# An Analysis of Inheritance Hierarchy Evolution

Murray I Wood
Dept. of Computer and Information Sciences
University of Strathclyde
Glasgow, UK, G1 1XH
murray.wood@strath.ac.uk

Lyubomir Ivanov
Dept. of Computer and Information Sciences
University of Strathclyde
Glasgow, UK, G1 1XH
lyubomir.ivanov.2015@uni.strath.ac.uk

Zenon Lamprou
Dept. of Computer and Information Sciences
University of Strathclyde
Glasgow, UK, G1 1XH
zinon.lamprou.2015@uni.strath.ac.uk

## ABSTRACT

This research investigates the evolution of object-oriented inheritance hierarchies in open source, Java systems. The paper contributes an understanding of how hierarchies, particularly large complex hierarchies, evolve in 'real world' systems. It informs object-oriented design practices that aim to control or avoid these complicated design structures. The study is based on a detailed analysis of 665 inheritance hierarchies drawn from a total of 262 versions of 10 open source systems. The research contributions include that: i) the majority of inheritance hierarchies are 'simple' in structure and remain that way throughout their lifetimes ii) the majority of hierarchies are stable in terms of size and shape throughout their lifetimes iii) there is a minority of large, complex, branching 'Subtree' hierarchies that continue to grow ever more complicated as the systems evolve iv) a detailed analysis of some of these larger hierarchies finds evidence of 'good' object-oriented design practices being used but also highlights the significant challenges involved in understanding and refactoring these complex structures. There is clear evidence that some of the complex hierarchies are emphasising reuse while others appear focused on type inheritance.

## CCS CONCEPTS

• Software and its engineering → Abstraction, modeling and modularity

## KEYWORDS

inheritance; hierarchy; evolution; empirical; open source; case study.

**ACM Reference format:**

FirstName Surname, FirstName Surname and FirstName Surname. 2019. Insert Your Title Here: Insert Subtitle Here. In *Proceedings of EASE 2019*

## 1 INTRODUCTION

The aim of this research is to analyse and understand how object-oriented inheritance hierarchies evolve across multiple versions of software systems. The goal is to contribute to practical design guidance on the use of object-oriented inheritance. This research studies how the size and shape of hierarchies evolve across multiple versions, how long they appear to 'live' for, and provides a detailed analysis of the evolution of some of the largest, most complex, multi-branching hierarchies.

It uses a purpose-built tool that provides a high-level visual summary of each hierarchy in a system across all its versions. The tool supports drilling into individual hierarchies, again providing a visual summary of each hierarchy. The Eclipse IDE is then used to provide a detailed examination of the hierarchy code and its classes. The tool is applied to 10 open source systems from the Qualitas Corpus Evolution package [19]. At least 16 versions are analysed for each of the 10 systems, 262 versions in total, with a sum of 665 inheritance hierarchies studied.

Inheritance is a core but controversial feature of most object-oriented approaches to design and implementation. A recent survey of practitioners found mixed views on how inheritance is used in practice [16]. One difficulty is that inheritance is used to support two distinct properties in mainstream languages such as Java - type inheritance (polymorphism) and module reuse. Other difficulties stem from depth of inheritance hierarchy [4], overriding of method definitions, and 'self calls' - where method calls are being propagated up a hierarchy and, potentially, out into the surrounding system.

The contributions made by this paper include a confirmation of the dominance of small, very simple hierarchies allowing design effort to be focused on a relatively small number of complex hierarchies. It also shows that the majority of hierarchies appear stable in terms of size and shape. It finds that the lifespan of many hierarchies seems quite short but this may be due to their classes being subsumed into other hierarchies.

The most significant contribution comes from the detailed analysis of the complex 'Subtree' hierarchies [17], how they evolve and their core properties. Some of these large hierarchies clearly exhibit the properties of type inheritance, focusing on the potential to cleanly substitute subclasses for superclasses. Other hierarchies are much more focused on reuse, requiring extensive use of type checking and casting.

## 2 RELATED WORK

Inheritance use is complicated by its dual roles, as a reuse mechanism and as a type substitution mechanism. The Liskov Substitution Principle (LSP) [9] imposes an extreme constraint on hierarchy design requiring a subclass to be a semantic substitute for a superclass and not to break the behaviour of any system in which the subclass is used as a substitute for the superclass (also known as ‘is-a’ inheritance).

Liskov also identifies “*convenience inheritance*”, where inheritance is used simply as a reuse mechanism, as a weak form of usage. While Liskov argues that reuse and subtyping should be kept separate [8], Meyer argues “*If we accept classes as both modules and types, then we should accept inheritance as both module accumulation and subtyping*” [10].

It is argued that inheritance overuse can lead to programs that are difficult to understand and change, because of the need to traverse up, down and across hierarchies to fully understand runtime behaviour [2, 18]. The concept of ‘fragile base classes’ has also been identified, where changes in a superclass may break a subclass or its dependents [11] – though recent work disputes how much impact fragile base classes actually have in practice [15]. Addressing the dangers of unintended inheritance interactions, Bloch argued that developers should “*design and document for inheritance or else prohibit it*” [1].

In their design patterns catalogue, Gamma et al. introduce the principle of “*favouring object composition over class inheritance*” [5], arguing that composition should be preferred as a reuse mechanism (though many of their patterns still use inheritance).

Previous work analysing inheritance use in practice has found significant inheritance usage, with Tempero et al. finding that across 93 applications from the Qualitas Corpus [19, 20] “*around three-quarters of user-defined classes use some form of inheritance in at least half the applications in our corpus*”. They also found that most classes appear in shallower hierarchies, two-thirds of inheritance uses were for type-substitutability, and that around 20% of uses could have been achieved using composition instead of inheritance. Collberg et al. also found a predominance of shallow hierarchies and a small number of large outliers [3], with a depth of inheritance all the way up to 39.

Recently, Stevenson and Wood [17] identified a number of ‘patterns’ of inheritance usage in a study of 2440 hierarchies from 14 open source systems taken from the Qualitas Corpus. They identified five different categories of hierarchy shape – Line, Branch, Line-Branch, Branch-Line and Subtree. They found that 74% of hierarchies were either Line (width = 1) or Fan (depth = 1) shape. Fifteen percent were Subtree shape – hierarchies with multiple branch points - but, because of their size, these contained 63% of all classes defined using inheritance.

Nasseri et al. [12] studied the evolution of inheritance hierarchies using seven open source systems taken from SourceForge. They studied 156 versions in total. Their focus was how evolution affected the Depth of Inheritance (DIT) metric [2]. Their main finding was that 96% of classes added during evolution were at DIT level 1 or level 2. This had the tendency to increase the shallow breadth of hierarchies through time. In related research [13], Nasseri et al. found that most of the inheritance changes were again in shallow areas of the hierarchies (level three or shallower) and that many of the changes led to a ‘squashing’ of the hierarchies.

