Inheritance Usage Patterns in Open-Source Systems

Jamie Stevenson
Department of Computer and Information Sciences,
University of Strathclyde, Glasgow, UK
jamie.stevenson@strath.ac.uk

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

ABSTRACT

This research investigates how object-oriented inheritance is actually used in practice. The aim is to close the gap between inheritance guidance and inheritance practice. It is based on detailed analyses of 2440 inheritance hierarchies drawn from 14 open-source systems. The original contributions made by this paper concern pragmatic assessment of inheritance hierarchy design quality. The findings show that inheritance is very widely used but that most of the usage patterns that occur in practice are simple in structure. They are so simple that they may not require much inheritance-specific design consideration. On the other hand, the majority of classes defined using inheritance actually appear within a relatively small number of large, complex hierarchies. While some of these large hierarchies appear to have a consistent structure, often based on a problem domain model or a design pattern, others do not. Another contribution is that the quality of hierarchies, especially the large problematic ones, may be assessed in practice based on size, shape, and the definition and invocation of novel methods – all properties that can be detected automatically.

CCS CONCEPTS

• Software and its engineering → Abstraction, modeling and modularity

KEYWORDS

Object-oriented, inheritance, open-source, empirical, design guidance.

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180168

1 INTRODUCTION

Inheritance is a key feature of the widely used object-oriented paradigm, allowing practitioners to define new program elements by building on what already exists and reducing duplication in source code. The aim of this research is to help close the gap between guidance on how inheritance should be used and how it is actually used in practice. Long standing guidance covers matters such as depth and width of hierarchies, reuse and type substitutability of hierarchy members, and keeping the core of the hierarchy abstract. This study provides a detailed analysis of how inheritance is actually used in practice by examining 2440 hierarchies. The study investigates the inheritance usage patterns that are present in production-quality code, with the aim of informing design choices and objectively improving design quality.

The study is based on 14 Java systems, mainly open source systems from the Qualitas Corpus [30]. Java is chosen due to the availability of 'real world' open source systems, its current popularity as an object-oriented development language [4], and its common use as a teaching language. The similarities between Java and other languages used in industry, such as C#, mean that this work will be open to replication in the wider software ecosystem.

In languages such as Java, inheritance is used to provide two quite distinct properties – type inheritance (polymorphism) and module reuse:

• Type inheritance - the subclass (subtype) is considered a subtype of the parent class (type). The new class can be substituted for the parent class in any design context.
• Module reuse - the subclass inherits the superclass attributes and method definitions, saving redefining and duplicating them in the subclass.

These two properties are often implemented using the same inheritance mechanism and mixed together in one hierarchy. Much of the advice on inheritance is concerned with how type inheritance and module reuse can be used while keeping designs comprehensible and maintainable.

Other aspects associated with inheritance that can affect practitioner comprehension and their ability to make changes include: the depth of an inheritance hierarchy - where there is a need to understand a sequence of ancestors [11]; overriding of method definitions - where children can alter the behaviour inherited from parents; and, self-calls - where method calls are

being propagated up a hierarchy and, potentially, out into the surrounding system.

The original contributions made by this paper concern pragmatic assessment of inheritance design quality. The results confirm that inheritance is being widely used, and therefore very important. The majority of hierarchies appear simple in structure – shallow and narrow – only causing limited maintenance challenge. However, the majority of classes actually defined using inheritance appear in large, more complex hierarchies that do require design assessment. Manual assessment highlights that these are often modelling a problem or solution domain concept. The new findings highlight the potential to automatically categorise and evaluate inheritance hierarchies based on branching points, shape, method definition, method invocation, as well as depth.

Throughout the paper the term class (and subclass, superclass) has been used generally to mean a member of an inheritance hierarchy. When it is more appropriate to refer to the 'type' (the set of requests to which a class can respond [11]) defined by a class then type (and subtype, supertype) is used. The term 'types' is also used to refer to the collection of classes, abstract classes and interfaces in a Java system.

