
5.9.4 Conclusions 51
5.10 Apache Ant 1.8.4 51
5.10.1 Description 51
5.10.2 Summary 51
5.10.3 Findings 52
5.10.4 Conclusions 54

6 Discussion 55
6.1 Summary of Findings 55
6.2 Findings per Hierarchical Smell 55
6.2.1 Deep Hierarchy 55
6.2.2 Wide Hierarchy 55
6.2.3 Multipath Hierarchy 56
6.2.4 Cyclic Hierarchy 57
6.3 Evaluation of the Tool 58

7 Recommendations and Conclusions 61

8 References 65

Table of Figures

Figure 4.1 28

Figure 5.1 31

Figure 5.2 31

Figure 5.3 32

Figure 5.4 33

Figure 5.5 34

Figure 5.6 35

Figure 5.7 35

Figure 5.8 36

Figure 5.9 38

Figure 5.10 40

Figure 5.11 40

Figure 5.12 41

Figure 5.13 42

Figure 5.14 44

Figure 5.15 45

Figure 5.16 48

Figure 5.17 49

Figure 5.18 52

Figure 5.19 52

Figure 5.20 54

1 Introduction

A programming paradigm is defined by its constraints and features. The presence or lack of support for different features defines the programming paradigm(s) supported by a programming language. In his discussion on the object oriented paradigm in Smalltalk, Rentch (1982) contends that although the object oriented approach, typically, facilitates objects, data hiding, encapsulation and modularisation; a dogmatic definition may be misleading and other features such as inheritance, a form of sharing between objects, are not key requirements. This view is also held by Lopes, in her recent programming guideline textbook (Lopes, 2014) which attempts to informally identify the elements of each programming style. There is a difference in opinion regarding this topic and the idea that inheritance is not a requirement is refuted by the definition of object oriented programming provided by the ISO standard which describes it as "pertaining to a technique or a programming language that supports objects, classes, and inheritance" (ISO, 2015) but also by the online account of the person who coined the term, Alan Kay (Kay, 2003). Meyer (1997) also explains that not only is inheritance a key invariant, but multiple inheritance specifically is required to define types properly.

Despite not being considered a key invariant by all researchers, inheritance is provided by most general-purpose programming languages such as C++, C#, Java and Python. These are multi-paradigm languages which support the object-oriented programming style. In these, additional features such as composition, inheritance and polymorphism are provided to support reuse in the object-oriented approach. The way in which each language implements these features differs. Java, C# and PHP only provide "Single Inheritance", where each class can have only one parent-class. In contrast, others such as C++ and Python support "Multiple Inheritance" which allows a class to have multiple parent-classes. Other languages that provide support for objects do not facilitate inheritance at all, an example is the Go programming language which was implemented under the philosophy that implementing interfaces provides a sufficient alternative (Google, 2010).

The difference in the way inheritance is implemented mostly results from differences in opinion and the intent of each programming language. Because of the differences between opinion and implementation, it is useful to examine how inheritance is being used in practice. This can be achieved by evaluating the metrics proposed by Chidamber and Kemerer (1994) or by examining a designs adherence to commonly accepted guidelines such as those proposed by Robert C. Martin (2005) known as SOLID or GRASP by Craig Larman (1997). A recently released textbook (Suryanarayana,Samarthyam and Sharma, 2014) proposes a set of design choices which can result in future issues in an object oriented software system. These are commonly known as "Smells" and those pertaining to inheritance hierarchies are called hierarchical smells.

The term smell was introduced by Fowler and Beck (1999) and describes structures in code that suggest the need for refactoring. The implication is that if these artefacts of bad design remain in place, they will become a larger problem than they already are over the life-cycle of a software system because they are poor choices in the core structure of the software rather than simple mistakes that will lead to bugs. Some of these pertain to the size of code such as a method being too long or a class being too lazy (not providing enough functionality), others suggest that code is duplicated unnecessarily or exists in the wrong class. Such guidelines are not strictly defined on purpose and are usually accompanied by proposed refactoring techniques that can be applied to resolve the potential issues. Generally, these involve moving code, removing duplicate code and adjusting hierarchy trees.

Hierarchical smells are a category of smell patterns that emerge because of poor design of the class hierarchies in software systems. A recent software engineering textbook: Refactoring for software design smells: managing technical debt (Suryanarayana,Samarthyam and Sharma, 2014) has proposed a list of these and includes descriptions, examples and proposed refactoring options for each one. It also lists the rationale for their classification as smells, as usually they are considered to manifest themselves when key enabling techniques and principles of software design are violated. Examining these newly defined smells would constitute a useful part of analysing how programmers use class inheritance. It can complement studies on object-oriented software metrics and use of design patterns to determine if inheritance is being misused in ways that violate core principles. These hierarchical smells are listed below:

- **Missing Hierarchy,** the use of conditional statements in place of a hierarchy.
- **Unnecessary Hierarchy,** the application of non-meaningful hierarchy.
- **Unfactored Hierarchy,** the presence of duplicate classes within a hierarchy.
- **Wide Hierarchy,** a hierarchy that is excessively wide.
- **Speculative Hierarchy,** a hierarchy based on imaginative needs.
- **Deep Hierarchy,** an excessively deep hierarchy tree.
- **Rebellious Hierarchy,** the rejection of superclasses methods by a subclass even when there is an “IS-A” relationship.
- **Broken Hierarchy,** a hierarchy that lacks an IS-A relationship between classes.
- **Multipath Hierarchy,** when a class inherits a superclass or interface through multiple paths in the hierarchy.
- **Cyclic Hierarchy,** the dependence of a superclass on a subclass.

The first objective of this project is to review the nature of these smells and present their definitions in the context of previous methods used for critiquing object-oriented designs. Using this knowledge, it then presents a study of popular open source object oriented systems and how such smells are present in them. As the formal account of these definitions is relatively new, this procedure is useful for evaluating the practical use of detecting these patterns in the real world. The procedure and methods used to perform the study will be detailed in a dedicated chapter. The distinct steps for achieving the objectives were as listed in order:

- Perform a study of the nature of hierarchical smell patterns and literature to determine which and how can be detected automatically.
- Develop a software tool that can perform the task of automatically detecting the selected hierarchical smell patterns.
- Test the tool and evaluate its effectiveness in performing the task.
- Select a suitable set of object-oriented Java systems to analyse for the study.
- Perform the analysis and assess how common these smells are and determine whether these patterns are useful in practice.
- Reflect on the findings and utility of the tool with the intent of drawing conclusions and proposing future work

What follows is a review of the required background literature and related definitions which the subsequent study is based upon.

2 Literature Review

The process of designing and implementing a software tool that can detect hierarchical smell patterns requires a review of relevant literature and related work with the aim of establishing the direction and research goals of the study. All the smells defined in the manual cannot be detected automatically, this is mainly because of their subjective or general definition and the considerable time constraints for performing the study. The literature review presents a background of how inheritance relates to the object-oriented style, the necessary definitions and an overview of inheritance metrics and guidelines. Some background to code smells and techniques used to refactor them is also provided. This is followed by an analytical presentation of the hierarchical smell patterns as introduced in Refactoring for software design smells: managing technical debt (Suryanarayana,Samarthyam and Sharma, 2014) which for the rest of this work may be referred to as the "textbook". The review intends to reveal a subset of the hierarchical smells that can be detected automatically and define a research approach that can be applied to determine their presence in open source Java software systems.

2.1 Useful Definitions

Inheritance and sub-typing are two closely related concepts in software engineering. Inheritance is a feature present in many programming languages, a class or object "inherits" from another when it uses its state (data) and/or behaviour (functionality) (Oracle, 2012). Subtyping is a separate feature in which a type is expected to be able to substitute another type without breaking the behaviour of the program and is a form of polymorphism, which is the provision of a single interface to entities of different types (Stroustrup, 2012). In most programming languages, the act of inheriting attributes also means that the inheriting class (child class) is a subtype of the class it inherits from (known as the parent class or superclass). Because the subtype relation is based on inheritance in most statically typed languages that support the object oriented paradigm (Cook,Hill and Canning, 1989) there is some ambiguity, and the two features are often confused.

The focus of this study is on hierarchical smells in the Java programming language. This is because the proposed hierarchical smells and their examples are defined in this language, even though the smells are also intended to be applicable to C# and C++. Java uses the "extends" keyword for establishing a subclass relationship. The type of inheritance supported in Java is known as single inheritance in which a class is permitted to inherit from only one other class. Additionally, in Java, every class necessarily has a parent class. This means that if a parent is not explicitly defined, it implicitly inherits any fields and methods defined in the "Object" class which is a base class that provides basic functions (Gosling *et al.*, 2015). The fact that only single inheritance is supported and the "Object" class is a superclass of all other classes means that the type hierarchy in all Java software systems can be represented as a single-root tree structure.

Although abstract classes can be used to define interfaces in Java, a separate reference type is provided called the "Interface". The Java interface historically contains only method definitions, static methods and constant data fields. Java version 8 also adds support for default methods which can provide a default inheritable behaviour. Interfaces are used in Java by means of the "implements" keyword. A class can implement multiple interfaces and inherits those implemented in a superclass. Interfaces can be used to loosen the coupling between types in a Java system and to replace some of the functionality that multiple type inheritance would provide and are inherited by subclasses. Java interfaces are present in the definitions of hierarchical smells and are used to define several of them.

For the requirements of defining hierarchical smells, it is also necessary to explain the way dependencies are defined by the relevant programming textbook (Suryanarayana,Samarthyam and Sharma, 2014). The term is used to describe instances where a class "knows" about another one. There are three ways in which this can occur. These are when a class contains an object of another class, a class refers to the type name of another class or when a class accesses data members or methods of the other class. This resembles coupling as described in the literature (Chidamber and Kemerer, 1991) however the relationship is transitive, meaning that if A is dependent on B and B is dependent on C then A is dependent on C, however, A and C would not be coupled in the traditional definition.

2.2 Inheritance metrics

Inheritance metrics are a key dimension of studying inheritance in object oriented systems. The initial work on defining metrics for object oriented systems (Chidamber and Kemerer, 1994) proposed six different measurements that measure the size or complexity of an object oriented architecture. These are the Weighted Methods Per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling Between Object Classes (CBO), Response For a Class (RFC) and Lack of Cohesion in Methods (LCOM). Because some of these metrics are useful in the process of detecting hierarchical smell patterns, short definitions are provided:

Weighted Methods Per Class, this measurement is calculated as the sum of the complexity of all the methods provided by a single class. A specific measurement of the complexity of a method, however, is not provided and when calculating the metric one must select a suitable complexity method that can be summed up in this manner. This is proposed as a metric because the sum of the complexity of a class can be used to gauge the effect of a parent class on subclasses since it will inherit all of its methods (Chidamber and Kemerer, 1991), which can point to issues with maintaining and reusing the class.

Depth of Inheritance Tree, DIT is an integer that represents how many ancestor classes a single class in an inheritance tree (or lattice) has. In cases where multiple inheritance is present the longest path is used. In Java, multiple inheritance is not available but this number is always inflated by one extra class due to the requirement of inheriting the class "Object" at the top level of the hierarchy putting the minimum value for the metric at 1 instead of 0, this is the same in Smalltalk which the original metrics were defined for. This metric is evaluated because of the effect the higher number of parent classes have on a class. A class deeper in the hierarchy is more likely to contain more inherited and overridden methods, thus resulting in higher complexity. A design with deep hierarchies can also point to a higher complexity of the overall design of the system.

Number of Children, NOC represents the number of immediate subclasses a class has. High values for this metric most likely indicate higher levels of reuse but also can point to misuse of subclassing because correct abstraction of the classes would lead to intermediate types which could properly provide the common methods and fields. This metric is proposed to give an idea of the potential influence each parent class has on the design.

Coupling Between Object Classes, CBO is the number of other classes to which one is coupled. A class is said to be coupled to another when it accesses methods or data from it. The reason for calculating this metric is because coupling prevents independent reuse of the class in a separate application, changes in one class affect the other and according to the authors, results in more complicated requirements for testing.

Response For a Class, RFC indicates the size of the set of methods that can be invoked as a response to a received message by an object of the class, this can include methods called from outside the class. This proposed metric can be used to indicate the complexity of testing and debugging a class if the number of method calls is high. It can also indicate the overall complexity and the expected time for testing the class.

Lack of Cohesion in Methods, this metric expresses the similarity between some class's methods by calculating their cohesion. Cohesion indicates how specific the scope of actions a class performs are to the concept it implements. High cohesion alongside low coupling are considered desirable traits in object oriented systems (Larman, 1997), a high measure of it promotes encapsulation. A low degree of cohesion which can be indicated by LCOM is documented to imply that a classes' functionality may need to be split into multiple classes and may imply that due to its mixed purpose the class has a high level of complexity.

Some of these metrics can be directly tied to the hierarchical smells and the way they are related will be detailed in a further section. These metrics are the DIT and NOC metrics and have been studied further to determine reasonable thresholds for their values. The DIT metric has been studied (Ferreira *et al.*, 2012) and it was identified that 2 is a reasonable value for DIT. This study contends that the value is compatible with the guideline "favour composition over inheritance" and was reached by studying a large number of Java systems and observing how deep inheritance trees are in practice. Sherif and Sanderson (1998) found that the numbers for DIT in the systems studied do not surpass 4 and that the maximum value for NOC should be 6.

2.3 Inheritance guidelines

In addition to these metrics which can be used to determine hierarchical smells, another way of determining the smell patterns was the violation of inheritance guidelines. The most widely known ones are remembered by the mnemonic "SOLID". They are:

- Single responsibility principle (SRP): "A class should encapsulate only once concept".
- Open/closed principle (OCP): "A class should be open for extension but closed for modification".
- Liskov's substitution principle (LSP): "Objects in a program should be replaceable with instances of their subtypes without altering the correctness of the program".
- Interface segregation principle (ISP): "Many client-specific interfaces are better than one general-purpose interface".
- Dependency inversion principle (DIP): "One should depend upon abstractions, not concretions".

A summary of the enabling techniques provided in the textbook (Suryanarayana,Samarthyam and Sharma, 2014) is also useful since they are all pertinent to object hierarchies.

- **Apply meaningful classification:** This enabling technique suggests that classification should be applied to objects with the more general types at a higher level in the hierarchy than specialised ones. It recommends that when applying classification, commonalities should be captured in parent classes and in order to achieve a more meaningful result it is best to observe the commonalities in methods rather than data.
- **Apply meaningful generalisation:** This enabling technique recommends that when elements from subtypes are factored out into supertypes, the generalisation should make logical sense. The subtype must always maintain an "IS-A" (subsumption) relationship with the supertype. Generalisation should result in less duplicated code in a system.
- **Ensure substitutability:** The manual references this enabling technique and references Liskov's Substitution Principle. This principle dictates that a subclass should be able to be

substituted in place of a parent class without altering the behaviour of the program. This allows for dynamic polymorphism and improves the extensibility, modularity and reusability of the design.