In a survey on design quality with industry practitioners [16], Stevenson and Wood found a mixed response in terms of the value of inheritance. Specific comments on inheritance usage included: “*avoid ... it always ends up biting me*”, “*you don’t want your ears to pop when traversing down the inheritance hierarchy*”, and “*derived types must satisfy the Liskov Substitution Principle ... very difficult to achieve, so we try to use composition*”.

## 3 STUDY DESIGN

### 3.1 Research Objectives

The high-level goal of this research is to improve the guidance relating to object-oriented inheritance hierarchy design. This is done by analysing how inheritance hierarchies develop, particularly complex, multi-branching hierarchies and then trying to understand their design qualities. This is achieved by studying the evolution of 665 hierarchies across a total of 262 versions from 10 open source Java systems.

The research questions are:

1) How do object-oriented inheritance hierarchies evolve in terms of their size and shape across many versions of a software system?
2) How do complex, multi-branch, ‘Subtree’ hierarchies evolve across many versions of a software system?
3) What are the design qualities of complex, multi-branching, ‘Subtree’ inheritance hierarchies?

### 3.2 Study Corpus

Ten open source Java systems were selected from the Qualitas Corpus evolution distribution [19]. The systems in the evolution package have a development history consisting of at least ten versions. The systems chosen covered a range of problem domains, development histories and sizes – see Table 1. The second column lists the number of versions analysed for each system (at least 16). The third column of Table 1 lists the total number of hierarchies found across all versions of each system. The fourth column is the number of classes in the final version of the system. The fifth column indicates the system problem domain.

**Table 1: Summary of Analysed Systems**

| Projects | No. of Versions | No. of Hierarchies | No. of Classes | Domain |
|---|---|---|---|---|
| Ant | 23 | 58 | 1290 | Build Tool |
| Antlr | 22 | 72 | 385 | Parser Generator |
| ArgoUML | 16 | 117 | 2560 | UML Diagramming |
| FreeCol | 32 | 27 | 1310 | Colonisation Game |
| FreeMind | 16 | 27 | 50198 | Mind Mapping |
| JGraph | 39 | 23 | 187 | Graph Drawing |
| JMeter | 24 | 36 | 1143 | Web App Testing |
| JStock | 31 | 17 | 867 | Stock Management |
| JUNG | 23 | 59 | 858 | Data Modelling |
| Lucene | 36 | 229 | 3729 | Search Engine |
| Total | 262 | 665 | 62527 | |

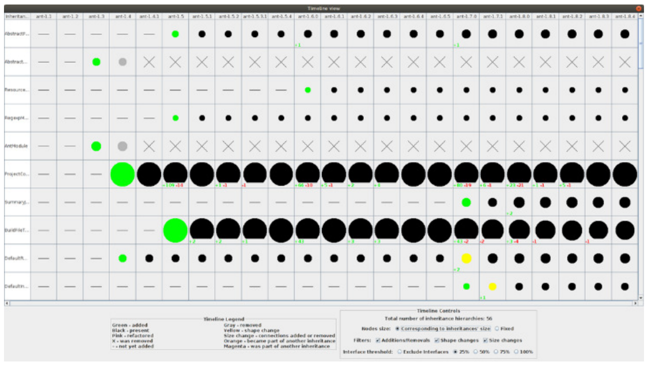
The number of systems, number of versions and the range of domains analysed are comparable to related work [12, 13, 17]. The range of system sizes is in keeping with system sizes found by Radjenović et al. in a review of code-survey research - where less than 200 classes was categorised as a small system, 200-1000 classes medium sized, and 1000 or greater as large [14].

## 3.3 Study Instrumentation

Analysis was performed using a purpose-built tool based on the Eclipse JDT Core. The tool provides a high-level, graphical summary of hierarchy evolution and supports drilling down into the details of individual hierarchies. While this tool is novel, the core components are very reliable as they are sourced from the Eclipse Project. The tool identifies the following properties of all the inheritance hierarchies in a system and then tracks the changes in hierarchy properties across multiple versions of the system:

- Size - the number of classes in a particular hierarchy.
- Depth and breadth of hierarchies.
- Age of a hierarchy – how many versions a hierarchy is present in.
- Shape – the 'shape' of hierarchy as defined by Stevenson and Wood [17] – 'Fan', 'Line, 'Subtree', 'Line-Branch or 'Branch-Line'.
- Changed / Stable – a hierarchy is 'stable' if there are no changes in shape or size between successive versions, otherwise it is categorized as 'changed'.

Analysis is based on the collection of all versions in the evolution corpus for a system – the tool is given the root directory that contains all versions. The tool has two phases, extraction of all inheritance data and then visualisation of the data. This means that systems can easily be visualised without the need for time-consuming re-analysis (phase one).


Timeline Legend
Green - added
Black - present
Pink - refactored
X - was removed
- - not yet added
Gray - removed
Yellow - shape change
Size change - connections added or removed
Orange - became part of another inheritance
Magenta - was part of another inheritance

Timeline Controls
Total number of inheritance hierarchies: 56
Nodes size: [✓ Corresponding to inheritances' size] Fixed
Filters: [✓ Additions/Removals] [✓ Shape changes] [✓ Size changes]
Interface threshold: Exclude interfaces [✓ 25%] 50% 75% 100%

Figure 1: Top-Level GUI Summarising Hierarchy Evolution

The visualisation shows a wide range of hierarchy properties in its top-level graphical interface – see Figure 1. There is a separate row for each hierarchy in the system – the root classes are in the leftmost column. Each column shows a version of the system – the version numbers are in the top-most rows. Each cell summarises the changes in properties of a hierarchy relatively to the previous version. A dash ('-') means the hierarchy wasn't previously present. A cross ('X') means that the hierarchy was previously present but is no longer present. Circles represent hierarchies. The size of a circle is an indication of its relative size. A green circle indicates the first appearance of the hierarchy, grey denotes that the hierarchy is no longer present, black represents no major change. A yellow circle indicates a change in hierarchy 'shape' – if you hover the mouse it shows the old and new shapes. Orange indicates a hierarchy was previously independent but has now been integrated into another hierarchy. Purple indicates a hierarchy was previously within another hierarchy but is now independent. Other properties highlighted in this top-level view include the number of classes added and changed in any step. There is also a variety of controls to change nodes sizes and to filter the view.

The evolution of two hierarchies Token (top row) and Event (bottom row) can be seen in Figure 2. Event does not have a single change between its addition and removal 12 versions later. Therefore, Event is stable between any two versions. Token, however, has changed twice - between the 2nd and the 3rd versions it has acquired an extra class (there is a green '1' in the bottom left of the 4th cell); between the 6th and the 7th versions it has also acquired an extra class, which has also caused a shape change.