2 RELATED WORK

Previous research aims to guide both the structure and the semantics of inheritance hierarchies. Inheritance use is complicated by its dual roles, as a reuse mechanism and as a type substitution mechanism [16]. While Liskov argues that reuse and subtyping should be kept separate [19], Meyer argues that "*If we accept classes as both modules and types, then we should accept inheritance as both module accumulation and subtyping*" [22].

It is argued that inheritance overuse, or even abuse, 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 [5, 29]. The concept of 'fragile base classes' has also been identified, where changes in a superclass may break a subclass or its dependents [23] – though recent work disputes how much impact fragile base classes actually have in practice [26]. Addressing the dangers of unintended inheritance interactions, Bloch argued that developers should "*design and document for inheritance or else prohibit it*" [2].

The Liskov Substitution Principle (LSP) [20] 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 relationships). Liskov also identifies "*convenience inheritance*", where inheritance is used simply as a reuse mechanism, as a weak form of usage.

In their design patterns catalogue, Gamma et al. introduce the principle of "*favouring object composition over class inheritance*" [11], arguing that composition should be preferred as a reuse mechanism (though many of their patterns still use inheritance). Martin's Dependency Inversion Principle (DIP) [21] argues that "*no dependency should target a concrete class*", which, when applied to hierarchies, advocates that only leaves should be concrete classes with the hierarchy core constructed from abstract classes.

Meyer provides a taxonomy of twelve different "*faces of inheritance*" [22] that takes a more relaxed approach to implementation inheritance, arguing there is a lack of "*strong theory that should support any such indictment*" but does note that "*has relation without is*", "*taxonomy mania*" and "*convenience inheritance*" are all improper uses of inheritance.

Although early object-oriented advocates, Johnson and Foote, suggested that hierarchies should be "*deep and narrow*" [17] to maximise reuse, most authors now propose that depth should be restricted: "*In practice, inheritance hierarchies should be no deeper than an average person can keep in his or her short-term memory. A popular value for this depth is six*" [25]. The SonarQube tool treats inheritance depth over five as a severe quality issue suggesting that "*Most of the time, a too deep inheritance tree is due to bad object oriented design*" [3].

Metrics associated with inheritance focus on Depth of Inheritance Tree (DIT) and Number of Children (NOC) associated with hierarchy members, again with the viewpoint that larger values for these metrics are an indicator of potential design weakness [5]. On the other hand, Gill and Sikka highlight that assessing negative factors such as hierarchy magnitude/size and complexity do not shed much light on the core desirable properties of hierarchies – reuse and substitutability [12].

Suryanarayana et al.'s hierarchy smells [28] also address depth, with a 'rule of thumb' that six is too deep, and width, suggesting that more than nine subclasses is too wide. Their bad smells also address 'broken' (non-LSP), 'rebellious' (overriding that restricts or cancels behaviour), cyclical, unfactored and unnecessary hierarchies.

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 [31] "*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 [7], with a depth of inheritance all the way up to 39.

In a survey on design quality with industry practitioners [27], 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*", "*abstract inheritance over object inheritance*", and "*derived types must satisfy the Liskov Substitution Principle … very difficult to achieve, so we try to use composition*". Respondents also indicated more concern for the depth of hierarchies than the width.

3 STUDY DESIGN

3.1 Research Objectives

The research goal is to investigate how inheritance is actually used in practice and to determine the extent to which it can be related to design guidance on software quality. This was achieved by analysing inheritance usage in 14 open source systems – see Table 1.

The research questions addressed were:

1. How is inheritance used in practice in open-source Java systems?
2. To what extent can inheritance hierarchies used in open-source systems be characterised?
3. To what extent can inheritance usage patterns be related to design quality where quality is defined via guidance, metrics, code smells and modelling?

The intention is to describe what is observed, rather than validate a metric or to find a correlation.