- **Avoid redundant paths:** This enabling technique is more related to object oriented system designs using multiple inheritance. What it means is that a class should only inherit from another once, rather than multiple times through different paths. In Java inheritance trees this does not occur for class inheritance but it may when inheriting interfaces since multiple implementation of these is possible.
- **Ensure proper ordering:** This final guideline suggests that within a taxonomy of classes relationships should maintain consistency. A class should not depend, or even know, of a subtype. This relates to the dependency inversion principle.

Often, such guidelines and enabling techniques overlap as they are oriented towards facilitating a similar object-oriented design outcome. Although they are well documented, in many cases, they are sidestepped or ignored, a behaviour which may result in what is known as technical debt. Technical debt (also known as design debt) (Technopedia, 2017) is a term coined by Ward Cunningham which describes cases in software engineering where convenience and development time are prioritised over application of the best overall solution. Cunningham does not say that such debt should not be incurred but suggests that it should be "repaid" as soon as possible, otherwise the issue will develop into a larger problem which will be harder to manage in the future.

2.4 Software Smells

Software smells, a concept introduced by Kent Beck and Martin Fowler (1999), can be thought of as patterns in software systems which suggest the possibility of refactoring. Their research in refactoring proposed various techniques for improving the design of software. These patterns were introduced so that one could identify when to apply refactoring. The term implies that if a software project is left unmaintained and required improvements are postponed, it will result in the smells getting worse in the same manner as a physical odour would. One could also think of software smells as common instances of technical debt that can to some degree be identified and formalised, even though when identifying these "smells" there is specific mention that precise criteria are not given and that identifying smell patterns requires a measure of intuition.

The idea of software smells was introduced with 22 example smell patterns. Mantyla, Vanhanen and Lassenius (2003), provided a higher-level taxonomy of them and studied the inter-relation between them. In their study, they propose 7 categories that smells can be grouped into and explain that a flat taxonomy of this kind aids in their study. These are:

- **Bloaters,** instances where code abstractions have grown to a size where they cannot be effectively handled.
- **Object-Orientation Abusers,** these are cases in which possibilities of object oriented design are not fully utilised.
- **Change Preventers,** these are code structures which "considerably hinder" modification of a software.
- **Dispensables,** these are instances of things that can be removed entirely.
- **Encapsulators,** these deal with data communication mechanisms or encapsulation. Commonly these smells are introduced in pairs and decreasing one of them would result in an increase in the other, requiring some trade-off.
- **Couplers,** these are usually instances of smells that pertain to tight coupling between classes.
- **Other,** this category is for other miscellaneous smells that can't fit into any other categories such as the smell described as "too many comments".

Many of these software patterns have been studied to various degrees and for some of them, static analysis tools have been developed using a variety of approaches. Van Emden and Moonen (2002) present an approach to detecting smells by using a coloured graph to represent data extracted from a software system. This is a computer-aided manual approach in which the intention is to indicate which parts of the system concentrate the most instances of smells and would benefit the most from refactoring. Their tool "jCosmo" can detect the "Switch Statement" smell as introduced previously (Beck,Fowler and Beck, 1999) and also detect "Type Cast" and "instanceOf" smells which appear to have been introduced by the same article.

Another venture in detecting smells is a series of publications which introduced and expanded the capabilities of the tool JDeodorant. The smells detected were instances of disproportionately large classes or methods (Fokaefs *et al.*, 2011), "Feature Envy" (Fokaefs,Tsantalis and Chatzigeorgiou, 2007), duplicate code (Mazinanian *et al.*, 2016) and type-checking smells (Tsantalis,Chaikalis and Chatzigeorgiou, 2008). Some of the techniques employed constitute a novel approach to the task, feature envy is detected by detecting the distance between a method and a class. Large classes, commonly known as god classes, are detected using a hierarchical clustering algorithm. Both techniques could possibly be applied to other instances of smells.

These studies were both performed on Java systems and display several possible creative techniques for inferring smell presence. They also show that innovative approaches or human involvement in the detection process may be required in many cases due to the subjective nature of software smells and their loose definitions. Studying the literature also reveals that definitions of smells are being continuously expanded and added. An example of this are articles such as Van Deursen *et al.* (2001) which introduced 11 new smell definitions. An important aspect of proposing a new kind of smell is to explain how it fits in the already established taxonomy and to pair it with an appropriate refactoring solution without which it is not useful.

2.5 Refactoring

A small set of the refactoring techniques introduced in the same textbook as the original smells (Beck,Fowler and Beck, 1999) are used for resolving hierarchical smells, these are:

- **Extract Method,** this technique refers to creating a suitably named method out of a code fragment.
- **Pull up Method,** when methods with identical results exist in subclasses they can be moved to the superclass.
- **Push Down Method,** when behaviour provided by a superclass isn't relevant for all the subclasses then it must be moved to the subclasses that need it.
- **Form Template Method,** when two methods in subclasses perform similar steps in the same order but some of those steps differ, a single method can be formed which defines the steps and similar behaviour but defers to the subclasses for the variable behaviour. Then the method can be pulled up to a superclass.
- **Extract Class,** when one class does work that should be done by two, a new class should be created for the secondary behaviour and relevant fields and methods should be moved there.
- **Collapse Hierarchy,** if a superclass and subclass are not very different, they should be merged together.
- **Move Method,** this suggests that if a method is used by more features of another class than the one it is defined in, then a new method with a similar body should be created in the class that uses it the most. The old method should be turned into delegation or removed altogether.
- **Remove Redundant Paths,** this refactoring is not suggested in the relevant chapter of the Beck and Fowler refactoring textbook. It is proposed as a solution to a hierarchical smell and suggests that if a class inherits from another through multiple routes then these extra routes should be removed if possible.

2.6 Hierarchy Smells

Refactoring for software design smells: managing technical debt (Suryanarayana,Samarthyam and Sharma, 2014) is a programming textbook which focuses on 25 instances of technical debt and expands on the smells identified in a previous article by the same authors (Ganesh,Sharma and Suryanarayana, 2013). It is clearly stated that the motivation behind this attempt is not only to add to the list of smells but to also group them into categories, suggest how they are caused and display how they can be addressed by applying what they refer to as proven and sound design principles. The focus of this study is on the hierarchical smells introduced and detailed in Chapter 6 of this textbook. It is worthwhile noting that significant changes to the list of hierarchical smells have been made in comparison to the ones proposed in the initial article.

The smells are introduced with some background on how taxonomies can be used to provide structure so that the complex diversity in real-world entities can be managed when modelled into a real-world software system. It is specified that when referring to hierarchies, this is done referencing Type Hierarchies which express an "IS-A" relationship rather than object hierarchies which express the "Part-Of" relationship. In defining the concept of a hierarchical smell, it is made clear that the focus is on how these instances of software misdesign violate the principles of hierarchy in a taxonomy of classes which have been listed in the previous subsection on inheritance guidelines. Although the coding examples provided are in the Java programming language, due to its popularity, these smells have been proposed with languages such as C# and C++ in mind. Java 7.0 is the version of Java which was used to provide examples of smells in the Java Class Libraries which are provided with the Java Development Kit (JDK). They are presented here in the same order used in the book, which is by grouping them according to the enabling technique which was violated in each case.

2.7 Apply Meaningful Classification

2.7.1 Missing Hierarchy

This smell describes situations in which other programming techniques have been applied, such as conditional statements, to manage variation in behaviour instead of opting for the creation of a type hierarchy which could encapsulate the differences. This is related to, but not the same as, the "Switch Statements" smell which was proposed by Fowler and Beck (1999) who describe such control structures as an inappropriate replacement for polymorphism in object oriented designs. Using "Tagged Types" reveals the potential for an unexploited "IS-A" relationship which can be used as a basis for a hierarchy. In cases where this smell is present the textbook contends that since the "Apply Meaningful Classification" enabling technique was not followed, this results in a missing hierarchy.

The reason that programmers would implement conditional statements over the use of hierarchy are listed as a) a misguided view that this would result in simple or straightforward design (which it possibly does but results in technical debt) b) a procedural approach resembling the style that would be used in a C-like language where such structures are the norm and c) the fact that developers sometimes just overlook inheritance as a design technique. There are two suggested refactoring options for this smell. In cases where multiple implementations have common method calls it is suggested that one should use an interface to abstract the common procedures. In cases where the conditionals can be transformed into classes, then a hierarchy of classes should be constructed so that use of polymorphism can replace the statements.

The impacted quality attributes which will be positively affected by these refactoring options are: a) The resulting code is more understandable because of the relative simplicity of the resulting code compared to long conditional statements, b) new classes can be added without modifying the client code, c) classes are reusable because behaviour is encapsulated within them d) testing is easier because code is isolated effectively, e) the resulting program is more reliable because type based checking can result in subtle bugs as updating part of the code can result in need for changes in other parts which can be overlooked. Also, use of primitives or strings as type codes does not safeguard against invalid type codes which can lead to unexpected behaviour.

Detecting this smell automatically with static analysis may be difficult to perform without human participation. It is, however, possible to search code for conditional statements with common method calls and semi-automatically detect instances of the smell. The existence of conditional statements, of course, does not always infer the existence of a missing hierarchy and the manual concludes that sometimes it may not be possible to avoid them.

2.7.2 Unnecessary Hierarchy

This type of a hierarchical smell appears when inheritance has been applied needlessly. The idea is that inheritance should be applied only when it is deemed meaningful to avoid avoidable complication of the software structure. Alternative terms used to describe this smell are "Taxonomania" (taxonomy-mania) and "Object classes" which imply that the programmer is creating new classes instead of new objects.

The potential reasons provided for a design being over-engineered in such a way are two. Programmers may be over-using subclasses instead of adding a field for a given attribute such as colour (creating types such as RedCar and BlueCar instead of a Car class which has a field for colour). The other reason is a tendency by some designers to overuse taxonomies and force-fit them to their designs even though it is not warranted by the context. As a guide, if functionality between classes is the same and the only thing that differs is their data state then the application of inheritance may not be required. The suggested refactoring is to remove the hierarchy and use a field to encapsulate the difference between the types of the former inheritance structure, another alternative is to look for language features that can replace the hierarchy (enumeration is a suggested in the textbook).

The advantages of performing these refactoring operations resemble those of the missing hierarchy. Needlessly applying inheritance can lead to unneeded complexity, difficulties in extending the design (in the previous example from page 12 one would have to add a new Car type every time a new colour was added to the catalogue) and the introduction of extra requirements for testing as each unnecessary subclass must be tested separately.

This kind of smell may be difficult to detect automatically or by a human with the aid of a computer. It may be possible that classes with a high value for the metric NOC (Number of Children) could be a result of the superfluous creation of classes. Alternatively, one could detect of instances of primitive type names or attributes in the names of defined classes, this could indicate that a separate class has been created to manage each type. This would be an interesting topic to study, especially in cases where the interface (or methods provided) by classes at the same inheritance level are identical or similar.

2.8 Apply meaningful generalisation

2.8.1 Unfactored Hierarchy

The Unfactored Hierarchy is a specific instance of the 'Duplicated Code' smell. The duplicate code smell, as initially proposed suggests that if the same code structure exists in more than one place it will be beneficial to find a way to unify them. The proposed solution is to perform the 'Extract Method' refactoring and invoke the code from both places it existed initially. It is also suggested that in cases where the code is duplicated in sibling (same parent) subclasses then the 'Pull Up Method' refactoring can be applied. In cases where the code has been slightly altered the refactoring of 'Form Template Method' can also be applied so that the differences can be implemented in the child classes.

The Unfactored Hierarchy smell also defines a second form of this smell that occurs when the code is duplicated in super and subtypes. The ways this smell is materialised is either in public methods that have a similar signature or implementation or a combination of both. In these cases, the commonalities can be factored out into a supertype. The rationale given for refactoring this smell is that any common functionality is elevated to the supertype and thus duplication is avoided, also, since this relates to public methods it is practical for common functionality to be part of the supertype interface rather than concrete types so that supertypes can be altered independently.

The common reasons provided for such errors being made is the practice of copying code to create sibling classes. Another more complex reason is that outliers are improperly dealt with, what this means is that although the functionality is required for most sibling classes, some of them do not implement it. The suggested in these cases is to create an intermediate type and add an extra level to the inheritance tree.

This smell can be detected by identifying cloned code by checking each classe's subclasses or across sibling classes. Tools exist that can perform this task and an anecdote in the book describes an instance where many runtime 'instanceOf' checks could be removed from a system by applying "Pull Up Method" refactoring since these checks were a result of an unfactored hierarchy smell. This smell impacts all the usual quality attributes of a system which subsequently means that the program is made more difficult to understand, modify, extend, reuse and test. The conclusion is that single-inheritance languages are more prone to this smell because functionality cannot be factored out into a separate superclass, for Java version 8 this effect is mitigated by the fact that interfaces can implement default methods

2.8.2 Wide Hierarchy

The Wide Hierarchy is described as situations where the hierarchy has too many classes on the same level on inheritance, indicating that there may be too few levels of inheritance resulting from an inadequate application of generalisation. The rationale provided for this smell is that a client will have to maintain references to specific types, something that will make the hierarchy inflexible to change and extension. It also may be a result of the Unfactored Hierarchy smell where common functionality is duplicated instead of moved up to higher levels of the taxonomy.

The potential causes provided by the authors is that some developers simply do not consider intermediate classes when introducing subclasses, the other cause may be that the initial design does not include many subclasses but in later stages when they are introduced, necessary refactoring is not applied. The suggested refactoring for this smell is the "Extract Class" refactoring which is implemented by introducing the necessary intermediate abstractions. Detection of this smell is quite straightforward and can be performed by calculating the NOC values for each class in a project and comparing them to a suitable threshold. The textbook (Suryanarayana,Samarthyam and Sharma, 2014) proposes the threshold for this smell at more than 9 superclasses. Detecting this smell is suggested because its presence affects the understandability, changeability, extensibility and reusability of a design. Testing, however, is not mentioned as an affected quality attribute.

An exception for this smell may be when it results from a programming language feature, such as when it is required to extend a class. In Java, classes without an explicit supertype will extend the class "Object". For this reason, in the subsequent study of software systems class "Object" is expected to have high values for NOC without it necessarily being a software smell. Also, some interfaces or abstract classes which specify a protocol are mentioned to possibly have high values for NOC but not suffer from the smell.

2.8.3 Speculative Hierarchy

The Speculative Hierarchy is a hierarchical smell which is the result of an imagined need instead of a real requirement. What this means is that although generalisation should be applied for current and future needs, there may be instances when these future needs never materialise and the intermediate types are redundant. The problem with future-proofing is that often, predicted needs are inaccurate. Another potential cause the textbook provides for this smell is a tendency for designers to over-engineer their solutions and the types provided do not correspond to actual requirements.

Usually, this smell will result in a hierarchy that looks like a list with each class having only one subclass. The redundant types can complicate the understandability and testability of the software design and the suggested refactoring is to apply the 'collapse hierarchy' refactoring. Detecting this smell could be performed by searching a type hierarchy for instances where types have a single subtype and examining whether this is required. This smell is also part of the initial 22 that were proposed where it is called "Speculative Generality" and it is suggested that it may be found where a class is found to be used only by test cases, in this case, it can be removed entirely.