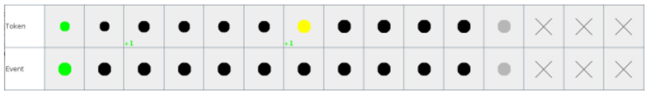


Figure 2: Evolution of Two Hierarchies

The tool supports 'drilling down' into any hierarchy. Selecting a node will show a visualisation of that hierarchy – see Figure 3. This view enables detailed investigation of hierarchies, it shows the number of nodes, shape, depth and largest breadth. Edge colour-coding indicates added and deleted edges, as well as inheritance and interface implementation. Node colour-coding indicates root of inheritance (yellow), a root in the previous version (green), a node was previously in another hierarchy (orange), interfaces (red) and concrete classes (black). The visualisation of interfaces can be switched on and off – see later discussion.

To analyse the details of specific hierarchies the Eclipse IDE was used. Specific versions were opened and Eclipse commands such as 'Open Type Hierarchy' and 'Java Search – Type – References' were used to understand the hierarchy properties.

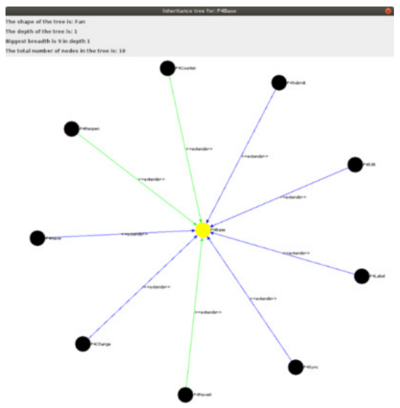

Inheritance tree for: P4Base
The shape of the tree is: Fan
The depth of the tree is: 1
Biggest breadth is 9 in depth 1
The total number of nodes in the tree is: 10

Figure 3: Visual Representation of Individual Hierarchy

## 4 RESULTS

### 4.1 Change in Size

The first analysis is the change of hierarchy sizes as the systems evolved. Table 2 summarises these results.

**Table 2: Change in Size During Evolution**

| Projects | Fixed | Stable | Unstable |
|---|---|---|---|
| Ant | 34 (59%) | 22 (38%) | 2 (3%) |
| Antlr | 56 (78%) | 14 (19%) | 2 (3%) |
| ArgoUML | 74 (63%) | 33 (28%) | 10 (9%) |
| FreeCol | 17 (63%) | 10 (37%) | 0 (0%) |
| FreeMind | 16 (59%) | 10 (37%) | 1 (4%) |
| JGraph | 15 (65%) | 8 (35%) | 0 (0%) |
| JMeter | 22 (61%) | 14 (39%) | 0 (0%) |
| JStock | 13 (76%) | 4 (24%) | 0 (0%) |
| JUNG | 45 (76%) | 9 (15%) | 5 (8%) |
| Lucene | 153 (67%) | 61 (27%) | 15 (7%) |
| Total: | 445 | 185 | 35 |
| Average | 67% | 28% | 5% |

'Fixed' means that there was no change in the hierarchy size (number of nodes) throughout its lifespan. The definition of 'Stable' is that the hierarchy changed size in 10% or less of system versions. 'Unstable' indicates a hierarchy changes size in more than 10% of versions. These results clearly show that the large majority of hierarchies had no or very little change in size throughout their history. Typically, only 5% of hierarchies change their size in more than 10% of versions. Around 67% of hierarchies do not change their size at all – they are fixed size. Only the three largest systems (Lucene, ArgoUML and JUNG), had more than two hierarchies that regularly changed size during their lifespan.

### 4.2 Hierarchy Age

The next analysis was of hierarchy 'age' – what percentage of a system lifetime was a hierarchy present for – see Table 3. 'Newborn' means that the hierarchy was present in less than 20% of the system versions, 'Young' means that the hierarchy was present between 20% and 50% of the system versions, 'Old' means present between 50% and 80%, and 'Persistent' means greater than 80% of versions. This is a similar approach to Gîrba et al. who used 10%/50%/90% as hierarchy age boundaries [6].

These results suggest that the majority of hierarchies in these systems have quite a short lifespan – though there is considerable variation within individual systems. Considering the size analysis in the section above, although hierarchies appear relatively stable, that stability is often across a shorter lifespan than the whole system lifespan.

In seven of the systems 'Newborn' hierarchies dominate at around 40%. The two main exceptions are JMeter and JStock. Possible explanations for this are that both of these are relatively small projects. Also, both of these systems seem relatively stable in general, with little refactoring. JStock has by far the largest number of 'persistent' hierarchies at 76%.

**Table 3: Hierarchy Age Profiles**

| Projects | Newborn | Young | Old | Persistent |
|---|---|---|---|---|
| Ant | 14 (24%) | 12 (21%) | 19 (33%) | 13 (22%) |
| Antlr | 28 (39%) | 29 (40%) | 15 (21%) | 0 (0%) |
| ArgoUML | 43 (37%) | 14 (12%) | 23 (20%) | 37 (32%) |
| FreeCol | 9 (33%) | 9 (33%) | 5 (19%) | 4 (15%) |
| FreeMind | 19 (70%) | 1 (4%) | 2 (7%) | 5 (19%) |
| JGraph | 12 (52%) | 6 (26%) | 1 (4%) | 4 (17%) |
| JMeter | 4 (11%) | 14 (39%) | 6 (17%) | 12 (33%) |
| JStock | 0 (0%) | 3 (18%) | 1 (6%) | 13 (76%) |
| JUNG | 19 (32%) | 17 (29%) | 13 (22%) | 10 (17%) |
| Lucene | 119 (52%) | 44 (19%) | 47 (21%) | 19 (8%) |
| Total | 267 | 149 | 132 | 117 |
| Average | 40% | 22% | 20% | 18% |

Figure 4 shows an interesting age analyses from JUNG with some persistent, growing hierarchies but a lot of apparent change with hierarchies seeming to disappear and new ones appearing.

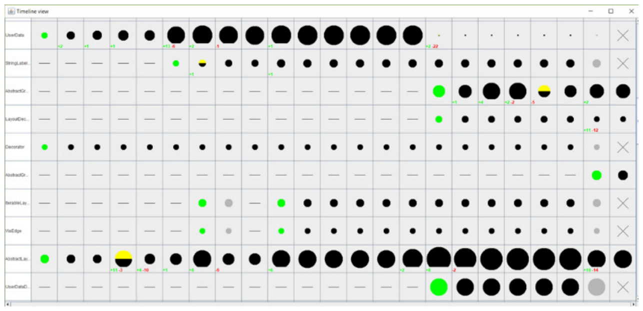


**Figure 4: JUNG – A Range of Different Age Categories**

Throughout this work there was always a question of how to deal with Java interfaces. For most of the analyses it seemed clear that interfaces should not be included – in size and shape analyses. However, there was some concern that the true, stable root of a hierarchy could be an interface – following 'Program to an Interface not an Implementation' [5] – and hierarchies might appear to come and go but, actually, the interface remained stable. It is also possible that hierarchies might merge under a single interface.