Table 1: Study Corpus Details

Name	Version	Types	Domain
aoi	2.8.1	493	3D, graphics, media
ant	1.8.4	1204	parser, build
argouml	0.34	1976	diagram visualisation
axion	1.0 M2	237	database
azureus	4.8.1.2	3319	torrent client
columba	1.0	1181	email client
freecol	0.10.7	654	game
freecs	1.3.20100406	139	chat server
freemind	0.9.0	445	diagram visualisation
galleon	2.3.0	258	3D, graphics, media
eclipse sdk	4.3	22629	IDE
gizmoball	-	86	student design
jdk	8u60	7713	language library
jhotdraw	7.0.6	310	graphics framework

3.2 Study Corpus

Eleven open source Java systems were selected from the Qualitas Corpus [30], including six from the evolution distribution. (The evolution package systems have a development history consisting of at least ten versions.) Together, the chosen systems covered a range of application domains, system sizes and development histories– see Table 1. Three additional systems were included: jdk – the core library of the Java language, an example of Java as it is spoken by language designers; jhotdraw – an exemplar of good design and design pattern usage [9]; gizmoball – a small game design challenge originally from MIT [34] and an example of design guidelines as taught to students. The third column of Table 1 provides the total number of types in each system.

While the corpus analysed here is smaller in number of systems than other recent studies, high-level similarities with other corpora will be shown. The range of system sizes is well 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 [24].

3.3 Study Instrumentation

Analysis was performed using a purpose-built tool based on the Eclipse JDT Core . While this tool is novel, the core components are very reliable as they are sourced from the Eclipse Project. Previous similar studies have used a range of tools e.g. purpose-built bytecode analysis [31], Byte Code Engineering Library [1], Soot Framework [32], Codecrawler [18], and MOOSE [13]. It was noted that Tempero et al.'s study on inheritance use in Java "*had memory limitations that restricted the size of the systems than we could analyse*" [32]. Similarly, Sabané estimated that the time and effort required to understand and adapt (possibly unsuitable and usually out of date) existing research tools is too high [26].

Each analysis is recorded in code or script, so experiments are explicitly documented to promote reuse, validation, and replication. In the interests of reproducibility and ease of tracing what processing was done, each sub-experiment was codified in its own package. Once an experiment has been run, the package responsible is not modified thereafter, providing a permanent record of the experimental process and environmental variables.

All counts are based only on the code within the system package, including any third party types and Java library types (excluding the ubiquitous Java.lang.Object). An effort was made to exclude testing code from the analysis, however for some projects this was difficult to achieve as it was tightly integrated with the main code base. Code written for use with injection frameworks was considered part of current development practice and therefore included in the analysis.

4 RESULTS

4.1 Use of Inheritance

Fourteen systems were analysed comprising a total of 40644 types (system range 86–22629 types) – see Table 1. The systems studied contain a total of 2440 inheritance hierarchies (range 6–1277 hierarchies), consisting of 22913 hierarchy members (range 2-1366 members per hierarchy). Abstract classes make up between 2% and 12% (median 6%) of the total types in the systems. Interfaces make up between 4% and 29% (median 10%) of total types.

Figure 1 shows proportions of types (including abstract classes and interfaces) defined by inheritance and interface implementation in each system. The figure shows that between 26% and 76% (median 54%) of all types are defined using inheritance (blue and green bars together) and between 54% and 85% (median 80%) are defined using either inheritance or interface implementation, or both (blue, green and red). For concrete classes only (not shown here), 38-80% (median 64%) are defined via inheritance and 71-90% (median 85%) are defined using either inheritance or interfaces, or both. Therefore, this result is generally in keeping with Tempero et al.'s finding that "*around three-quarters of user-defined classes use some form of inheritance in at least half the applications*" [31]. Here 13/14 systems define

69% or more of types using inheritance or interface implementation. 10/14 systems define at least 50% of types using inheritance alone.