2.8.4 Deep Hierarchy

This smell is present when a hierarchy is excessively deep. In this case, it is possible that instead of "Applying Meaningful Generalisation" the designer has applied excessive generalisation. The rationale provided for this smell is that if the inheritance hierarchy is excessively deep, then it may be hard to predict the behaviour of code in types in lower levels of the hierarchy. This can possibly occur because a method may have been overridden many times in the hierarchy tree and can confuse a developer, for this reason, deep hierarchies should be avoided. The maximum depth suggested by the textbook is more than 6 levels which can be detected by comparing to the DIT metric. This smell can also be the result of a speculative hierarchy.

Two potential causes for this smell are listed. Some developers may focus excessively on reuse without taking note of the effect this has on understandability and usability of the resulting tree. Also, it can be a result of the application of the speculative hierarchy which is described above because of imagined future needs. The suggested refactoring is to apply the "Collapse Hierarchy" operation if the hierarchy is unnecessary or speculative. The affected quality attributes of this smell are the software systems understandability, extensibility, testability and reliability. Reliability is affected due to the previously mentioned issue that is that it may be difficult to predict which code will be executed. Finally, it is suggested that one must consider that frameworks and software which is oriented towards reuse may need to contain deep hierarchies and in these cases, they are not a smell.

2.9 Ensure Substitutability

2.9.1 Rebellious Hierarchy

The Rebellious Hierarchy smell is present when a type rejects methods defined in one of its parent types. What this means is that although the subclasses maintain an "IS-A" relationship with their superclasses, some of its methods do not. The way in which this rejection occurs is that although the superclass defines specific functionality for a method, the subclass overrides this behaviour by simply throwing an exception, printing a message, returning an error value or replacing with an empty method. These are not the only ways this can happen but are provided as examples. This is another smell introduced in the original 22 and was called "Refused Bequest", the refactoring technique suggested was the "Push Down Method".

The rationale provided is that when a class overrides inherited behaviour, the norm is to specialise it instead of restricting or cancelling it. The argument made is that although the rejecting class conceptually manifests an "IS-A" relationship, it is broken by the lack of conformance to the inherited interface. Since inherited behaviour is rejected, the subclass cannot be safely substituted in place of the supeclass, a fact which breaks Liskov's substitutability principle, thus resulting in a smell.

There are two paths which developers typically introduce this smell through. The first one is with a hierarchy which stems out of a concrete type and may display this behaviour in absence of proper refactoring. This will most likely occur in real-world projects as the design evolves. When developers are aware that they are rejecting superclass behaviour they should consider refactoring. The other cause provided is that developers may attempt to create what is called a "Swiss Army Type" that provides everything resulting in the subtypes having to reject it.

The suggested refactoring is the "Move Method" refactoring. If the behaviour being rejected is only relevant to some of the subclasses then it should be moved to the relevant classes. Also, if all the subclasses reject the method then one should apply the "Remove Method" refactoring on the superclass. It is also possible that adding an intermediate type is a possible solution. The advantage of this refactoring is that it will improve all quality attributes making the code more understandable, modifiable, extensible, reusable and reliable. The added reliability comes from the fact that behaviour will be consistent between subclasses and superclasses.

2.9.2 Broken Hierarchy

The broken hierarchy is a smell that arises when the class and its subclass do not share a conceptual "IS-A" relationship. There are three different forms described: a) The data and behaviour defined in the superclass are applicable to the subclass even though they do not share an "IS-A" relationship, b) There is no "IS-A" relationship but the subclass does not reject any irrelevant methods, c) The subclass rejects methods as in the Rebellious hierarchy smell.

The rationale behind this smell is that violation of an "IS-A" relationship in a hierarchy not only makes it harder to understand the structure of the hierarchy, it also leads to subclasses being incompatible with their superclasses. When this occurs, the result will be a violation of the principle of substitutability. This smell is in part a generalisation of the rebellious hierarchy smell.

The listed potential causes for this smell are two, a) Inheritance is applied with the intent of reuse but a conceptual "IS-A" relationship does not exist, or b) an inexperienced designer mistakenly inverts the inheritance relationship between types. The suggested refactoring is usually case dependant. Replacing inheritance with delegation, the composite design pattern, inverting the hierarchy or introducing a common supertype are all options, depending on the specific manifestation of the smell. These smells are challenging to detect because "IS-A" relationships are conceptual and difficult to examine with software.

As a practical consideration, it is suggested that sometimes use of the "Class Adapter" design pattern will result in the smell because the adapter inherits from the adpatee even though they do not share an "IS-A" relationship, in these cases it is suggested that an "Object Adapter" is used. This may not be practically possible if the adapter must override some behaviour from the adaptee. The designer should be aware of this issue when applying the pattern.

2.10 Avoid redundant paths

2.10.1 Multipath Hierarchy

The multipath hierarchy is when a type is inherited both directly and indirectly. This results in redundant paths in the hierarchy and in multiple inheritance languages such as C++ this can be caused by multiple inheritance. In single inheritance languages such as Java and C++, it is not possible since the inheritance structure is a tree and not a lattice. The smell, however, can still be present when a class inherits (and thus implements implicitly) an interface from a superclass but also (re)implements it explicitly. This can constitute a problem because it makes it hard to understand the relationship between classes. It can also be an indication that the developer that re-extended the class was not aware that the interface had been implemented already and may override methods that shouldn't be.

The potential cause for this smell is that designers have simply overlooked the inheritance path and it is noted that this is usually the result of a deep hierarchy. The suggested refactoring in these cases is to remove the redundant path. Introducing this smell on purpose may be necessary when the hierarchy is complex and it is important that a class implements a specific interface. This may help with clarity when designing a framework and is often done intentionally. The textbook provides examples from the java.io libraries that do this with the Serializable interface.

Detecting this smell in Java systems is straightforward, a software tool can be used to traverse up an inheritance tree and look for interfaces that are implemented twice.

2.11 Ensure proper ordering

2.11.1 Cyclic Hierarchy

The Cyclic (or Circular) Hierarchy Smell is present when a class is dependent on one of its subclasses in the way described earlier in this chapter. The superclass either contains one of its subclasses, refers to a name of one of its subclasses or accesses data members or methods defined in a subclass. The rationale provided for this smell is that hierarchical organisation is intended to simplify the relationship between classes, if classes depend on their subclasses then it makes the design harder to understand. The authors of the textbook contend that this smell breaks both the "Ensure Proper Ordering" enabling technique and the "Acyclic Dependencies Principle". The most severe case of this smell is when a class makes active use of a subclass which has not been initialised.

The potential causes for this smell are listed as the improper assignment of responsibilities across types, what this means is that the subclass has been assigned functionality that belongs to the superclass. The other reason is that such indirect dependencies it is difficult to be aware of all the dependency interrelationships between classes. As a practical consideration, the textbook contends that this smell may be acceptable if it is known that the subclasses which a class depends on will not change independently. This somehow justifies the existence of this smell pattern in the Java libraries.

Depending on the case there are several refactoring options provided. One can remove the reference to the subtype if it is unnecessary and if there is extensive coupling it may be possible that the types must be merged. Also, moving methods from the subtype to the supertype may resolve the need for the dependency. If none of these refactoring operations is suitable, it may be a viable solution to apply the state or strategy design patterns which can allow the type to change its behaviour at runtime.

Detecting this smell is simple in implementation, one can search through each class for its dependencies and check if they are one of its superclasses. The problem with this approach is that as the tree dependencies gets larger the search space gets disproportionately larger and it is increasingly computationally more expensive to resolve all dependencies in each level of depth. Although the Java libraries are prone to this smell it is reasonable to believe that this is a conscious choice by the developers as these are programming language libraries that have been used for more than 20 years.

This is the subset of the total amount of smells that have been proposed in the textbook that are related to inheritance hierarchies. Having established this background the next step is to introduce an appropriate research method including which smells and systems will be studied. This can be used to define the functionality of the tool to be developed and determine the specifics of the practical part of the study.

(INTENTIONALLY BLANK PAGE)

3 Research Method

This study of hierarchical smells will attempt an in-depth inspection of selected open source software systems. The objective is to not only calculate the frequency of a subset of hierarchical smells but also examine the systems manually with the intention of understanding the causes behind them and the utility of detecting them in practice. This is performed on a different level of analysis for each system because of the differences in size and domain of each system. Some smells may be caused by necessary attributes of the system such as a high number of classes or a design compromise such as the requirement for compatibility between different versions, there will be an attempt made to discover such characteristics. The custom tool which has been developed is used as a guide but manual inspection of the software systems where smell presence has been indicated will be performed so that the study can lead to useful conclusions. The cloned code detection tool (Copeland, 2005) is also employed when instances of wide hierarchies are found to determine if these are instances of the Unfactored Hierarchy smell.

The trade-off from human involvement, in contrast to a purely automatic approach, in this study is the number of systems that can be focused on. For this reason, the only systems used will be the 10 systems selected from the Qualitas Corpus (Tempero, 2010) recent release edition will be examined. Any systems that are not found to display instances of any smells when running the tool will be noted but not examined further and will be replaced in the study.

3.1 Qualitas Corpus

The Qualitas Corpus is a collection of software systems which is designed to be used for performing studies on software code. When performing such a study it is important that it can be reproduced so the use of a "curated" collection of systems is important. This was introduced with an article (Tempero *et al.*, 2010) that details the rationale behind its creation which is because previous versions of software studies did not list the specific systems or versions that they studied. There are several versions of the Qualitas Corpus and the one used for this study is the 20130901r release which is the most recent release and contains the 112 most recent versions of each system in the collection.

3.2 Selection Criteria

Several commonly studied software systems such as Eclipse, JHotDraw and the Java Class Libraries (JCL) provided with the Java Software Development Kit (SDK) were included in the study. The other 7 projects were selected from the Qualitas Corpus by considering their domain and then their size with the aim of studying a diversified set of systems with different numbers of classes. The final set of systems selected was (Table 3.1):

Name	Version	Domain	Number of Classes
JRE	1.6.21	programming language	10714
Eclipse SDK	4.3	IDE	33874
JHotDraw	7.5.1	3D/graphics/media	1070
Weka	3.7.9	tool	2390
Azureus (Vuze)	4.8.1.2	database	7680
FreeCol	0.10.7	games	1310
Marauroa	3.8.1	games	204
ArgoUML	0.34	diagram generator/data visualization	2560
FreeMind	0.9.0	diagram generator/data visualization	912
Apache Ant	1.8.4	parsers/generators/make	1290

Table 3.1

3.3 Detection Tool

The detection tool has been developed specifically for this task and details of the implementation and usage are provided in the following chapter. It can detect four of the hierarchical smell patterns which are the Multipath, Cyclic, Wide and Deep Hierarchy smells. The reason for selecting the specific patterns is that they can be objectively defined in a way that allows the computer to detect them by examining a tree representation of the project. This tool will be used on all the systems selected from the Qualitas Corpus to detect any instances of these smells. An extra function of this tool is to output a visual representation in text format of the software systems hierarchical tree to the standard output. Examining this manually for instances of a certain kind of Speculative Hierarchy will be performed by looking for branches of the inheritance tree where classes have only one child for more than two levels of inheritance depth.

3.4 Manual Examination

By printing the tree representation that has been constructed internally, the study intends to locate irregular patterns that can indicate any useful information that can be extracted by the system. An example of this can be found in the Java Class Libraries corba package in which a Deep Hierarchy appears to be the result of poor design. Also, the tool output is designed in such a way that it can be filtered with standard *nix command line tools such as grep and wc which can be used to filter results and find specific classes representing smells. For example, one can count not only the occurrences of the Multipath Hierarchy smell but also filter the results for the interface "Serializable", an interface common in this smell, to count only these instances.

3.5 PMD's Copy/Paste Detector

This software tool will be used on instances of Wide Hierarchies to determine if they are also instances of the Unfactored Hierarchy. This tool that uses the Karp-Rabin string matching algorithm (Karp and Rabin, 1987) to detect instances of identical code. It can be used for six different programming languages and can ignore the values of literals so that detection can be more complete.

3.6 Result Format

For each system, there is a standard format for the results. Each has a summary of numerical results for the four smells that the tool can detect. A short description which is compiled from information on the systems homepage and Wikipedia description. A short discussion of observations made for each smell and a paragraph of conclusions regarding the overall observations that were made while examining each system. This is followed by a discussion of each smell which summarises the findings and learnings about each of them and an evaluation of the utility of the tool and suggested improvements based on the observations made during its use. The final chapter suggests some ideas on how the smells could be more useful based on general conclusions, discusses what can be learned from them, what has been learned from the study and the proposals for future work that will be undertaken.

(INTENTIONALLY BLANK PAGE)

4 Detection Tool

Performing the study required the development of a static analysis software tool that can detect the selected instances of hierarchical smells. This tool depends on only one external framework/library which is the Eclipse Java Development Tools (JDT) which provide the necessary functionality to extract information from Java source files and can be found online (Foundation, 2017). The tool extracts the necessary information from source code and builds a data structure that represents the projects class hierarchy. It then analyses the tree that is built so that it can detect the smells. The specific steps for performing these operations are:

- Extract the name, package and parent class and field types from each public class in the project.
- Create separate nodes for each class to store the data.
- Link each class node to its superclass by comparing their names and providing a reference to it.
- Generate a list of direct subclasses for each node.
- Resolve the names of dependencies to class names.
- Remove all circular hierarchies* and classes orphaned as a result
- Detect each selected smell in order.

*In this case, the circular hierarchies referred to are not the cyclic hierarchy smell but are instances where two or more classes extend (←) each other. This could be three classes A, B, C where A←B, B←C, C←A. This situation would cause a stack overflow when searching and is not permitted by the Java compiler, for this reason these classes and all their subclasses are removed.

Some details about each step are presented below.

4.1 Extracting information from Java source files

Extracting information from the Java source file is performed using the Eclipse JDT framework which is offered as an open source framework with the Eclipse IDE and is intended for creating plugins. The JDT libraries can also be used independently from the IDE to create standalone applications. Its approach to accessing information in code is to parse each Compilation Unit, which in the case of Java is a source file, into an Abstract Syntax Tree (AST) representation. The AST is a data structure which represents code in a similar way to a parse tree. The difference with the AST is that it does not all retain all the details (such as parentheses) available in the code and focuses on retaining semantic correctness. Each node of the AST can then be visited by implementing a custom version of the ASTVisitor (Eclipse, 2013) that can access the information available.

4.2 Internal Representation of the Hierarchy Tree

The same concrete visitor is used to access every Compilation Unit and returns with a list of "ClazzNode" objects which have been stored internally. Each of these objects contains the information retrieved from a single Compilation Unit. This information is the name of the class and the class they extend, the names of interfaces they implement and the names of all types and variables they access. This information is then used to link the "ClazzNodes" with each other. The result is that each object of this type will have a reference to the object representing its supertype but also a list of the objects that extend it. There is a third list of the classes it depends on which is constructed by searching the list of "ClazzNodes" and matching them to any of the types referenced in the body of the class. Internal classes are not represented in this structure.