To investigate this, analysis was performed that explored whether there was a difference if interfaces were treated as hierarchy roots. The analysis determined which classes/interfaces were used by the rest of the system to access the hierarchy. If the hierarchy was accessed most via an interface then that was considered the hierarchy 'root'. Across most of the analysed systems this made very little difference – the profiles were almost identical to Table 3. As expected, the total number of hierarchies identified occasionally varied slightly (by a few only).

JUNG was one system where there was a difference – the number of 'newborn' hierarchies increased from 43 to 47. Also, without interfaces, Antlr is the only project with no 'Persistent' hierarchies (no hierarchy appears present in more than half the versions). With interfaces, there is a hierarchy with the root TokenStream that is present in all analysed versions.

## 4.3 Hierarchy 'Shape'

Much of the prior work has found that most inheritance hierarchies are small and simple [3, 17, 20]. In a one-off, snapshot of open source systems, Stevenson and Wood [17] found that hierarchies were dominated by 'Line' and 'Fan' shapes. There were only a small number of more complex 'Subtree' hierarchies, around 15%. This analysis focused on where these hierarchies come from – are they in the design from the start, or do they evolve? Are there any insights from evolution that could be used to avoid such complex hierarchies during system development?

The first analysis repeated the work of Stevenson and Wood and categorised the hierarchies across all versions according to their shape. The same definitions of shape were used:

- Line: Maximum breadth of the hierarchy = 1.
- Branch: Maximum depth of hierarchy = 1 (root is 0).
- Line-Branch: Root has one child which has more than one child. All child branches are breadth = 1.
- Branch-Line: Root has more than one child. All child branches are breadth = 1.
- Subtree: All other hierarchies. They have multiple branch points.

**Table 4: Shape of Inheritance Hierarchies**

| Projects | Branch-Line | Line | Fan | Line-Branch | Subtrees |
|---|---|---|---|---|---|
| Ant | 4 (7%) | 22 (38%) | 26 (45%) | 0 (0%) | 6 (10%) |
| Antlr | 3 (4%) | 37 (51%) | 22 (31%) | 2 (3%) | 8 (11%) |
| ArgoUML | 4 (3%) | 47 (40%) | 44 (38%) | 4 (3%) | 18 (15%) |
| FreeCol | 0 (0%) | 8 (30%) | 11 (41%) | 1 (4%) | 7 (26%) |
| FreeMind | 1 (4%) | 5 (19%) | 17 (63%) | 0 (0%) | 4 (15%) |
| JGraph | 3 (13%) | 15 (65%) | 5 (22%) | 0 (0%) | 0 (0%) |
| JMeter | 2 (6%) | 17 (47%) | 11 (31%) | 0 (0%) | 6 (17%) |
| JStock | 0 (0%) | 2 (12%) | 11 (65%) | 3 (18%) | 1 (6%) |
| JUNG | 1 (2%) | 25 (42%) | 23 (39%) | 0 (0%) | 10 (17%) |
| Lucene | 17 (7%) | 90 (39%) | 89 (39%) | 3 (1%) | 30 (13%) |
| Total: | 35 | 268 | 259 | 13 | 90 |
| Average | 5% | 40% | 39% | 2% | 14% |

Table 4 shows results in keeping with the previous work with Line and Fan again dominating and Subtree making up typically 14% of all hierarchies. It should be noted that there are four systems in common in this study with the earlier work of Stevenson and Wood (Ant, ArgUML, Freecol and FreeMind) – though they only looked at a single version of each of these systems. If these systems are removed from the analysis, then there is only an average of 11% Subtrees. The previous study investigated 2440 hierarchies and found that 15% were Subtrees. A key point in the work of Stevenson and Wood was that, due to their size, these 15% of hierarchies contained 63% of all hierarchy members.

## 4.4 Stability of Shape

The next analysis examines the extent to which hierarchies maintained their shape category – see Table 5. Hierarchies are categorised as 'Fixed' or 'Changed' depending on whether they change shape or not.

**Table 5: Shape Stability**

| Projects | Fixed | Changed |
|---|---|---|
| Ant | 43 (74%) | 15 (26%) |
| Antlr | 64 (89%) | 8 (11%) |
| ArgoUML | 102 (87%) | 15 (13%) |
| FreeCol | 25 (93%) | 2 (7%) |
| FreeMind | 23 (85%) | 4 (15%) |
| JGraph | 19 (83%) | 4 (17%) |
| JMeter | 31 (86%) | 4 (14%) |
| JStock | 17 (100%) | 0 (0%) |
| JUNG | 51 (86%) | 8 (14%) |
| Lucene | 198 (86%) | 31 (14%) |
| Total: | 573 | 92 |
| Average | 86% | 14% |

The vast majority of hierarchies (86%) do not change shape during system evolution. Most of the changes in shape again appear relatively simple e.g. there were 36 changes from Line to Fan, 24 changes from Fan to Branch-Line and 18 changes from Fan to Line. Across all the changes, there were a total of 34 transitions into Subtrees. The most likely change for a Subtree is for it to become a different, potentially more complex Subtree (see later discussion).

One interesting hierarchy is AbstractLayout from JUNG that implements the core JUNG Layout interface using a variety of graph layout algorithms. This is an example of an interface sitting at the root of a hierarchy, with an abstract class directly underneath. Figure 5 shows the high-level summary of this hierarchy's changes. The yellow circles indicate shape change, the change in circle size indicates hierarchy size change, the green and red numbers indicate classes added (green) or removed (red).

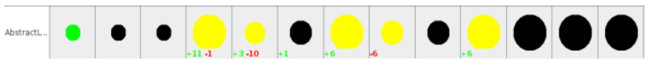


**Figure 5: AbstractLayout from JUNG Evolution Summary**

The initial shape of AbstractLayout was a Fan. There are three concrete variations of Layout subclasses. Three versions later ten new classes are added to the hierarchy creating a Subtree shape. In terms of design quality it is at stages such as this that the designer should look very closely at the overall design of the hierarchy. One subclass, SpringLayout, has changed its parent connection to extend one of the newly added classes IterableLayout. Oddly, the hierarchy then goes through a series of four shape changes removing and adding these same classes before finishing as a Subtree hierarchy.

In terms of design quality, the Layout classes are all implementing versions of (sophisticated) graph layout e.g. CircleLayout and SpringLayout, and do appear to implement variations of a single abstraction. However, different subclasses also add methods to the Layout interface e.g. FRLayout (Fruchterman-Reingold) adds methods for the attraction and repulsion of nodes. To access these methods types must be declared as FRLayout or cast from Layout to FRLayout - which is what happens in one of the JUNG sample programs.