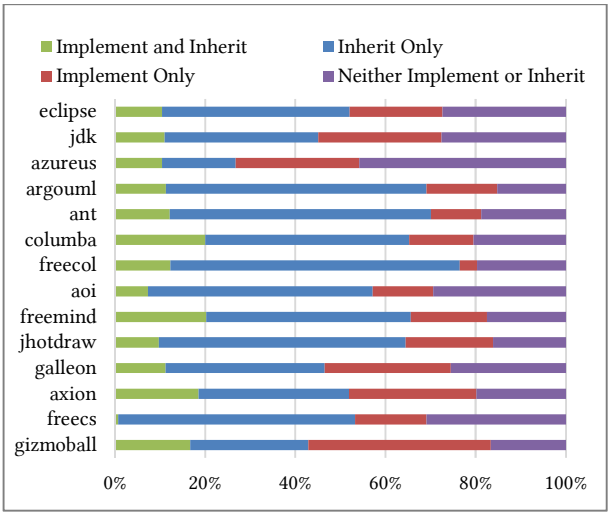


Figure 1: Total Types Defined Using Inheritance

4.2 Hierarchy Size, Depth and Width

The large majority of the 2440 hierarchies analysed were small: 80% of hierarchies contained seven or less members. Ninety percent contained 36 members or less. Figure 2 shows the distribution of hierarchy sizes following a typical power law distribution with the vast majority of hierarchies in the smaller sizes to the left and the long tail with few hierarchies all the way up to maximum size 1366.

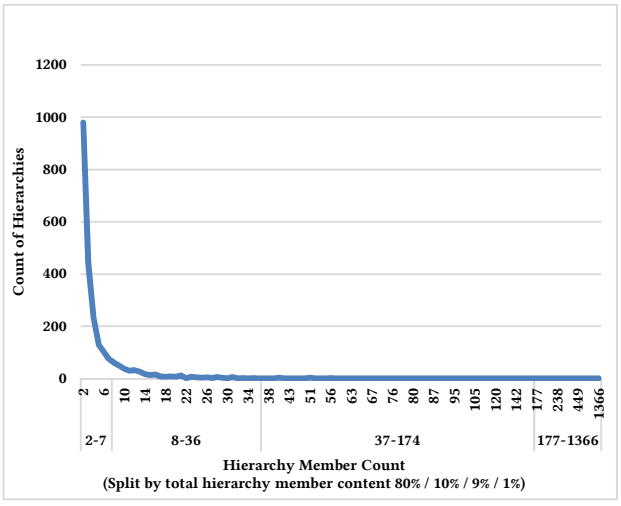


Figure 2: Distribution of Hierarchy Sizes

Over 80% of the extended classes are extended only once (64%) or twice (17%) indicating that most individual uses of inheritance do not significantly widen the containing hierarchy. On the other hand, all systems had individual classes that were extended five times or more, almost all (10/14) had classes that were extended at least 18 times, the maximum number of children found for any single class was 172.

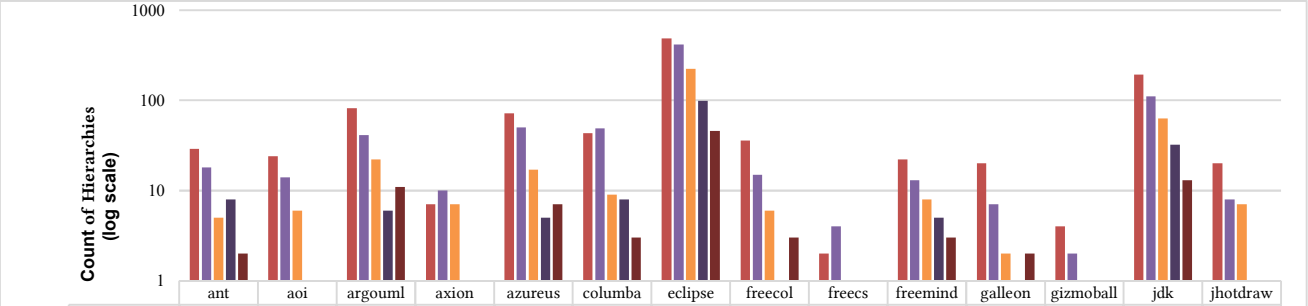
Table 2 shows the number of hierarchies found at each depth level across all of the corpus. Hierarchy depth ranged from DITMAX 1 to 10 (DITMAX is the maximum depth of a hierarchy with counting starting at zero at the root). Seventy one percent (1739/2440) of hierarchies observed were of depth DITMAX 1. Ninety percent of all hierarchies are depth 2 or less, 98% are depth four or less. Of the 2% (52) of hierarchies of depth five or more, 26 of these were found in eclipse and 14 in jdk.