A similar operation is also performed for the classes from the JDK libraries and each class in the project is matched against these also. The outcome of these a tree structure which represents the projects type hierarchy and can be traversed recursively just as any other tree. At the top of this tree is the class "Object" which is constructed by default and is not parsed from the JDK libraries.

An alternative approach to creating a single tree that represents the whole project would be to only resolve local classes and study the resulting hierarchies separately without creating a unified tree. This may be a better approach for studying width and depth of hierarchies but would not allow for fully studying the multipath hierarchy smell because many of these originate from subclasses of Java Class Library types that re-implement interfaces that have been inherited.

4.3 Detecting Hierarchical Smells

4.3.1 Deep Hierarchy

This smell is detected by recursively retrieving all the superclasses of each type in the hierarchy tree and counting them. This produces a value for the DIT metric which is then compared against the default threshold of 6 superclasses which was selected based on the textbook recommendation but can also be set to a different value with the –dt option. If the DIT metric for a class exceeds the threshold then the hierarchy is deemed deep and the smell is detected. If multiple classes with the same parent class are found to be deep in the hierarchy they will be counted as separate instances of the smell.

4.3.2 Wide Hierarchy

The wide hierarchy smell is detected by calculating the NOC metric for each class and comparing it to the threshold for this metric which by default is 9 subclasses. This threshold was also selected because of the suggestion in the textbook. The default threshold can also be set to a different value with the –wt <value> option. Since each class has a list of references to its subclasses, it is sufficient to use the size of this list to calculate the NOC value.

4.3.3 Multipath Hierarchy

In Java, this smell happens only with multiple implementation of interfaces rather than inheritance because of single inheritance. The smell is detected by iterating upwards in the tree starting from each class and comparing its implemented interfaces to those of each superclass. If any of the interfaces match then this instance of the multipath hierarchy smell detected and output. Multipath Hierarchy detection only looks for re-implementation of the interfaces by project types and ignores those that already exist in the classes that Java provides. These will be studied separately.

4.3.4 Cyclic Hierarchy

This smell is detected by checking each of the dependencies to see if it is derived from the class that is being checked. This is done recursively for dependencies of dependencies to a distance of 5 classes by default. This threshold was determined empirically based on computation times, but can also be set with the –ct option. This depth was selected because the growth of the search space makes the task very computationally expensive after it. For this depth, when a dependency is found to be a subclass (even if it is not immediate) the notification is output to the console.

4.4 Use of the tool

4.4.1 Usage

The tool is developed in Java and provides a Command Line Interface (CLI). The user must provide the path to the Jar or Directory containing the project they want to analyse as a command line argument, otherwise, the working directory is analysed. To include the Java Class Libraries the option j should be used, this is always advised unless it is the Java Class Libraries themselves which are being analysed. The JDK library used to resolve dependencies is version 8.0 update 131 because it was the recent version when the tool was created. By using the options in Figure 4.1 the user can determine which smells are detected and the output of the software.

Options:

-p	Output the hierarchy tree to the console.
-j	Include Java 1.8.131 classes in the final tree.
-d	Detect the Deep Hierarchy smell.
-w	Detect the Wide Hierarchy smell.
-m	Detect the Multipath Hierarchy smell.
-c	Detect the Cyclic Hierarchy smell.
-dt <value>	Specify a threshold for the Deep Hierarchy smell.
-wt <value>	Specify a threshold for the Wide Hierarchy smell.
-ct <value>	Specify a distance threshold for the Cyclic Hierarchy smell.

Figure 4.1

4.4.2 Examples

A sample usage of this command line tool is the one used to perform the study of JHotDraw 7.5.1 from the projects source folder (Figure 4.2):

```
$ java -jar ~/smtool.jar -pwdmc -j
```

Figure 4.2

To count instances of the multipath hierarchy in JDK 1.6.21 the command used from the root directory of the project was (Figure 4.3):

```
$ java -jar ~/smtool.jar -m | grep HIERARCHY | wc -l
```

Figure 4.3

These examples result in all the smells detected using the default thresholds. The threshold is the largest value for a metric that is not considered a smell. The default search distance for the cyclical hierarchy smell was set to 5 by default, the value is tied to the computational complexity of the task and was selected because of some of the systems in the study being very complex to analyse.

5 Results and Analysis

This chapter presents the findings which resulted from performing analysis with the tools described and attempts to provide insight into the reasons behind the existence of smells in each system. The intent is to discover themes and pinpoint any design artefacts which diverge from common practices or reasonable use of inheritance as defined by the guidelines and enabling techniques. Results are presented as a discussion of each individual system alongside some output extracts from the tool to aid understanding of the hierarchies being discussed. The overall implications of any findings and evaluation of the tool based on those are presented in a separate discussion chapter which follows.

For the following ten systems analysed an instance of a smell is discovered when:

- **Multipath Hierarchy,** a class implements an interface that is inherited from a superclass.
- **Cyclic Hierarchy,** a class (originating class) depends on a subclass by referencing, through composition, or by accessing its methods or data. The search depth, meaning the distance of classes searched from the originating class, for this study is 5.
- **Deep Hierarchy,** a class has more than 6 parent classes.
- **Wide Hierarchy,** a class has more than 9 direct subclasses.

For a hierarchy to display one of these smells one instance of it is required. The approach followed in this study is to construct a single hierarchy tree of all the types in each system starting from "Object" and by using the Java 8 classes to complete this tree. This means that each smell may be detected multiple times, this is what is being called an "instance" of a smell. This will also mean that if a project has 3 classes with more than 6 superclasses they will be detected separately even if they extend the same class. This was done so that the specifics of each smell can be studied in greater detail.

5.1 Java Runtime Environment 1.6.21 Class Libraries

5.1.1 Description

The Java Class Libraries (JCL) are provided with the Java Development Kit (JDK) and are the source code for the base framework provided with the Java programming language. In the Qualitas Corpus this is referred to as JRE, which should be the HotSpot runtime and is C++ proprietary software. Because of this and the fact that the packages listed in the Qualitas Corpus are those from the JCL, this is what is being studied. This framework provides classes and interfaces that can be used to develop Java software with useful functionality such as math, SQL, networking, GUI, images and more. This is the first system to be analysed and has been selected as it is the collection of core Java classes used in most systems. The tool automatically excludes classes if their parent class cannot be found and for JDK, 40 of these were removed mostly in the corba and swing packages. These were confirmed by checking the contents of the file. For example class "java.awt.GraphicsCallback" was removed although it should extend "sun.awt.SunGraphicsCallback", a class (and package, "sun.awt") which could not be found by searching the Class Libraries even though it is also listed in the package list of the Qualitas Corpus.

5.1.2 Summary

Multipath Hierarchy: 119

Cyclic Hierarchy: 449220

Wide Hierarchy: 53

Deep Hierarchy: 25

5.1.3 Findings

Multipath Hierarchy

In the 119 instances of this smell detected, the most common interfaces being implemented at different levels of the hierarchy are 49 instances of the "Serializable", 12 of "Accessible", 9 of "Cloneable" and 13 of "DirectBuffer". Although this has been defined as a smell the textbook contends that this may be performed at multiple levels of the hierarchy for reasons of clarity and this appears to be a deliberate choice in the case of the JCL since the framework is designed for reuse and it is highly unlikely that this smell occurs in the class libraries of a programming language resulting as an error.

Cyclic Hierarchy

There are 449220 instances of this smell detected. The tool detects dependency on the same subclass through different paths independently. It was not possible to independently confirm these Cyclic Hierarchy smells manually so a random selection of 15 instances was performed all which were confirmed. The Cyclic Hierarchy smell is particularly a problem when the dependency is through a class's constructor or when a class is intended to be replaced. This was not observed in the instances that were checked and it is expected that the JCL classes would be modified simultaneously, instead of types being changed without the others being considered, so the design may be justifiable. Adding new types, however, may be more complicated than in a loosely coupled system but is possible in such cases where a single developer, Sun or Oracle, has control over all the sources in a system and the types are expected to be stable.

A selected example of this smell can be seen in the following hierarchy tree extract from the "java.nio" package (Figure 5.1):

```
Package: java.nio
Object
|___Buffer                    Interfaces: [] --- Package: java.nio
| |___ByteBuffer              Interfaces: [] --- Package: java.nio
| | |___MappedByteBuffer      Interfaces: [] --- Package: java.nio
| | | |___DirectByteBuffer    Interfaces: [DirectBuffer] --- Package: java.nio
```

Figure 5.1

The smell in this hierarchy is the following method implementation (Figure 5.2) where the "ByteBuffer" is constructing one of its subclasses "DirectByteBuffer" and was subsequently detected by the tool:

```
public static ByteBuffer allocateDirect(int capacity) {
        return new DirectByteBuffer(capacity);
    }
```

Figure 5.2

Wide Hierarchy

There are 53 instances of the Wide Hierarchy smell in this system. These are concentrated in specific packages such as "corba", "swing", "util" and "io". Commonly, these are abstract classes with many situation-specific subclasses such as events, exceptions and UI elements. The class with the most direct subclasses (105) is "java.lang.Exception" which is the class which all exceptions extend, "javax.swing.JComponent" which is the base class for most swing elements has 32 subclasses and "java.util.Event" which is the root class for all event state objects with 34. Most other instances of this smell were wide hierarchies for similar classes. Although wide hierarchies are not desirable, they seem to be unavoidable especially in specific subdomains of an extensive framework such as the JCL of which usually only a subset is used in projects.

Deep Hierarchy

There are 25 instances of the deep hierarchy counted in the Java Class Libraries this is a small number compared to the total number of classes (10714 according to the Qualitas Corpus but the sun package is missing as noted above) in the libraries. Different occurrences of this smell under the same parent class are counted separately in this study. These are mostly in the "corba" and "apache" packages. Seven of the instances are subclasses of the "LocPathIterator" in the "apache" package define iterators for different situations.

In the "corba" package, there is a deep hierarchy that stands out and can be seen in the hierarchy extract below (Figure 5.3).

```
Object                         Package: java.lang
|___ORB                        Package: org.omg.CORBA
| |___ORB                      Package: org.omg.CORBA_2_3
| | |___ORB                    Package: com.sun.corba.se.org.omg.CORBA
| | | |___ORB                  Package: com.sun.corba.se.spi.orb
| | | | |___ORBImpl            Package: com.sun.corba.se.impl.orb
| | | | | |___ORB              Package: com.sun.corba.se.internal.iiop
| | | | | | |___ORBSingleton   Package: com.sun.corba.se.impl.orb
| | | | | | | |___ORBSingleton Package: com.sun.corba.se.internal.corba
| | | | | | |___POAORB        Package: com.sun.corba.se.internal.POA
| | | | | | | |___PIORB       Package: com.sun.corba.se.internal.Interceptors
```

Figure 5.3

Some of these classes simply import from a different package and extend without changing the implementation while others provide multiple lines of code that affects the class. This kind of a hierarchy with names of the same type can cause confusion unless a developer has knowledge of the specifics of the hierarchy and ought to be refactored for simplicity. It is possible that this hierarchy developed over multiple versions of the "corba" package and represents a design compromise that stems from the initial design being iterated upon but an attempt to retain backwards compatibility.

5.1.4 Conclusions

The Java Class Libraries is a large framework which displays a high count of instances of each smell. The Cyclic Hierarchy smell was not explored to the full depth and would have likely not have been possible to compute within the time constraints of this work. It will be interesting to see how common this smell is in systems that will be analysed onwards as the dependency of classes on subclasses should be avoided. The wide hierarchies are limited to specific domains in which it makes sense that there are many subtypes as there may be many Actions, UI elements or Throwable types which do not benefit from classification but are required for diversification. There are not many deep hierarchies in the JCL but the example from the "corba" package shows how one can be created in a confusing and unhealthy manner.

5.2 Eclipse SDK 4.3

5.2.1 Description

Eclipse is an Integrated Development Environment (IDE) that provides support for many languages but is used primarily for Java development. It uses its own GUI framework called SWT instead of the AWT or Swing which are provided with Java. It is developed with a plugin architecture which allows developers to extend the environment with custom features such as static analysers or profilers. It was selected for its importance and size.

An important finding in Eclipse is that it displayed an instance of circular inheritance where three classes extend each other. The classes involved in it were defined as follows (Figure 5.4):

```
public abstract class ModelParticipantMergeOperation extends ModelMergeOperation
public abstract class AbstractModelMergeOperation extends ModelParticipantMergeOperation
public class ModelMergeOperation extends AbstractModelMergeOperation
```

Figure 5.4

Searching through this hierarchy resulted in a "StackOverflowError" for the software as it recursively iterated through these three classes until stack space was depleted. As a result, the tool was modified to remove and report such classes and their subclasses from the hierarchy before analysis as such hierarchies are not allowed in Java even if the classes are abstract.

5.2.2 Summary

Multipath Hierarchy: 172

Cyclic Hierarchy: 1483205

Wide Hierarchy: 193

Deep Hierarchy: 361

5.2.3 Findings

Multipath Hierarchy

There is a sum of 172 instances of the Multipath Hierarchy smell detected in Eclipse SDK. The most common interfaces in this smell were 48 instances for "Cloneable", 13 for "Serializable", 27 for various kinds of listeners. This is quite consistent with the findings in other systems with "Serializable" being the most common interface involved in this smell.

Cyclic Hierarchy

Eclipse is the biggest system in this study and has the most counts of this smell. Although it is built with plug-in extensions in mind it has a high count of java source files (22647) in the base system. There is also a high amount of interdependency between types. As with the JCL source files, it is impossible to analyse all the instances of this smell in the system but the presence of this was confirmed by random selection of results. This smell is often indicative of a monolithic design within different components even if the overall design is modular.