## 5 DETAILED ANLAYSIS OF SUBTREES

In terms of inheritance design quality, Subtrees seem important hierarchies – these are the relatively small number of complex hierarchies that contain most classes. This section provides a detailed analysis of some interesting Subtree hierarchies from four systems – Ant, ArgoUML, FreeCol and JMeter. It describes how they evolve and their key design characteristics, including how they are accessed by the rest of the system.

### 5.1 Subtree Analysis – Ant

Apache Ant is a free, open source Java automated build system that uses XML to describe the build process and its dependencies. Ant manages a range of build tasks such as compiling, testing and deployment, and uses a range of data types such as files and paths.

Six Subtrees were found across the history of Ant. Three of these only survived for one version. There are two particularly interesting hierarchies, Task and ProjectComponent. Figure 6 shows Task when it is introduced in the first version of the system – it already has 50 classes.

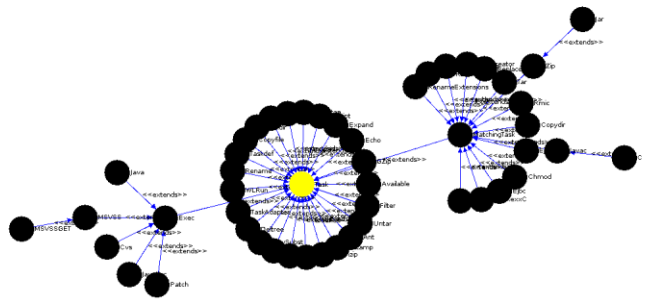


**Figure 6: The Task Hierarchy from Ant**

During the next two versions it continues to grow, 22 classes are added then another 18, making it a 90-class hierarchy. In the fourth version it is subsumed into the ProjectComponent hierarchy – see Figure 7.

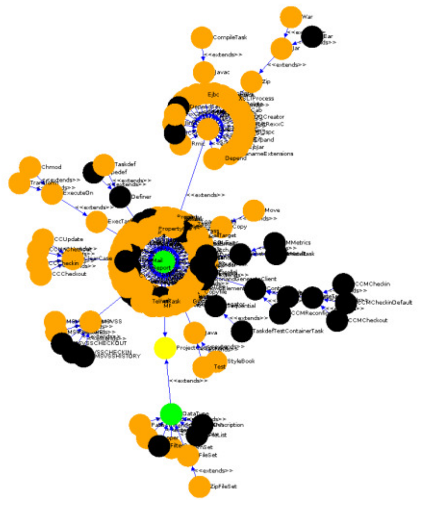


**Figure 7: The ProjectComponent Hierarchy from Ant**

The yellow node in Figure 7 is the root ProjectComponent. The green node above the root is the former root Task. The orange nodes represent classes that were present in other hierarchies before this version. ProjectComponent has not only sucked in the root of Task but also all of its subclasses. There is also another green node visible along with a few orange ones around it – the hierarchy DataType. At the time that ProjectComponent was added to Ant, its size is 135 classes. The majority of them come from Task. Thereafter, ProjectComponent keeps growing. Only two versions after its introduction, nearly 100 new classes are added and again it sucks in another independent hierarchy. Right until the last version, new additions happen more often than not, resulting in ProjectComponent having a total of 367 classes.

ProjectComponent is an abstract class, the class comment says "*Base class for components of a project, including tasks and data types. Provides common facilities*". It only defines a small number of methods to do with location, logging and projects. It has over 30 immediate subclasses, including Task and DataType.

A search of Ant version 1.8.4 within the Eclipse IDE finds 131 references to the ProjectComponent type. Many of these uses are in defining the ProjectComponent subtypes. It appears that Ant passes around many objects as type Object and then uses type checking (instanceof) and casting to convert to ProjectComponent and its subtypes. In keeping with the class comment, objects are cast to ProjectComponent for 'logging' in the contexts where logging functionality is required.

Task is an abstract subclass of ProjectComponent, it has over 100 subclasses of its own and Eclipse shows 451 references to the Task type. Many of these are also used in the definition of subclasses. The comment describes it as the base class for all Ant tasks. The definition adds new methods for tasks such as 'execute' and 'perform'. Task appears to follow a similar design model as ProjectComponent where Ant objects are type checked and then cast to be used in the Task context.

Another major subclass of ProjectComponent is the abstract class DataType. It is the "*base class for those classes that can appear inside the build file as stand alone data types*". The class has its own methods for managing Ant data types. DataType itself has 32 subclasses and is referenced 92 times in the source code. Its usage follows a similar design model where general Objects are type checked and cast within the DataType context.

Continuing down the inheritance hierarchy below DataType uncovers more specific types such as Path for managing Ant environment variable paths. Again, it adds many methods, has hundreds of references and follows the same model of usage.

The hierarchy DirectoryScanner in Ant is an example of a hierarchy that changes from a Fan shape to a Subtree shape. The hierarchy implements the FileScanner interface for scanning any type of directory. DirectoryScanner is an abstract implementation of this. It changes to a Subtree shape with the addition of the abstract subclass ArchiveScanner which has two concrete subclasses ZipScanner and TarScanner. While the hierarchy seems a good example of variations of a FileScanner abstraction, subclasses do add additional methods e.g. ArchiveScanner adds setEncoding which is then accessed via casting of the parent DirectoryScanner type.

## 5.2 Subtree Analysis – ArgoUML

ArgoUML is a widely used, Java open source CASE tool that supports UML modelling and diagramming. The design makes use of well-established design practices such as design patterns (Facade, Strategy, Factory Method, Observer …), Model-View-Controller and Programming to Interfaces. Some of the main design elements include Diagrams, Figures, Notations (for code generation), and GUI components [21].

ArgoUML is one of the larger systems that were analysed and had 18 hierarchies that were of Subtree shape when they were first introduced. Half of these hierarchies remained 'Fixed' in size with no change at all or were 'Stable'.

Critic is a key concept in ArgoUML – used to 'critique' the design. The Critic hierarchy is a major ArgoUML Subtree. It first appears as a 91-class hierarchy. In the following version a further 11 classes are added and in the final version it contains 105 classes – see Figure 8.

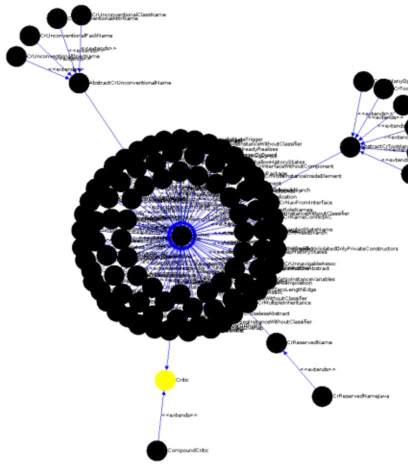


**Figure 8: The Final Version of Critic from ArgoUML**

Although Critic is a concrete class, code comments describe it as "*abstract*", and it is used as a static Singleton (code comments suggest instances shouldn't be created at all). Subclasses have to define a 'predicate' method associated with Critic-specific design checks. The Critic class defines many methods for managing general properties such as type, description and priority.

A search for the Critic type in version 0.34 of ArgoUML finds 287 type references. Critic has two direct subclasses – CompoundCritic (bottom node in Figure 8) and CrUML, which has around 90 direct subclasses (the central blue-tinged node in Figure 8). Although concrete, the design documentation again describes CrUML as "*abstract*", it only adds a few UML OCL properties and methods. Analysis of these subclasses shows strong evidence that Critic is a 'type inheritance' hierarchy rather than 'reuse' focused. Almost all of these subclasses override just a few methods in the Critic interface – typically the 'predicate' method and a method such as 'getCriticizedDesignMaterial'. Quite a few of the subclasses do add a public method to the interface 'computeOffenders' – however it appears that this method is always used as a local 'private' method within the subclass. A check for the casting of these subclasses finds very little, typically only in the GUI where it is required to know the specific type of a design critic.

There is a significant use of the Java Object class in ArgoUML (7580 type references found) and over one thousand uses of the instanceof method. One of the key uses of instanceof is in the Critic predicate method where the design element to be checked is passed as an Object type and then type checked.

## 5.3 Subtree Analysis – FreeCol

FreeCol is a Java open source colonisation game with aim of building a 'powerful nation'. Key concepts in the game include Players, Nations, Colonies, Trade, Buildings, Settlements and Maps.

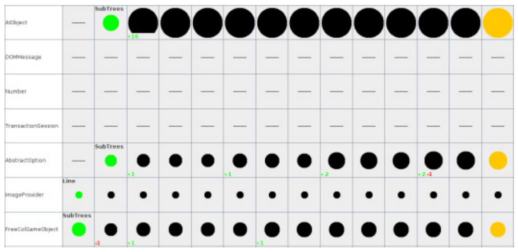


**Figure 9: Subtree Evolution in FreeCol**

Analysis of Subtrees in FreeCol reveals an interesting story partly shown by the lifespan view in Figure 9. It shows the evolution of three Subtrees – AIObject (top row), AbstractOption (3rd bottom row), and FreeColGameObject (bottom) row. These hierarchies are quite complex Subtrees all of which appear fairly stable throughout their lifespan. Something interesting occurs in the final column – the three of them turn orange – they are incorporated into another hierarchy under the abstract class FreeColObject.

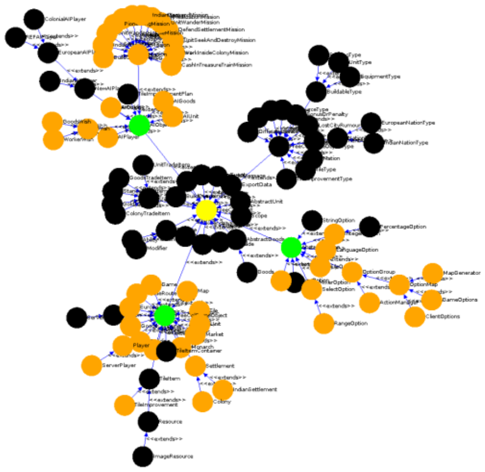


**Figure 10: The Initial Version of FreeColObject**

Figure 10 shows the initial the version of FreeColObject that integrates these three hierarchies. The three green nodes represent the three previous hierarchies – AIObject (top left), AbstractOption (bottom right) and FreeColGameObject (bottom left). They are all merged into this 108-class hierarchy, with roughly half of the classes coming from these three hierarchies (the orange nodes). The final version of FreeColObject contains 148 classes in total.

Analysis of the FreeCol source code version 0.10.7 shows a typical reuse focused use of inheritance in the FreeColObject hierarchy. It appears that FreeColObject is used to pass around subclasses – specifically AIObjects, FreeColGameObjects and AbstractOptions – largely to the GUI and for input/output. It only provides very general properties – the class comment says "*The FreeCol root class. Maintains an identifier and an optional link to the specification this object uses*". Searching for references to the class finds 206 uses. A portion of these references uses FreeColObject class methods and public static types. Objects are passed around as a collection (array) of FreeColObjects. A large number of uses involve type checks (instanceof) and casting to a subtype, even a subtype three levels down the hierarchy such as BuildingType.

A similar pattern is seen in its subclasses, for example FreeColGameObject. This class is the *"... superclass for all game objects in FreeCol*". It covers a large variety of game object subclasses such as Game, Player, Market and GoodsContainer. Again, the methods are very general such as managing resources, ids and XML representation. A search finds 411 uses and again the large majority appear to involve type checking and casting. There are some high level uses such as reading from an XML representation. The casts are to subtypes such as Unit (three levels down the hierarchy) and Settlement (also three levels down).

Moving down these hierarchies, many methods are added to the subclasses. For example, a subclass of FreeColGameObject is UnitLocation, which adds 'locations' to 'units' and implements the Location interface. This adds numerous methods for the manipulation of unit locations. Below that is a GoodsLocation class for managing locations where Goods and Units can be placed, again adding many methods. Below that is Settlement – "*The superclass of all Settlements*" – adding over 50 methods. The addition of such methods is a clear sign of a reuse-oriented hierarchy that is relying on type checking and casting to access this lower level functionality. There are two further levels of inheritance below Settlement.

## 5.4 Subtree Analysis – JMeter

JMeter is a free, open source, Java system that supports the performance testing of web applications [7]. Logical Controllers let you customize the logic that JMeter uses to decide when to send requests. Samplers tell JMeter to send requests to a server and wait for a response.

There are two major Subtree hierarchies in JMeter that are ever present - AbstractJMeterGuiComponent and AbstractTestElement. They start at size 55 and 57 classes respectively and both more than double in size during JMeter lifetime. The final shape of AbstractTestElement is shown in Figure 11.

The class AbstractTestElement is the abstract implementation of the key JMeter TestElement interface. TestElements are the components that can be tested in JMeter. There are over 100 classes that implement that interface and Eclipse finds 427 references to its use in JMeter version 2.9. It appears that much of the design of JMeter is written in terms of this core interface. TestElement contains over 40 methods associated with high-level features such as properties and names.

There are only 42 references to AbstractTestElement and many of these are used in the type definitions of its 51 subclasses. These are more concrete JMeter TestElements such as Controllers and Samplers.