A sample example of this smell can be seen in the tool output below (Figure 5.5):

```
Package: com.ibm.icu.util

CYCLIC HIERARCHY:  -> Calendar -> GregorianCalendar
CYCLIC HIERARCHY:  -> Calendar -> JapaneseCalendar
CYCLIC HIERARCHY:  -> Calendar -> BuddhistCalendar
CYCLIC HIERARCHY:  -> Calendar -> TaiwanCalendar
CYCLIC HIERARCHY:  -> Calendar -> PersianCalendar
CYCLIC HIERARCHY:  -> Calendar -> IslamicCalendar
CYCLIC HIERARCHY:  -> Calendar -> HebrewCalendar
CYCLIC HIERARCHY:  -> Calendar -> ChineseCalendar
CYCLIC HIERARCHY:  -> Calendar -> IndianCalendar
CYCLIC HIERARCHY:  -> Calendar -> CopticCalendar
CYCLIC HIERARCHY:  -> Calendar -> EthiopicCalendar
CYCLIC HIERARCHY:  -> Calendar -> DangiCalendar
```

Figure 5.5

From Figure 5.5 we can observe that the abstract class "Calendar" depends on its concretions. An example of this in code is the following code from Calendar.java (Figure 5.6):

```
static Calendar createInstance(ULocale locale) {
        Calendar cal = null;
        TimeZone zone = TimeZone.getDefault();
        int calType = getCalendarTypeForLocale(locale);
        if (calType == CALTYPE_UNKNOWN) {
            // fallback to Gregorian
            calType = CALTYPE_GREGORIAN;
        }

        switch (calType) {
        case CALTYPE_GREGORIAN:
            cal = new GregorianCalendar(zone, locale);
            break;
        case CALTYPE_JAPANESE:
            cal = new JapaneseCalendar(zone, locale);
            break;
        case CALTYPE_BUDDHIST:
            cal = new BuddhistCalendar(zone, locale);
            break;
        case CALTYPE_ROC:
            cal = new TaiwanCalendar(zone, locale);
            break;
        case CALTYPE_PERSIAN:
            cal = new PersianCalendar(zone, locale);
            break;
        case CALTYPE_ISLAMIC_CIVIL:
            cal = new IslamicCalendar(zone, locale);
            break;
        case CALTYPE_ISLAMIC:
            cal = new IslamicCalendar(zone, locale);
            ((IslamicCalendar)cal).setCivil(false);
            break;
        case CALTYPE_HEBREW:
            cal = new HebrewCalendar(zone, locale);
            break;
        case CALTYPE_CHINESE:
            cal = new ChineseCalendar(zone, locale);
            break;
        case CALTYPE_INDIAN:
            cal = new IndianCalendar(zone, locale);
            break;
        case CALTYPE_COPTIC:
            cal = new CopticCalendar(zone, locale);
            break;
        case CALTYPE_ETHIOPIC:
            cal = new EthiopicCalendar(zone, locale);
            break;
        case CALTYPE_ETHIOPIC_AMETE_ALEM:
            cal = new EthiopicCalendar(zone, locale);
            ((EthiopicCalendar)cal).setAmeteAlemEra(true);
            break;
        case CALTYPE_DANGI:
            cal = new DangiCalendar(zone, locale);
            break;
        case CALTYPE_ISO8601:
            // Only differs week numbering rule from Gregorian
            cal = new GregorianCalendar(zone, locale);
            cal.setFirstDayOfWeek(MONDAY);
            cal.setMinimalDaysInFirstWeek(4);
            break;
        default:
            // we must not get here, because unknown type is mapped to
            // Gregorian at the beginning of this method.
            throw new IllegalArgumentException("Unknown calendar type");
        }

        return cal;
    }
```

Figure 5.6

An interesting thing to note about this part of the hierarchy is that although it suffers from the cyclic hierarchy smell, appropriate classification has been applied to avoid a wide or multipath hierarchy as seen below (Figure 5.7):

```
Object
|___Calendar                 Interfaces: [Serializable, Cloneable, Comparable]
| |___CECalendar             Interfaces: [] --- Package: com.ibm.icu.util
| | |___CopticCalendar       Interfaces: [] --- Package: com.ibm.icu.util
| | |___EthiopicCalendar     Interfaces: [] --- Package: com.ibm.icu.util
| |___ChineseCalendar        Interfaces: [] --- Package: com.ibm.icu.util
| | |___DangiCalendar        Interfaces: [] --- Package: com.ibm.icu.util
| |___GregorianCalendar      Interfaces: [] --- Package: com.ibm.icu.util
| | |___BuddhistCalendar     Interfaces: [] --- Package: com.ibm.icu.util
| | |___JapaneseCalendar     Interfaces: [] --- Package: com.ibm.icu.util
| | |___TaiwanCalendar       Interfaces: [] --- Package: com.ibm.icu.util
| |___HebrewCalendar         Interfaces: [] --- Package: com.ibm.icu.util
| |___IndianCalendar         Interfaces: [] --- Package: com.ibm.icu.util
| |___IslamicCalendar        Interfaces: [] --- Package: com.ibm.icu.util
| |___PersianCalendar        Interfaces: [] --- Package: com.ibm.icu.util
```

Figure 5.7

Wide Hierarchy

The Eclipse SDK has 193 instances of the Wide Hierarchy smell. Of these, 19 have a width of greater than 50 subclasses. It is evident that more classes exist in wide hierarchies than deep hierarchies because elements of the user interface are built with different classes that are created by extending existing types. These are mainly Dialogs, Handlers, Actions and other elements such as Wizards. This is consistent with the size of the system and the complexity of a large integrated application such as an IDE and for such a project may be unavoidable.

Deep Hierarchy

Eclipse has 361 classes at a hierarchy depth larger than 6. This is a large count compared to what is seen in other systems but is small compared to the total number of classes in the system. What is observed is that this is not a result of single hierarchy chains which would appear particularly problematic but the outcome of multiple levels of classification, mostly of classes that handle different kinds of Actions. Filtering the tree results (total number of classes parsed 19042) for class names that contained "Action" revealed 1360 results while filtering the instances of Deep Hierarchy for the string "Action" revealed that 115 out of 361 instances of the smell were of a class that contained the string. This was calculated with the following (Figure 5.8) use of the tool and *NIX commands:

```
$ java -jar smtool.jar -jd | grep HIERARCHY | grep Action | wc -l
```

Figure 5.8

Similarly, there were 49 classes containing the string "Dialog" which is the second most common in this smell. There was no indication found in this system that any of these hierarchies are problematic or caused by some other smell such as speculative hierarchy.

5.2.4 Conclusions

Eclipse SDK is the second largest system in the Qualitas Corpus and displays instances of all the smells being detected by the tool. The nature of interfaces and classes involved in this smell is consistent with findings in other systems. The "Serializable" and "Cloneable" interfaces that are commonly implemented through multiple paths, a pattern commonly detected in other systems analysed. Deep and wide Hierarchies were observed but their domain was also like that of other systems, particularly wide hierarchies of Actions and UI elements. The more concerning fact is the amount of Cyclic Hierarchy smells that were detected and may cause problems with altering the functionality or structure of the program.

5.3 JHotDraw 7.5.1

5.3.1 Description

The version of JHotDraw 7.5.1 is the most recent version available in the Qualitas Corpus. This is a Java GUI framework for creating technical and structured graphics. It was built as a design exercise by Erich Gamma and Thomas Eggenschwiler. For this reason, it is a very commonly studied system.

5.3.2 Summary

Multipath Hierarchy: 4

Cyclic Hierarchy: 2

Wide Hierarchy: 13

Deep Hierarchy: 18

5.3.3 Findings

Multipath Hierarchy

JHotDraw displays only four instances of this smell which allows for a case by case analysis. The abstract class “ODGAttributedFigure” implements the interface “ODGFigure” which only defines one method: isEmpty(), but does not provide a concrete definition. This is separately implemented by both its subclasses “ODGRectFigure” and “ODGEllipseFigure”. The implementation in each subclass is different so the method cannot be “Pushed Up”. It can be concluded that this was done on purpose for reasons of clarity because the classes only differ by one level of inheritance and the method is not overridden at any point. These classes were parsed from a sample application ODG.

This same pattern is seen with the class “AbstractAttributedDecoratedFigure” which implements the interface “DOMStorable” when this interface is also inherited from “AbstractAttributedFigure”. In this case, the methods read() and write() are implemented concretely in the subclass by calling the implementation of the superclass then calling a method that adds the extra behaviour. In this case, since both classes provide implementations for the methods which reference each other hierarchically, it is apparent that this is a result of deliberate design and not mistaken overriding. Finally, there is one instance of “Serializable” which is implemented by both “BezierPath” and its parent class “ArrayList”.

Cyclic Hierarchy

One instance of this smell is detected through three different paths. This is the dependence of “ColorFormatter” on its subclass “ColorToolTipTextFormatter”. The shortest path for this is through the “ColorUtil” class which provides a static method “toColor” and does not operate on the subclass. Although in this instance the “ColorFormatter” depends on a class that knows of its subclass, it does not operate on it in a way that could result in a foreseeable issue (such as participating in constructor calls). This appears to be deliberately designed and probably does not pose a problem.

Deep Hierarchy

"LabeledLineConnectionFigure" and "DependencyFigure" are two instances of the deep hierarchy smell at 7 levels of inheritance. These extend the "LineConnectionFigure" which extends the "LineFigure" type. It is interesting to note that the "LineFigure" intermediate type is a concrete type which is extended only by "LineConnectionFigure" which overrides all its behaviour. Because of this, it is possibly not a suitable intermediate type.

The other instance is the wide hierarchy of toolbar types that exists at significant depth of 7 and is built up from the toolbar types provided in the samples package. The depth of this hierarchy is primarily due to inheriting the swing type "JToolBar". This is a wide hierarchy of 11 bottom level concrete types. The degree of code duplication in these types is low and providing different classes of toolbar concretions is reasonably sound. The hierarchy extract for can be seen below (Figure 5.9):

```
Object
|___Component                         Package: java.awt
| |___Container                       Package: java.awt
| | |___JComponent                    Package: javax.swing
| | | |___JToolBar        I           Package: javax.swing
| | | | |___JDisclosureToolBar        Package: org.jhotdraw.gui
| | | | | |___AbstractToolBar         Package: org.jhotdraw.samples.svg.gui
| | | | | | |___ActionsToolBar        Package: org.jhotdraw.samples.svg.gui
| | | | | | |___AlignToolBar          Package: org.jhotdraw.samples.svg.gui
| | | | | | |___ArrangeToolBar        Package: org.jhotdraw.samples.svg.gui
| | | | | | |___CanvasToolBar         Package: org.jhotdraw.samples.svg.gui
| | | | | | |___FigureToolBar         Package: org.jhotdraw.samples.svg.gui
| | | | | | |___FillToolBar           Package: org.jhotdraw.samples.svg.gui
| | | | | | |___FontToolBar           Package: org.jhotdraw.samples.svg.gui
| | | | | | |___LinkToolBar           Package: org.jhotdraw.samples.svg.gui
| | | | | | |___StrokeToolBar         Package: org.jhotdraw.samples.svg.gui
| | | | | | |___ToolsToolBar          Package: org.jhotdraw.samples.svg.gui
| | | | | | |___ViewToolBar           Package: org.jhotdraw.samples.svg.gui
| | | | |___SummaryToolBar            Package: org.jhotdraw.samples.svg.gui
```

Figure 5.9

Wide Hierarchy

There are several wide branches of the hierarchy tree in JHotDraw. These are the subclasses of "AbstractTool" which provides a tree structure underneath it, "AbstractHandle" which has subclasses for different types of Handle, "AbstractSelectedAction" which provides classes which handle different Actions and others. The wide hierarchy of this system appears to stem from the requirements of the JHotDraw application to provide functionality for differing types of Shapes and Tools. For this reason, it is perhaps necessary that these wide hierarchies exist because of the nature of the application which is to offer different tools for creating, linking and manipulating objects.

5.3.4 Conclusions

JHotDraw is a system that was created as a design exercise and appears to have a good balance of inheritance width and depth. When Wide and Deep hierarchy smells are detected they are a reasonable result of the nature of the system, as GUI elements (primarily in samples) will sometimes result in wide hierarchies because of their need to implement different graphical interface elements. The appearance of Cyclic and Multipath Hierarchy smells is also very low compared to other systems analysed which shows that although the system was designed before the smells where formally defined, application of design patterns and adherence to object-oriented guidelines results in a sound design with low need for refactoring.

5.4 Weka 3.7.9

5.4.1 Description

Weka 3.7.9 is the most recent version of Weka in the Qualitas Corpus. It is a Java software tool that implements machine learning algorithms for data mining either through a GUI or called directly from code as a library.

5.4.2 Summary

Multipath Hierarchy: 125

Cyclic Hierarchy: 305

Wide Hierarchy: 7

Deep Hierarchy: 3

5.4.3 Findings

Multipath Hierarchy

The tool detects 125 instances of the Multipath Hierarchy smell in Weka. These can be categorised by the interface that is being inherited multiple times. These interfaces are "Serializable" (87), "OptionHandler" (27), "RevisionHandler" (5), "TechnicalInformationHandler" (2), "CapabilitiesHandler"(1), "DataSourceListener" (2), "FileSourcedConverter" (1).

The "Serializable" interface is inherited in most of these cases from a Swing superclass. Serializable is a Java library interface that very commonly implemented at many levels of a hierarchy most likely for clarity, this has been observed across multiple systems. The "OptionHandler" is the other interface which is involved in many counts of the multipath hierarchy smell in this system. This is an interface that has been defined in the Weka system and provides an interface to classes which "understand options". These are mostly subclasses of "AbstactClassifier". It appears that the use of the multipath hierarchy smell is deliberate in Weka although in most cases it could have been avoided because of the shallow depth of the classes that implement it.

Cyclic Hierarchy

This smell appears to be very common in the Weka software. For a depth of dependency of 5, there were 305 instances of the smell detected. The existence of this many instances smell is very likely to complicate reuse of the classes in the system as well as the tasks of maintaining and extending the software program. This is due to the high inter-dependence of the classes in the system. It is however not fully explored and does not appear to affect constructor calls. An example of these can be seen from the selected output below (Figure 5.10):

```
Package: weka.core.pmml
CYCLIC HIERARCHY:  -> Expression -> Constant
CYCLIC HIERARCHY:  -> Expression -> FieldRef
CYCLIC HIERARCHY:  -> Expression -> Apply
CYCLIC HIERARCHY:  -> Expression -> NormDiscrete
CYCLIC HIERARCHY:  -> Expression -> NormContinuous
```

Figure 5.10

By examining the code of Expression.java it is found that this was correctly detected in the following method (Figure 5.11):

```
public static Expression getExpression(String name,
                                   Node expression,
                                   FieldMetaInfo.Optype opType,
                                   ArrayList<Attribute> fieldDefs,
                                   TransformationDictionary transDict)
                                   throws Exception {
    Expression result = null;
    if (name.equals("Constant")) {
      // construct a Constant expression
      result = new Constant((Element)expression, opType, fieldDefs);
    } else if (name.equals("FieldRef")) {
      // construct a FieldRef expression
      result = new FieldRef((Element)expression, opType, fieldDefs);
    } else if (name.equals("Apply")) {
      // construct an Apply expression
      result = new Apply((Element)expression, opType, fieldDefs, transDict);
    } else if (name.equals("NormDiscrete")) {
      result = new NormDiscrete((Element)expression, opType, fieldDefs);
    } else if (name.equals("NormContinuous")) {
      result = new NormContinuous((Element)expression, opType, fieldDefs);
    } else if (name.equals("Discretize")) {
      result = new Discretize((Element)expression, opType, fieldDefs);
    } else if (name.equals("MapValues") ||
        name.equals("Aggregate")) {
      throw new Exception("[Expression] Unhandled Expression type " + name);
    }
    return result;
}
```

Figure 5.11

The following extracted hierarchy tree (Figure 5.12) which has been confirmed by examining the files manually shows how these are indeed subclasses of Expression. This is also implied by the return type of the method getExpression above.

```
Object
|___Expression        Interfaces: [Serializable] --- Package: weka.core.pmml
| |___Apply           Interfaces: [] --- Package: weka.core.pmml
| |___Constant        Interfaces: [] --- Package: weka.core.pmml
| |___FieldRef        Interfaces: [] --- Package: weka.core.pmml
| |___NormContinuous  Interfaces: [] --- Package: weka.core.pmml
| |___NormDiscrete    Interfaces: [] --- Package: weka.core.pmml
```

Figure 5.12

Wide Hierarchy

This smell exists in various parts of the Weka design. Some of the classes with a high count of subclasses are Java Swing types such as “JDialog” and “JPanel”. “JPanel” specifically has 113 direct subclasses which implement different kinds of classes, inspecting them shows that this was a design choice made so that different kinds of panels can be added to the interface. This is an instance where inheritance has been preferred over object composition for creating a swing interface. “AbstractClassifier” with 35 subclasses is another class which produces this smell, this is interesting to note because it is the originating class in the taxonomy for 72 (of 305) of the Cyclic Hierarchy smells and 29 of the multipath hierarchy instances. This abstract type appears to be tightly tied into the rest of the system.

Deep Hierarchy

There are only three classes in this test which are deeper than six levels of inheritance. These are all subclasses of JPanel and this is the reason they are deep in the hierarchy. Generally, the Weka system does not suffer from the deep hierarchy smell.

5.4.4 Conclusion

Weka is a software system that has been built with a monolithic design. Although the machine learning functionality has been separated from the user interface it still displays a high number of interdependent classes and several very wide hierarchies. These characteristics lead to the conclusion that it has not been built with attention to reuse of classes but rather with the intent of integrating the functionality.

5.5 Azureus 4.8.1.2

5.5.1 Description

Azureus 4.8.1.2, now known as Vuze, is the most recent version of Azureus in the Qualitas Corpus. Azureus/Vuze is a BitTorrent client implemented in Java using the Standard Widget Toolkit (SWT) which is the same "heavyweight" GUI toolkit developed for the Eclipse project.

5.5.2 Summary

Multipath Hierarchy: 15

Cyclic Hierarchy: 5799

Wide Hierarchy: 14

Deep Hierarchy: 0

5.5.3 Findings

Multipath Hierarchy

This system appears to have 15 instances of the Multipath Hierarchy smell. Multiple implementation of interfaces is not localised to specific parts of the system. This appears to have been done on very shallow levels of inheritance and is done even when the interface defined methods are not overridden.

Cyclic Hierarchy

This system has a high occurrence of the Cyclic Hierarchy smell. The tool detects 5799 instances of it even though the threshold was reduced to 5. This was done because the values of 6 and 7 had very high execution times and was one of the systems that affected the selection of a lower search depth for the rest of the study. In this system, it is apparent that dependency of various classes on their subclasses is very common which is an indicator of the high interdependence of classes even if the hierarchy is relatively shallow. An example of this are the "IConsoleCommand", "OptionsConsoleCommand" and TorrentCommand" classes that depend on 21, 4, 10 of their subclasses respectively through the "ConsoleInput" class which it knows about due to the following method definition.

```
public abstract void execute(String commandName, ConsoleInput console, List<String> arguments);
```

Figure 5.13

This is a smell because it provides a method signature that subclasses must implement. But one of the arguments (ConsoleInput console) that specifically "knows" about these subclasses and registers them to the console.

Wide Hierarchy

Azureus has 14 classes with greater than 9 subclasses. This is a result of the fact that it is a system with a very low depth of inheritance which implies low application of type classification. Many of these wide hierarchies are a direct result of creating separate classes to handle different datatypes. An indicator of the low inheritance use in the design is that there are 2462 subclasses for the class Object, meaning that they do not have an explicit parent type in the system.

Deep Hierarchy

Azureus does not have instances of the Deep Hierarchy smell, meaning that it does not have inheritance trees deeper than 6 levels. This is a side effect of the low use of inheritance in the system.

5.5.4 Conclusions

Azureus is a software system that makes very low use of deep inheritance trees and two observations can be made about it. The first is that wide hierarchies are between one and three levels deep and many classes do not have an explicit superclass so the inheritance tree is very shallow. The second observation that can be made about the system is that even though the inheritance tree is shallow there is still a high degree of dependence of classes on their subclasses as observed from the cyclic hierarchies.

5.6 FreeCol 0.10.7

5.6.1 Description

FreeCol 0.10.7 is an open source clone of the video game Sid Meier's Colonisation. This is a strategy game and was selected to introduce more variety to the study because most of the systems used are programming tools.

5.6.2 Summary

Multipath Hierarchy: 26

Cyclic Hierarchy: 81283

Wide Hierarchy: 15

Deep Hierarchy: 69

5.6.3 Findings

Multipath Hierarchy

The FreeCol system has 26 instances of the multipath hierarchy smell. Of these, 22 instances are children of “FreeColPanel” which implement “ActionListener” which is also implemented by their parent Class. In this hierarchy, “FreeColPanel” also has subclasses that implement different interfaces which are also subtypes of the “EventListener” interface (which is a superclass of “ActionListener”). This leads to complexity in the hierarchy because classes such as “FindSettlementDialog” implement two kinds of listeners while they only should be implementing one, with the other being implicitly rejected.

Cyclic Hierarchy

FreeCol displays an abnormally high count of the cyclic hierarchy smell (84447). This is caused by the design of the system and is a direct result of tight coupling between types in this system. This design is very integrated and makes removing or adding types a complicated procedure. Resolving these issues would require a complete redesign of the system. A search even at a depth of one class distance reveals the following example (Figure 5.14):

```
Package: net.sf.freecol.client.gui.option
Object
|___OptionUI                       Interfaces: [OptionUpdater]
| |___AbstractUnitOptionUI         Interfaces: [ItemListener
| |___AudioMixerOptionUI           Interfaces: []
| |___BooleanOptionUI              Interfaces: []
| |___FileOptionUI                 Interfaces: []
| |___FreeColActionUI              Interfaces: [ActionListener]
| |___IntegerOptionUI              Interfaces: []
| |___LanguageOptionUI             Interfaces: []
| |___ListOptionUI                 Interfaces: [ListSelectionListener]
| |___ModOptionUI                  Interfaces: []
| |___SelectOptionUI               Interfaces: []
| |___SliderOptionUI               Interfaces: []
| | |___PercentageOptionUI         Interfaces: []
| | |___RangeOptionUI              Interfaces: []
| |___StringOptionUI               Interfaces: []
| |___UnitTypeOptionUI             Interfaces: []
```

Figure 5.14

In this hierarchy the “OptionUI” class knows of these subclasses from the following method (Figure 5.15) which uses a switch-like statement with many instance of checks to return instances of its subtype:

```
    @SuppressWarnings("unchecked")
    public static OptionUI getOptionUI(GUI gui, Option option, boolean
editable) {
        if (option instanceof BooleanOption) {
            return new BooleanOptionUI(gui, (BooleanOption) option, editable);
        } else if (option instanceof FileOption) {
            return new FileOptionUI(gui, (FileOption) option, editable);
        } else if (option instanceof PercentageOption) {
            return new PercentageOptionUI(gui, (PercentageOption) option,
editable);
        } else if (option instanceof RangeOption) {
            return new RangeOptionUI(gui, (RangeOption) option, editable);
        } else if (option instanceof SelectOption) {
            return new SelectOptionUI(gui, (SelectOption) option, editable);
        } else if (option instanceof IntegerOption) {
            return new IntegerOptionUI(gui, (IntegerOption) option, editable);
        } else if (option instanceof StringOption) {
            return new StringOptionUI(gui, (StringOption) option, editable);
        } else if (option instanceof LanguageOption) {
            return new LanguageOptionUI(gui, (LanguageOption) option,
editable);
        } else if (option instanceof AudioMixerOption) {
            return new AudioMixerOptionUI(gui, (AudioMixerOption) option,
editable);
        } else if (option instanceof FreeColAction) {
            return new FreeColActionUI(gui, (FreeColAction) option, editable);
        } else if (option instanceof AbstractUnitOption) {
            return new AbstractUnitOptionUI(gui, (AbstractUnitOption) option,
editable);
        } else if (option instanceof ModOption) {
            return new ModOptionUI(gui, (ModOption) option, editable);
        } else if (option instanceof UnitListOption) {
            return new ListOptionUI(gui, (UnitListOption) option, editable);
        } else if (option instanceof ModListOption) {
            return new ListOptionUI(gui, (ModListOption) option, editable);
        } else {
            return null;
        }
    }
```

Figure 5.15

Wide Hierarchy

There are 20 instances of the Wide Hierarchy smell in FreeCol. This is a comparatively large count of instances of this smell compared to the count of classes this system (1310 according to the Qualitas Corpus) and what was observed in other systems. Some cases of it are subclasses of Java Swing types such as the "JPanel" and "JComponent", two classes which commonly have a large count of direct children. Further levels of inheritance have been applied to some of these subclasses but these subclasses have high counts of subclasses also. An example is "FreeColDialog" which is a subclass of "JPanel" (which has 8 direct subclasses) and has 29 direct subclasses. Also, "FreeColAction" has 40 direct subclasses even though there exist intermediate types in its hierarchy. Many of these action classes have near identical code which simply delegates to other methods in the GUI when an action is performed. Similarly, "FreeColDialog" (29 subclasses) and "FreeColPanel" (22 subclasses) have a high count of subclasses that define different kinds of "JPanel" that will be created for the games' GUI. The textbook recommends that when many classes exist that differ only by their field values they can be replaced with a single class that loads field data from an enumeration.

Deep Hierarchy

The existence of many deep hierarchies in this system is a result of it being bottom heavy. What this means is that the classes such as “FreeColDialog” and “FreeColPanel” exist at 6 levels of inheritance from “Object” and have a high number of subclasses. The depth is explained by the fact that “JPanel” exists at 4 levels of inheritance. This approach shows that for the game FreeCol the approach of extending “JPanel” is preferred over the use of composition which is common for this system.

5.6.4 Conclusions

This system has a high count of each smell detected by the software tool and this is a result of creating separate classes for each Action or GUI element existing in the game, even though many of these classes usually contain single method delegation with most operations ultimately being handled by large “God Classes”. The underlying philosophy of using inheritance and creating specific types for each action results in some classes such as Canvas.java (2340 lines with comments) handling all GUI and Action types for the game in a tightly coupled manner with many type specific methods. The consequence of this design is that when one adds a new type of action or information dialog/panel they will have to also modify GUI.java, Canvas.java and any other classes that require knowledge of the specific types. This architecture results in the high count of cyclic hierarchies and makes extending the software a more complicated process than if it was “programmed to an interface instead of an implementation” and “favours inheritance over composition” in many instances. The result is a tightly integrated system that offers little to facilitate modification, reuse or extension.

5.7 Marauroa 3.8.1

5.7.1 Description

Marauroa is a Java framework for creating online games such as Arianne or Stendhal. This system has been selected because of the design issues observed in FreeCol. Marauroa has been through various rewrites and redesigns over the years and it will be interesting to see if this has resulted in a system that displays a sound design for extension and reuse of types as it is as much a framework as it is a game. This system is comparatively small with 184 .java files.

5.7.2 Summary

Multipath Hierarchy: 0

Cyclic Hierarchy: 33

Wide Hierarchy: 3

Deep Hierarchy: 0

5.7.3 Findings

Multipath Hierarchy

There are no instances of the multipath hierarchy smell in this framework.

Cyclic Hierarchy

There are 33 instances of this smell in the system. This most of them are from the "MessageHandler" abstract class depending indirectly on its concretions.

Wide Hierarchy

There are two instances of this smell in the system. Message with 29 subclasses and "MessageHandler" with 12. These subclasses are used instead of switch type structures and contain almost identical code except for some strings and could be replaced with a single class that loads those messages from an enumeration. The multiple-class approach appears to be very common in object oriented systems and is preferred to the use of switch-type structures which are also commonly observed to generate the cyclic hierarchy smell.

Deep Hierarchy

There are no instances of the Deep Hierarchy smell in this system. The deepest level of inheritance is 5 for some exceptions and the "LinkedRPObjectList" class which extends the Java type "LinkedList".

5.7.4 Conclusions

This system makes little use of inheritance and displays a very shallow class hierarchy. With most classes being direct subclasses of "Object" it is reasonable that it does not display many of the inheritance smells. It is positive that this is the case because classes provided by a framework are expected to be reused and extended without the final product (the game) displaying software smells before it is created but the low count of classes in the system may also be a contributing factor.

5.8 ArgoUML 0.34

5.8.1 Description

ArgoUML is an application for generating UML diagrams. It is a large system from the Qualitas Corpus with 2560 classes and has been selected as one of the "diagram generator/data visualization" systems that will be analysed. It is a commonly studied system and although it has existed since 1999, it is still in beta. Although it is popular, development is relatively stagnant with the latest release being in 2011.

5.8.2 Summary

Multipath Hierarchy: 28

Cyclic Hierarchy: 518

Wide Hierarchy: 26

Deep Hierarchy: 45

5.8.3 Findings

Multipath Hierarchy

The multipath hierarchy is generally avoided in this system with top level classes only implementing the interfaces. Below is an example (Figure 5.16) where only Checklist (org.argouml.cognitive.checklist) implements the interface "Serializable" but not its subclass. The Java type "ArrayList" also implements it but it is justifiable to reimplement the interface in the top local project class (as it is the highest level of the hierarchy in the local project). This is visible in the hierarchy extract below (Figure 5.16):

```
Object
|___AbstractCollection         Interfaces: [Collection]
| |___AbstractList             Interfaces: [List]
| | |___ArrayList    Interfaces: [List, RandomAccess, Cloneable, Serializable]
| | | |___Checklist            Interfaces: [List, Serializable]
| | | | |___ChecklistStatus    Interfaces: []
```

Figure 5.16

Another example of an instance of multipath hierarchy in this system is the following hierarchy extract. Here it can be seen that the "StylePanel" implements 5 interfaces. The subclasses "StylePanelFig" and "StylePanelFigAssociationClass" implement three of which one which is also inherited. The result is that "StylePanelFigNodeModelElement" inherits 7 interfaces, re-implements 3 of them and then implements an additional one. This happens while all the other classes in the tree do not implement any interfaces explicitly. The issue is that the "StylePanel" class implements many of the methods in the interfaces with empty bodies, which is not technically a "Rebellious Hierarchy" because the methods being rejected are never implemented, however, such a la carte implementation of interfaces can be an issue for those who intend to reuse and maintain the software as the promised interfaces are never actually implemented (Figure 5.17).

```
Object
|___Component                                       Interfaces: [ImageObserver, MenuContainer, Serializable]
| |___Container                                     Interfaces: []
| | |___JComponent                               Interfaces: [Serializable, TransferHandler.HasGetTransferHandler]
| | | |___JPanel                                    Interfaces: [Accessible]
| | | | |___AbstractArgoJPanel                      Interfaces: [Cloneable, Orientable]
| | | | | |___StylePanel                            Interfaces: [TabFigTarget, ItemListener, DocumentListener,
ListSelectionListener, ActionListener]
| | | | | | |___StylePanelFig                       Interfaces: [ItemListener, FocusListener, KeyListener]
| | | | | | | |___StylePanelFigNodeModelElement    Interfaces: [ItemListener, FocusListener, KeyListener,
PropertyChangeListener]
| | | | | | | | |___StylePanelFigClass             Interfaces: []
| | | | | | | | |___StylePanelFigInterface         Interfaces: []
| | | | | | | | |___StylePanelFigPackage           Interfaces: []
| | | | | | | | |___StylePanelFigAssociationClass  Interfaces: [ItemListener, FocusListener, KeyListener]
| | | | | | | | |___StylePanelFigMessage           Interfaces: []
| | | | | | | | |___StylePanelFigUseCase           Interfaces: []
```

Figure 5.17

Cyclic Hierarchy

For the size of the project (2560 classes), this system displays a comparatively small count of this smell (518 paths). These appear to be deliberate versions of the smell with many classes like "XmiFilePersister" and "AbstractFilePersister" depending on subclasses via the "PersistenceManager" class, Actions and UMLDiagram depending on their subclasses via the "ProjectManager" class and other similar situations. This appears to be a feature of the software design and there are no instances (of the subset that was checked) where this is done through the constructor of each class.