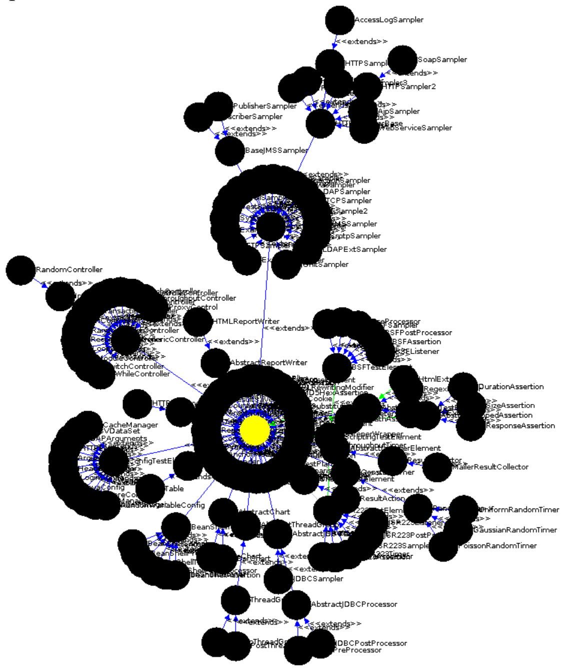


**Figure 11: JMeter AbstractTestElement**

There is again significant use of type checking and casting in JMeter, with 279 uses of instanceof found. As well as use of the TestElement interface JMeter also passes objects around using the Java Object class.

There is evidence that JMeter is using the Factory Method design pattern [5] to install objects that implement the TestElement interface. Many of the implementations of TestElement include a createTestElement method that returns a concrete implementation behind the TestElement interface.

Moving down the hierarchy, key classes associated with specific types of TestElement, such as Controllers, are added with their own type-specific methods. The root of the controller sub-hierarchy GenericController, "*... the basis of all the controllers*", adds six methods from the Controller interface. It has 23 subclasses of its own. Similarly, the AbstractSampler subclass of TestElement implements the single-method Sampler interface and has 20 subclasses of its own. This pattern of subclass hierarchy clusters can be seen in Figure 11 outside the core circle. There are appears to be a relatively small amount of type checking and casting of TestElement objects within JMeter into these more concrete subtypes. It does therefore appear, in the main, that the hierarchy is emphasising type inheritance over reuse.

## 6 ANSWERS TO RESEARCH QUESTIONS

### 6.1 How do Inheritance Hierarchies Evolve in Terms of their Size and Shape?

The results suggest that there was limited change in size across the system histories. Only 5% of hierarchies change their size in more than 10% of versions and around 67% of hierarchies were fixed size. This could be related to system size, it was the three larger systems (Lucene, ArgoUML and JUNG) that contained hierarchies that regularly changed size during their lifespan. As discussed, a small number of larger Subtree hierarchies do change size considerably.

Many hierarchies appear to have a relatively short lifespan – only 18% of hierarchies seemed to be in more than 80% of versions, with 40% appearing in less than 20% of versions. However, a deeper analysis is required to discover what is really going on here. Interfaces need to be considered, often they are the true root of hierarchies. Also, when hierarchies disappear they are often being merged into other hierarchies, perhaps under an interface.

In keeping with findings from previous work [3, 17, 20], the large majority of hierarchies found were very simple 'Line' and 'Fan' shape, around 80%. These hierarchies are so simple, depth or breadth one, that they shouldn't cause major design challenges. Again, in keeping with previous work [17], it was found that around 14% of hierarchies across all versions were the more complex, multi-branching Subtree shape.

The vast majority of hierarchies (86%) do not change shape during system evolution. Most of the changes in shape again appear relatively simple, staying away from the Subtree shape. It is the changes within the Subtree category that may be most interesting.

### 6.2 How do Complex, Multi-branch Subtree Hierarchies Evolve?

Out of the 665 hierarchies, 90 were classified as Subtree when they were first created. A further 34 Subtrees appeared during evolution from simpler hierarchy shapes. Again, it was the larger systems that contained more Subtrees (ArgoUML and JUNG). Previous work [17] found more hierarchies and more Subtrees in larger systems such as Eclipse, but still the same approximate percentage of Subtrees overall (18% in Eclipse).

The detailed Subtree discussions in Sections 5.1 to 5.4 show that many of the Subtrees tend to grow ever more complicated during their lifespan e.g. the ProjectComponent hierarchy in Ant growing from 135 classes to 360 and the TestElement hierarchy in JMeter growing from 57 to 159 classes.

### 6.3 What are the Design Qualities of Subtree Inheritance Hierarchies?

The detailed analysis of Subtree hierarchies in Sections 5.1 to 5.4 provide a range of design quality insights. Some of the Subtree hierarchies do appear designed consistently with type inheritance (LSP) as the aim whereas others are reuse focused hierarchies. It does appear possible to distinguish between these two key design motivations.

The Critic Subtree from ArgoUML is a good example of type inheritance. Subclasses (subtypes) appear consistent with the single Critic abstraction, they add few, if any, public methods to the root abstraction, they are accessed via the root interface and there is very little casting to specific subtypes. The variation in behaviour is achieved by overriding a small number of common methods. The FileScanner hierarchy from Ant and Layout from JUNG are other good examples. The TestElement hierarchy from JMeter also seems to emphasise type inheritance though subclasses do include additional Java interface implementations.

On the other hand, other major Subtree hierarchies exhibit reuse characteristics. Hierarchies such as Ant's ProjectComponent and FreeCol's FreeColObject are very general system hierarchies that mix abstractions amongst their subhierarchies. Subclasses have significantly different interfaces and add many new public methods. Type checking and casting are used to access these.

Another key finding is the sheer complexity of some of these hierarchies, regardless of whether they are type substitution or reuse focused. Understanding a 100-plus class hierarchy is a daunting task. Studying the 51 subclasses of AbstractTestElement in JMeter it is difficult to see where to start if adding a new class.

It is even harder to start to think about refactoring such a hierarchy; they have so many internal and external dependencies. Challenges include understanding what is inherited, what and where it is overridden, what interfaces are being implemented, how the different types are used in the rest of the system, when and where they are used as a general type, and when and where they are used as a specific type. It is easy to see why practitioners are so wary of inheritance [16]. It is therefore vital to consider design alternatives when first introducing these complex hierarchies.

In all the systems there were clear signs of well-established design practices being adopted. Most of the systems were using Java interfaces and layering the hierarchy designs from interfaces, through abstract classes to concrete classes e.g. JUNG Layout. There were clear signs of design pattern [5] usage, especially Factory Methods to install concrete subtypes behind an interface.

The widespread use of the Java Object class was surprising. Three out of the four systems studied in detail made use of Object to pass a variety of types around the systems. In tandem with this practice, was a widespread use of type checking and casting to convert either the Object type to a more concrete system type or one of the more general system-specific types to a subtype.

## 7 THREATS TO VALIDITY

Using open source systems as a proxy for real-world development is a threat to the validity of this work. Given the difficulty of analysing propriety source code, open source is often used as a substitute for closed source software. Open-source systems may not be subject to the same design and review practices associated with commercial software.