Wide Hierarchy

There are 33 wide hierarchies in this system. Many of these are of the commonly observed types in other Java systems such as direct subclasses of Swing types and Actions and Exceptions. Specifically, for JCL classes in this system, there are 35 subclasses of "JPanel", 13 of "JComponent", 10 of "Exception" and 39 of "AbstractAction". There are also some project classes with high counts of subclasses such as "UndoableAction" with 75, "CrUML" with 91, "UMLModelElementListModel" with 77 and "AbstractPerspectiveRule" with 79. This is in line with what can be observed in most other systems. The Wide hierarchies usually exist when intermediate classification cannot exist or creation of intermediate types is not meaningful.

Deep Hierarchy

Most instances of this smell are justifiable. Sixteen instances of this smell are "UMLCheckBox2" and its direct and indirect subclasses, these all extend "JCheckBox", a Swing type which has 5 superclasses. The other classes which are deep in the hierarchy are subclasses of another Swing type, “JPanel”, and therefore have an inflated depth in the hierarchy.

5.8.4 Conclusions

ArgoUML shows the same sort of wide and deep Hierarchies that can be observed in most other Java systems that were analysed. These are generally extensions of Swing types and Objects used to internally represent the software. There are some instances where the Multipath Hierarchy appears to exist inconsistently and methods are implemented with an empty body which can be problematic. The Cyclic Hierarchy does exist but its presence is insignificant, especially compared to what is observed in other Systems.

5.9 FreeMind 0.9.0

5.9.1 Description

FreeMind is a multi-platform mind mapping software tool which uses Swing for its user interface. Mind mapping software is used to organize thoughts and ideas by organising them hierarchically and creating connections between concepts. This tool is commonly analysed in software studies and was selected because it is a mid-sized system and represents another system in the “diagram generator/data visualization” category, albeit in a different context than ArgoUML.

5.9.2 Summary

Multipath Hierarchy: 3

Cyclic Hierarchy: 9934 (depth 5)

Wide Hierarchy: 8

Deep Hierarchy: 0

5.9.3 Findings

Multipath Hierarchy

FreeMind has a low count of this smell which allows closer examination of this system. Two instances of it are in the class “NodeMotionListener” which reimplements the interfaces “MouseListener” and “MouseMotionListener”, but not “MouseWheelListener”, which are inherited from the JCL abstract class “MouseAdapter”. The “MouseAdapter” class does not provide implementations for the methods defined in them and defers to subclasses for that. The result is that the method “mouseWheelMoved” has not been implemented even though the inherited interface suggests it has.

Cyclic Hierarchy

The Cyclic Hierarchy smell exists in this system and these are mostly instances of the same classes involved in the Wide Hierarchy below depending on their subclasses. This is mostly Actions that depend on their subclasses through the "MindMapController" class via multiple paths. This appears to be intended design and the cyclic hierarchy smell is very common in systems or frameworks that have or provide classes for creating a User Interface.

Wide Hierarchy

The Wide Hierarchy smells in the FreeMind system mostly (5 out of 8) exist in two kinds of classes that have already been observed in other systems to produce wide hierarchies. These are Swing UI elements and their associated Swing Actions. Another less common wide part of the hierarchy was that of "MindMapNodeHookAdapter" with 14 subclasses, a type of Class Adapter that exists in this project. Class Adapters necessarily extend the Class that they are adapting so it appears to be an unavoidable smell for this category of classification also.

Deep Hierarchy

The tool does not detect any instances of the Deep Hierarchy smell for FreeMind.

5.9.4 Conclusions

FreeMind appears to display a low occurrence of smells and from this facet of analysis is a very well designed system. It has no deep hierarchies and the few wide ones are consistent with what is seen in other systems. Cyclic hierarchies exist but are of the kind observed in most systems.

5.10 Apache Ant 1.8.4

5.10.1 Description

Apache ant is an alternative to GNU make for automating software builds and is used primarily for Java projects. It can easily integrate JUnit tests and is very popular with Java developers although newer alternatives such as Apache Maven have emerged. This system is a popular Java system to study and was included for that reason.

5.10.2 Summary

Multipath Hierarchy: 37

Cyclic Hierarchy: 4274

Wide Hierarchy: 9

Deep Hierarchy: 3

5.10.3 Findings

Multipath Hierarchy

Multipath hierarchy in this system is detected in 37 instances. From these 32 are instances of the interface “Cloneable” which is a common observation. Another interface, “HotDeploymentTool” is present in three instances of this smell and is inconsistently used in the four classes that implement it. The abstract superclass “AbstractHotDeploymentTool” implements two of the methods, “setTask” and “validateAttributes” and its subclass “GenericHotDeploymentTool” does not explicitly reimplement the interface but provides an implementation for the third method “deploy”. The subclass of “GenericHotDeploymentTool”, “JonasHotDeploymentTool” reimplements the interface and overrides only the method “validateAttributes” while the only other subclass of “AbstractHotDeploymentTool”, “WebLogicHotDeploymentTool” also reimplements the interface, overrides the “validateAttributes” method and implements the “deploy” method. This kind of inconsistency is unnecessary and can be confusing, leading to errors. The extract can be seen in Figure 5.18:

```
Object
|___AbstractHotDeploymentTool    Interfaces: [HotDeploymentTool]
| |___GenericHotDeploymentTool   Interfaces: []
| | |___JonasHotDeploymentTool   Interfaces: [HotDeploymentTool]
| |___WebLogicHotDeploymentTool  Interfaces: [HotDeploymentTool]
```

Figure 5.18

Additionally, the abstract class “BasicShape” implements “DrawOperation” but does not provide a default implementation for the method in it, “executeDrawOpertation”, this is not a problem but the multiple paths are unnecessary. In the tree extract below (Figure 5.19) we can also observe one of the instances of “Cloneable” being implemented by “Datatype” when inherited from “ProjectComponent” but subsequently not by its children classes.

```
Object
|___ProjectComponent    Interfaces: [Cloneable]
| |___DataType          Interfaces: [Cloneable]
| | |___ImageOperation  Interfaces: []
| | | |___BasicShape    Interfaces: [DrawOperation]
| | | | |___Arc         Interfaces: [DrawOperation]
| | | | |___Ellipse     Interfaces: [DrawOperation]
| | | | |___Rectangle   Interfaces: [DrawOperation]
```

Figure 5.19

Cyclic Hierarchy

This system has various instances of the Cyclic Hierarchy smell. These instances cannot be all confirmed individually as in other systems but it appears that it this smell is shown where

Resources, Datatypes and Shapes depend on their subclasses. Ideally, this smell would not exist but it appears to be present in most Java systems with many types.

Wide Hierarchy

There are some wide branches in the ant hierarchy. This is a result of 32 direct subclasses for the "Datatype" class, 126 subclasses of Task and other common wide hierarchies such as subclasses of "DefaultCompilerAdapter". Such wide hierarchies are common and do not warrant further investigation as they do not seem irregular for the software type or display instances of cloned code.

Deep Hierarchy

The three situations where this hierarchy exists in ant are cases where classification is present and appears justified. There are no indications of a speculative hierarchy or a misapplication of taxonomy. The deep hierarchy chain that results in Taskdef is visible below (Figure 5.20), in this case the hierarchy extract contains some classes that are not related to the class being examined so that it can be visible how the tree does not have speculative classification some of the names suggest that there may be some instance of inheritance that doesn't comply with the "IS-A" relationship requirement such as definitions being a subclass of "Task". This was not fully explored but may be worth checking for a Broken Hierarchy smell.

```
Object
|___ProjectComponent
| |___Task
| | |___DispatchTask
| | | |___Symlink
| | |___TaskAdapter
| | |___AbstractCvsTask
| | | |___Cvs
| | | |___ChangeLogTask
| | | |___
| | | |___CvsVersion
| | |___AbstractJarSignerTask
| | | |___SignJar
| | | |___VerifyJar
| | |___Ant
| | |___Antlib
| | |___AntlibDefinition
| | | |___DefBase
| | | | |___Definer
| | | | | |___Componentdef
| | | | | |___Typedef
| | | | | | |___Taskdef    ← The class taskdef
| | | | |___ScriptDef
| | | |___MacroDef
| | | |___PreSetDef
```

Figure 5.20

5.10.4 Conclusions

Ant does show some inconsistencies with the way interfaces are implemented and this may be the result of multiple developers introducing classes in a different manner. The other attributes examined appear to be in line with what has been observed in other systems with some wide hierarchies and a low count of deep hierarchies. Overall, the printed output of the hierarchy tree shows that classification has been applied and the hierarchy appears to be balanced.

6 Discussion

6.1 Summary of Findings

The central theme of this study has been to explore the advantages of studying object-oriented systems from the aspect of hierarchical smell patterns. The intent is to determine if these patterns are common and whether there is utility in studying them. The discovery was that to a degree every system displayed instances of these smells for each of the four hierarchical smell patterns that were selected. The other finding was that there is much utility to performing static analysis of this nature on software systems because it does not only produce indications of smells but allows a developer to quickly learn critical information about how the system is structured by understanding the hierarchy and dependence paths between classes.

6.2 Findings per Hierarchical Smell

6.2.1 Deep Hierarchy

Automatically calculating the depth of classes in the hierarchy tree showed consistent findings. That is that in the systems studied, most deep hierarchies were the result of extending classes from the Java Class Libraries. These are usually swing types such as "JDialog", "JPanel" or "JToolBar". If the alternative approach of examining the depth of the hierarchies only within the individual systems types then the smell, in the systems studied, would be rare. Most of the utility of searching for this smell comes from some rare cases such as the example from the "corba" package in the Java Class Libraries. When such elements are discovered, collapse hierarchy or an appropriate refactoring ought to be considered.

6.2.2 Wide Hierarchy

Studying the subset of systems from the Qualitas Corpus showed that wide hierarchies are commonplace in these Java systems. This discovery would be true even if a larger threshold than the proposed 9 was used for this smell because many of these hierarchies are much wider than that. Wide hierarchies appeared to be commonly connected to specific types of classes and many times the suggested refactoring of adding intermediate types is not suitable. The categories of classes for which high values of the NOC metric were widely observed are:

- **GUI Elements,** such as "JPanel" and "JDialog".
- **Actions,** these are associated with GUI elements and provide the related functionality when these elements are interacted with.
- **Errors or Exceptions,** these subclasses of "Throwable" very commonly require multiple immediate subtypes to be thrown for different faults in the software.

- **Domain Specific Types,** these are unique to each software. In a game these may be the various units and in a drawing application the different drawing shapes, tools or adapter classes required.

Wide hierarchies of these types are too common to be considered an issue without considering other factors. It is more important to observe how these classes are integrated into the software to see if they affect the quality attributes of the software. In the inspection of FreeCol it was observed that classes in a wide hierarchy are all handled by single "God Classes" that know specifically about them and handle each with separate methods, this breaks the "program to an interface, not to an implementation" guideline and would complicate the task of adding new Object types to the system. This is more likely to happen with instances of project types instead of Swing Elements/Actions and Exceptions/Errors as those are handled by the Java Framework directly. Finally, it appears that developers prefer making different classes for different kinds of notifications or similar UI elements rather than using a single class and loading the information from a data structure such as an enumeration. Such a design would be complicated furthermore if the software was required to support multiple languages.

6.2.3 Multipath Hierarchy

Developers appear to use interfaces differently in each system. Some implement the interface at the highest level in the project, ignoring what is inherited from JCL types, while others deliberately implement them at every level of inheritance, possibly as a signal of intent. The most common interfaces represented in this smell are "Serializable" and "Cloneable" which are possibly also the most commonly used Java interfaces (this cannot be confirmed because there do not appear to be any empirical studies on the frequency of interfaces or classes in the Qualitas Corpus, this could be an interesting topic to explore). Importantly it was discovered that it is common to "implement" an interface without providing actual functionality for the methods provided. This behaviour does result in technical debt as classes do not provide the functionality they promise and may suggest that programmers are more interested in reusing classes provided by other developers rather than making their own reusable. Studying this smell with the methods of automatic detection, examination of the inheritance tree and code inspection shows that locating instances of this smell is useful because removing the unnecessary "implements" statements can make the program clearer and more consistent in its implementation.

The high counts of "Serializable" and "Cloneable" in the multipath smell is also possibly linked to these interfaces being what is known as a "Marker Interface" (Gößner,Mayer and Steimann, 2004). A marker interface is an empty interface that is required by some operation as a guarantee of compatibility. The issue with marker interfaces is that once implemented, they also inherently mean that the subclasses of these objects also support the operation even if this behaviour is not intended. Some programmers may explicitly implement the interface on the subclasses as an indication of this intent and it is probably the responsible thing to do especially in complex systems. Because of this marker interfaces may also need to be exempted from detection.

To support the claim that these interfaces are common a study (Gößner,Mayer and Steimann, 2004) has discovered that in the JDK (referred to as the JCL in this study) the "Serializable" interface is 22.8% of all interface implementations which may also provide an explanation for the frequency of their detection in the Multipath Hierarchy smell.

Finally, there is also an argument that these interfaces do not behave as interfaces should because they do not define publicly available class data or behaviour. According to the documentation (Oracle, 2016b) "The serialization interface has no methods or fields and serves only to identify the semantics of being serializable". It does, however, mention a field "serialVersionUID" which can be added explicitly or calculated automatically by the serialization runtime. Similarly, according to the documentation for the interface "Cloneable" (Oracle, 2016a), it is also an empty interface but overriding and calling the "clone" method (a protected method inherited from the Object class) results in a CloneNotSupportedException being thrown if the interface is not "implemented". This suggests that an approach akin to languages such as C# may be more suitable. In C#, the "Clone" method is defined in the interface "ICloneable" (Microsoft, 2017a) while "Serializable" (Microsoft, 2017c) is a non-inheritable attribute rather than an interface. For situations where inheritance of the attribute is preferred, there is also an interface called "ISerializable" (Microsoft, 2017b).