There are also validity threats in the selection of the corpus used in this study. The choice was somewhat limited by the availability of systems with a history of evolution. Care was taken to select a range of system sizes and problem domains. It is argued that the corpus shares similarities with corpora used in comparable studies. Overall, the systems studied here are smaller than many 'industrial strength' systems. Evidence suggests that larger systems are likely to have more hierarchies and more complex hierarchies, but possibly having a similar ratio of 'simple' to 'complex'.

The number of systems analysed and, in particular, the number of versions analysed, compares well to previous evolution studies. On the other hand, ten systems is quite small compared to previous census-style research. A strength of this work is the detailed analysis of numerous Subtree hierarchies. Extracting and analysing the source code for each individual system was a significant effort.

The use of a purpose-built tool is also a threat. Confidence is gained from the use of the well-regarded, widely used, Eclipse JDT framework. It was also reassuring to find that the in-depth code analysis produced findings that were consistent with the high-level results produced by the tool.

## 8 CONCLUSIONS

This paper contributes an understanding of how inheritance hierarchies evolve in object-oriented systems. The paper's main contribution is new insights into how complex, multi-branching, 'Subtree' hierarchies evolve and a detailed analysis of their design qualities. The paper confirms previous findings that, in practice, a large majority of hierarchies are simple in structure, only about 15% are more complex. It finds that a large majority of hierarchies are stable in terms of both size and shape. On the other hand, the average lifespan of most hierarchies appears to be relatively short, though it seems this may be because they are merged into new hierarchies. The work also identifies a challenge in terms of how to define and track hierarchies through multiple versions – is the stable root an interface, an abstract/concrete class, or is a more inclusive definition involving the whole hierarchy required?

The work confirms that the majority of hierarchies found in practice are simple – either depth or breadth of one. A detailed analysis of the remaining Subtree hierarchies finds that some are clearly designed with type inheritance as a goal whereas others have a more general reuse focus. Type hierarchies appear to implement a single abstraction, add little or no methods to the root interface, and are involved in little or no type checking and casting. Reuse focused hierarchies tend to have a very general abstraction at the root, have multiple, often quite distinct abstractions within the hierarchy, add new methods to their subclasses, and use type checking and casting to access objects defined by these classes.

Regardless of fundamental design motivation, it is clear that these large complex hierarchies are challenging to understand and maintain. In hierarchies with hundreds of subclasses, it is hard to determine where to add a new class that is consistent with the original hierarchy design. It is difficult to understand classes in this inheritance context and challenging to understand how the hierarchy interacts with the rest of the system. It seems clear, however, that type hierarchies make this task easier than multi-abstraction, reuse hierarchies.

The detailed analysis also found evidence of well-regarded design practices such as programming to interfaces, use of abstract classes and use of design patterns. On the other hand, many of the systems made significant use of the Java Object type, along with type checking and casting to convert to context-specific types.

It would be valuable for future work to look more closely at Subtree hierarchies and perform a detailed comparison against their design alternatives. What are the relative strengths of separate smaller hierarchies or alternatives based on interfaces and object composition – preferring object composition over class inheritance [5]? Is it possible to demonstrate superior design alternatives? To what extent are design choices system or context dependent? Finally, it is clear that developers should think carefully when introducing complex Subtree hierarchies into their designs – they are going to be difficult to remove or redesign thereafter.

## ACKNOWLEDGEMENTS

The authors are grateful for funding from an EPSRC Vacation Bursary and the Research Interns @ Strathclyde Programme.

## REFERENCES

[1] J. Bloch. 2008. *Effective Java*. Pearson Education India.

[2] S. R. Chidamber and C. F. Kemerer. 1994. *A Metrics Suite for Object Oriented Design*. IEEE Transactions on software engineering. **20**(6): p. 476-493.

[3] C. Collberg, G. Myles, and M. Stepp. 2007. *An Empirical Study of Java Bytecode Programs*. Software: Practice and Experience. **37**(6): p. 581-641.

[4] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood. 1996. *Evaluating Inheritance Depth on the Maintainability of Object-Oriented Software*. Empirical Software Engineering. **1**(2): p. 109-132.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education.

[6] T. Gîrba, M. Lanza, and S. Ducasse. 2005. *Characterizing the Evolution of Class Hierarchies*. in *Ninth European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE.

[7] E. H. Halili. 2008. *Apache Jmeter: A Practical Beginner's Guide to Automated Testing and Performance Measurement for Your Websites*.

[8] B. Liskov. 1988. *Keynote Address-Data Abstraction and Hierarchy*. ACM Sigplan Notices. **23**(5): p. 17-34.

[9] B. H. Liskov and J. M. Wing. 1994. *A Behavioral Notion of Subtyping*. ACM Transactions on Programming Languages and Systems (TOPLAS). **16**(6).

[10] B. Meyer. 1996. *The Many Faces of Inheritance: A Taxonomy of Taxonomy*. Computer. **29**(5): p. 105-108.

[11] L. Mikhajlov and E. Sekerinski. 1998. *A Study of the Fragile Base Class Problem*. ECOOP'98—Object-Oriented Programming: p. 355-382.

[12] E. Nasseri, S. Counsell, and M. Shepperd. 2008. *An Empirical Study of Evolution of Inheritance in Java Oss*. in *19th Australian Conference on Software Engineering (ASWEC)*. IEEE.

[13] E. Nasseri, S. Counsell, and M. Shepperd. 2010. *Class Movement and Re-Location: An Empirical Study of Java Inheritance Evolution*. Journal of Systems and Software. **83**(2): p. 303-315.

[14] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič. 2013. *Software Fault Prediction Metrics: A Systematic Literature Review*. Information and Software Technology. **55**(8): p. 1397-1418.

[15] A. Sabané, Y.-G. Guéhéneuc, V. Arnaoudova, and G. Antoniol. 2016. *Fragile Base-Class Problem, Problem?* Empirical Software Engineering. **22**(5).

[16] J. Stevenson and M. I. Wood. 2018. *Recognising Object-Oriented Software Design Quality: A Practitioner-Based Questionnaire Survey*. Software Quality Journal. **26**(2): p. 321-365.

[17] J. Stevenson and M. I. Wood. 2018. *Inheritance Usage Patterns in Open-Source Systems*. in *40th International Conference on Software Engineering*. Gothenburg, Sweden.

[18] D. H. Taenzer, M. Ganti, and S. Podar. 1989. *Problems in Object-Oriented Software Reuse*. in *ECOOP*.

[19] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. 2010. *The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies*. in *17th Asia Pacific Software Engineering Conference*.

[20] E. Tempero, J. Noble, and H. Melton. 2008. *How Do Java Programs Use Inheritance? An Empirical Study of Inheritance in Java Software*, in *Ecoop 2008–Object-Oriented Programming*, Springer. p. 667-691.

[21] L. Tolke and M. Klink. *Cookbook for Developers of Argouml: An Introduction to Developing Argouml, Revision 1.19*.