6.2.4 Cyclic Hierarchy

The cyclic hierarchy smell as not fully explored in this study. The reason behind this was the computational cost of searching through each classes dependencies, so the search was limited to 5 levels. Even with this practical limit the search in Eclipse SDK took 5 hours and found 1483205 paths. These had to be verified through random selection. What was observed when searching with a small threshold was that many of these examples are cases where an abstract or concrete class knows of its subclasses through the context of a switch or equivalent if statements. Use of switch statements (or equivalent programming structures using conditionals) which know of specific subclasses ties the implementation to these subclasses. This kind of coupling means that adding more subclasses requires modification of the parent class which breaks the open/closed principle that requires classes to be open for extension but closed to modification. Switch statements that know of their subclasses can often be replaced with a solution that uses polymorphism such as the design patterns state or strategy.

The more severe case of Cyclic Dependency where a class's constructor would depend on a possibly uninitialized subclass was not found and is probably not common because in Java systems it would usually result in an error. In the form this smell was detected it is not useful because the results cannot be used meaningfully. As a future work, an option would be to modify the tool so that only paths that begin from the dependencies of the constructor, instead of the whole class body, are searched indefinitely. This along with a low threshold examination of the rest of the class would result in a far smaller search space and more meaningful results as the detected paths would contain a much stronger indication of a problematic smell.

6.3 Evaluation of the Tool

The software tool proved to be an effective way to quickly analyse software systems and effectively guide the manual inspection of source code. Every instance of smell that was detected was confirmed to be a true positive but, although unexpected, it is possible that it may miss some instances where classes have the same names in different packages and alternate between using the full scope name (e. g "java.lang.Object") and the class name ("Object"). This could not be tested thoroughly given the time constraints but was not observed in practice. The command line interface and output format allow for filtering and measurement of results and it can confidently be used to output a text form tree structure of a hierarchy as well as detect the four smells it is intended to.

Development of this tool was done iteratively and much functionality was added during the application of the study either to mend bugs that were found or expand the functionality so that it could be used more effectively. A short list of the challenges involved in developing this tool is:

- **Performance,** because the software must parse JCL classes for every run, these were exported into a CSV-like file ("Java8.csv") that could be quickly read with each class and the information about it contained in each line. Run times were also improved by using HashMaps for connecting Strings to Class types instead of searching collections.
- **Flexibility,** command line arguments had to be implemented so that according to the users wish the different smells could be detected independently, thresholds could be redefined and optional features such as including Java 8 classes could be selected. This was done because the initial approach of analysing a system and creating a complete report of the findings was not practical for repeated runs and in depth inspection.
- **Usability,** the output had to be changed to a format that could be easily be filtered by tools such as "grep" and "wc". Also, some extra details such as interfaces implemented

and package were added to the tree print function so that deep and multipath hierarchies could be studied faster.

- **Bug fixes,** package resolution had to be added for all classes because in some cases some projects alternated between using the full package scope and simple names, this was also required for interfaces when detecting the Multipath Hierarchy smell because the tool compared the interface names without considering their package (e. g "java.io.Serializable" as opposed to "Serializable"). Additionally, some classes had to be automatically removed because their parent classes could not be found in the projects or they participated in circular hierarchies which produced stack overflow errors.

Future work may expand the tools' capability of performing a more complete job of hierarchical smell detection or could be expanded to provide more information about the system in general.

- An option is to include automatic detection of the consistency of existence of multipath hierarchies. If some parts of the system implement an interface at every level while others rely on inheritance this may be an indication of a problem.
- Automatic cloned code detection could be added so that it can be easier to see when classes in wide hierarchies can have some of their methods pushed up.
- Cyclic hierarchies can be limited to those originating from the dependencies of the superclass constructor.
- The redundant implementations of interfaces can be removed automatically in cases of the multipath hierarchy smell that are not used for marker interfaces.
- Interfaces can be parsed and linked to the methods they require so it can be found if these are being implemented with empty bodies.
- An "ignore list" may be added for wide hierarchies of specific types such as those that extend "AbstractAction" or "Throwable", so that results are more focused on wide hierarchies of uncommon types.

(INTENTIONALLY BLANK PAGE)

7 Recommendations and Conclusions

This attempt at studying software systems with the help of a static analysis tool that can automatically detect software smells shows that even when a smell can be described objectively, the specifics of the software system must be understood and considered before reaching the conclusion that they represent technical debt. This means that human examination of code may be in many cases necessary before deciding to apply refactoring. This has been known since the initial 22 smells were proposed by the authors Fowler and Beck (1999) who suggested that deciding when to refactor requires a measure of experience and intuition from the developer. Static analysis tools can, however, provide a quick introduction to a system and extract data that can assist developers in being more accurate and productive.

Deliberately following programming guidelines and enabling techniques such as seen in the systems JHotdraw or Marauroa appears to result in what can be described as good design with low counts of hierarchical bad smells even though they were designed and implemented before these smells were defined. For other systems, however, hierarchical smells are a useful aspect of software inspection because many bad design habits result in their presence. The definitions of smells may need to be reviewed if they are to be useful in practice because it was observed that the overly inclusive definitions of the hierarchical smells may be consistent with programming guidelines but may not useful for real world applications which appear to deliberately implement wide, multipath or cyclic hierarchies.

An appropriate revision of the smells would require a more specific justification and description of each. For the four smells studied in this work, it would be useful to specify some minor improvements for each of them.

- For Deep Hierarchies, it should be explicit if this includes types inherited from the programming language libraries or even toolkits such as Qt (Qt, 2017) in C++. The observations made by this study suggest that only project types should be counted and use a smaller depth threshold used (of possibly 3 to 4 classes) so that detection is oriented towards the few outlying instances such as that found in "corba" instead of subclasses of "JPanel".
- For the Wide Hierarchy smell, it may be useful to re-evaluate the characterisation of these as a smell as they are very common throughout the systems analysed. The threshold of more than 9 classes appears reasonable if the tool can automatically exclude those that are found commonly so that the focus can be on other cases which are unique to the system.
- For Cyclic Hierarchies, it is not meaningful to detect all paths due to the size of the search space and result output. The changes proposed in the tool evaluation would be a step towards a better understanding of the smell and produce more useable knowledge to

understand and pre-empt software malfunctions, this is more important for languages such as C++ because they do not use a managed environment. This is an important issue because uninitialized variables could be used without an Error or Exception being thrown.

- Multipath Hierarchies are the only smell from this list that can confidently be automatically repaired. This is sufficient for existing programs and a possible feature of IDEs could be to perform this check automatically or inform programmers with a warning so that they can be more mindful of overriding methods by mistake.

As an extension of this work, there will be future work undertaken that can further enhance the understanding the findings of this study. This will be done by creating a new version of the tool that can visualise results better with a GUI instead of a command line output. This will have the proposed enhancements from the previous chapter built-in as well as the capability to count interface implementations and class extensions for a whole project. This tool will be built to facilitate an extended study of all the systems in the Qualitas Corpus so that it can be revealed if interfaces such as "Cloneable" and "Serializable" are common only in these smells or are simply interfaces that see widespread use and therefore appear frequently in the Multipath Hierarchy smell. The same will happen for the Swing types.

Additionally, there is the option to expand this study to other languages, particularly those that implement a different approach to inheritance. The main option for this would be a study of C++ systems which can be expected to display different results for many of these smells, particularly the multipath hierarchy smell due to the use of multiple inheritance. Alternatively, a study of Python systems is a viable option because it is a popular scripting language that is commonly used in open source systems (Githut, 2017). Performing such a study does not necessarily require, but could be preceded by the creation of a software collection analogous to the Qualitas Corpus for these languages. Such a study could expand the findings and add more insight to the question of how these smells can be redefined or used.

The objectives of this work were to study the background of the hierarchical smells defined in the textbook (Suryanarayana, Samarthyam and Sharma, 2014), build a tool that could detect as many of these as possible and uncover how common these are in open source software as well as detect examples of the smells. These steps were performed and a tool was built to detect four of these smells which was then applied to ten systems. From the results of the output produced by the tool and manual inspection of the selected systems, it was found that these smells exist not only in theory but also in the code of open source software systems that are widely used. The discussion and analysis of the patterns observed reveals that there are specific pathways through which these smells are introduced and that sometimes they are not avoidable. These findings add some additional insight and aid the understanding these patterns by example.

(INTENTIONALLY BLANK PAGE)

8 References

Beck, K., Fowler, M. and Beck, G. (1999) 'Bad smells in code'. *Refactoring: Improving the design of existing code,* 75-88.

Chidamber, S.R. and Kemerer, C.F. (1991) *Towards a metrics suite for object oriented design.* ACM.

Chidamber, S.R. and Kemerer, C.F. (1994) 'A metrics suite for object oriented design'. *IEEE Transactions on software engineering,* 20 (6), pp. 476-493.

Cook, W.R., Hill, W. and Canning, P.S. (1989) Published. 'Inheritance is not subtyping'. *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages,* 1989. ACM, pp.125-135.

Copeland, T. (2005) *PMD's Copy/Paste Detector.* Available at: https://pmd.github.io/ (Accessed: 07-07-2017).

Eclipse (2013) *Class ASTVisitor* Available at: http://help.eclipse.org/kepler/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2FASTVisitor.html (Accessed: 18-08-2017).

Ferreira, K.A. *et al.* (2012) 'Identifying thresholds for object-oriented software metrics'. *Journal of Systems and Software,* 85 (2), pp. 244-257.

Fokaefs, M., Tsantalis, N. and Chatzigeorgiou, A. (2007) Published. 'Jdeodorant: Identification and removal of feature envy bad smells'. *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on,* 2007. IEEE, pp.519-520.

Fokaefs, M. *et al.* (2011) Published. 'JDeodorant: identification and application of extract class refactorings'. *Proceedings of the 33rd International Conference on Software Engineering,* 2011. ACM, pp.1037-1039.

Foundation, E. (2017) *Eclipse Java Development Tools (JDT).* Available at: https://www.eclipse.org/jdt/ (Accessed: 18-08-2017).

Fowler, M. and Beck, K. (1999) *Refactoring: improving the design of existing code.* Addison-Wesley Professional.

Ganesh, S., Sharma, T. and Suryanarayana, G. (2013) 'Towards a Principle-based Classification of Structural Design Smells'. *Journal of Object Technology,* 12 (2), pp. 1:1-29.

Githut (2017) Available at: http://githut.info/ (Accessed: 16-08-2017).

Google (2010) *The Go Programming Language FAQ.* Available at: https://golang.org/doc/faq - inheritance (Accessed: 2017-06-19).

Gosling, J. *et al.* (2015) *The Java® Language Specification.* Available at: https://docs.oracle.com/javase/specs/jls/se8/html/jls-4.html - jls-4.3.2 (Accessed: 2017-06-21).

Gößner, J., Mayer, P. and Steimann, F. (2004) Published. 'Interface utilization in the JAVA Development Kit'. *Proceedings of the 2004 ACM symposium on Applied computing,* 2004. ACM, pp.1310-1315.

ISO (2015) *ISO Information technology -- Vocabulary.* Available at: https://www.iso.org/obp/ui/ - iso:std:iso-iec:2382:ed-1:v1:en (Accessed: 2017-07-03).

Karp, R.M. and Rabin, M.O. (1987) 'Efficient randomized pattern-matching algorithms'. *IBM Journal of Research and Development,* 31 (2), pp. 249--260.

Kay, A. (2003) *Dr. Alan Kay on the Meaning of Object-Oriented Programming.* Available at: http://www.purl.org/stefan_ram/pub/doc_kay_oop_en (Accessed: 2017-07-03).

Lopes, C.V. (2014) *Exercises in programming style.* CRC Press.

Mantyla, M., Vanhanen, J. and Lassenius, C. (2003) Published. 'A taxonomy and an initial empirical study of bad smells in code'. *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on,* 2003. IEEE, pp.381-384.

Martin, R.C. (2005) *The Principles of OOD.* Available at: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod (Accessed: 2017-07-05).

Mazinanian, D. *et al.* (2016) Published. 'JDeodorant: clone refactoring'. *Proceedings of the 38th International Conference on Software Engineering Companion,* 2016. ACM, pp.613-616.

Meyer, B. (1997) 'Object-oriented software construction'.

Microsoft (2017a) *ICloneable Interface (System).* Available at: https://msdn.microsoft.com/en-us/library/system.icloneable(v=vs.110).aspx (Accessed: 17-08-2017).

Microsoft (2017b) *ISerializable Inteface (System.Runtime.Serialization).* Available at: https://msdn.microsoft.com/en-us/library/system.runtime.serialization.iserializable(v=vs.110).aspx (Accessed: 17-08-2017).

Microsoft (2017c) *SerializableAttribute Class (System).* Available at: https://msdn.microsoft.com/en-us/library/system.serializableattribute(v=vs.110).aspx (Accessed: 17-08-2017).

Oracle (2012) *The Java™ Tutorials: What Is Inheritance?* Available at: https://docs.oracle.com/javase/tutorial/java/concepts/inheritance.html (Accessed: 2017-06-21).

Oracle (2016a) *Cloneable (Java Platform SE 7).* Available at: https://docs.oracle.com/javase/7/docs/api/java/lang/Cloneable.html (Accessed: 17-08-2017).

Oracle (2016b) *Serializable (Java Platform SE 7).* Available at: https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html (Accessed: 17-08-2017).

Qt (2017) *Qt Reference Pages 5.9.* Available at: http://doc.qt.io/qt-5/reference-overview.html (Accessed: 18-08-2017).

Rentsch, T. (1982) 'Object oriented programming'. *ACM Sigplan Notices,* 17 (9), pp. 51-57.

Sherif, J.S. and Sanderson, P. (1998) 'Metrics for object-oriented software projects'. *Journal of Systems and Software,* 44 (2), pp. 147-154.

Stroustrup, B. (2012) *Bjarne Stroustrup's C++ Glossary.* Available at: http://www.stroustrup.com/glossary.html - Gpolymorphism (Accessed: 11-8-2017).

Suryanarayana, G., Samarthyam, G. and Sharma, T. (2014) *Refactoring for software design smells: managing technical debt.* Morgan Kaufmann.

Technopedia (2017) *Technical debt definition.* Available at: https://www.techopedia.com/definition/27913/technical-debt (Accessed: 2017-06-26).

Tempero, E. (2010) *Qualitas Corpus.* Available at: http://qualitascorpus.com/ (Accessed: 07-07-2017).

Tempero, E. *et al.* (2010) Published. 'The Qualitas Corpus: A curated collection of Java code for empirical studies'. *Software Engineering Conference (APSEC), 2010 17th Asia Pacific,* 2010. IEEE, pp.336-345.

Tsantalis, N., Chaikalis, T. and Chatzigeorgiou, A. (2008) Published. 'JDeodorant: Identification and removal of type-checking bad smells'. *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on,* 2008. IEEE, pp.329-331.

Van Deursen, A. *et al.* (2001) Published. 'Refactoring test code'. *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001),* 2001. pp.92-95.

Van Emden, E. and Moonen, L. (2002) Published. 'Java quality assurance by detecting code smells'. *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on,* 2002. IEEE, pp.97-106.