Table 2: Number of Hiearchies at each Depth of Inheritance

Depth	1	2	3	4	5	6	7	8	10
ant	47	12	2			1			
aoi	38	4	2	1	1				
argouml	120	17	14	7	3	1			
axion	16	5	4						
azureus	118	26	5	1	1				
columba	88	19	4		1				
eclipse	874	260	78	39	12	6	5	2	1
freecol	51	7		2		1			
freecs	6								
freemind	34	14	1	1	1				
galleon	27	4	1						
gizmoball	5	1			1				
jdk	287	74	24	12	8	5	1		
jhotdraw	28	6	2		1				

In terms of width, previous studies have focused on the Number of Children (NOC) of individual hierarchy members [5] or Breadth of Inheritance Tree (BIT) [15] which summarises the width of a hierarchy into a vector of widths at each hierarchy depth. In this study, the interest was primarily in capturing a single notion of width per hierarchy. Hierarchy width was therefore defined as BITMAX, the maximum count of classes at any depth throughout the entire depth of a hierarchy.

Like the depth results, small values again dominate across the systems. Many inheritance hierarchies (1043, 43%) do not branch (are width one) and 21% (515) of hierarchies branch only once (width two). Most of the analysed systems, though, also contain a hierarchy in the top 10% of widths, and many having at least one in the top 1%. The top 1% of hierarchy widths were in the range 44-542, suggesting that hierarchies grow more by widening, than by deepening, and that widening appears much less restricted than depth.



	ant	aoi	argouml	axion	azureus	columba	eclipse	freecol	freecs	freemind	galleon	gizmoball	jdk	jhotdraw
Line	29	24	82	7	72	43	490	36	2	22	20	4	192	20
Fan	18	14	41	10	50	49	418	15	4	13	7	2	111	8
Subtrees	5	6	22	7	17	9	224	6		8	2	1	63	7
Branch-Line	8	1	6		5	8	99	1		5	1		32	1
Line-Branch	2	1	11	1	7	3	46	3		3	2		13	1

Figure 3: Number of Occurrences of each Shape of Hierarchy in each System

4.3 Public Interfaces and Method Novelty

A native interface is defined in this study as all the public methods provided by a concrete class or abstract class. This includes inherited public methods from superclasses and interfaces – the total set of public methods a practitioner needs to deal with for each class. Analysis of the size of native interfaces among hierarchy members yields another long-tailed distribution with 80% of hierarchy members having 10 or fewer public methods. Most inheritance members (and thus hierarchies) have relatively small (narrow) public interfaces. On the other hand public interfaces of size 150 all the way up to 1260 were found – these tended to associated with some external standard or domain entity.

A potential indicator of hierarchy design quality is how subclasses add methods to the parent class's public interface. These additional, 'novel', methods cannot be invoked on the parent type and must be accessed indirectly via overridden methods or via the subclass type itself. Adding methods to the public interface of a subclass is of limited value if the subclass is being used as type substitute for a parent - it may suggest that the hierarchy is emphasizing reuse instead. On the other hand, the parent type interface may be being used in one design context and the added methods interface in an extended context. It may therefore be possible to assess the quality of hierarchy design by examining how and where novel methods are used.

Across all hierarchies, at depth level one, 47% of added methods are novel. By the time we get to a depth of four, 84% of the cumulative method definitions are novel. As depth increases, it appears the type interface is increasingly novel. On the other hand, around a twentieth (9542, 6.51%) of the hierarchy members analysed do not add any novel public methods to their interface. It is possible that these hierarchy members are mostly, or completely, motivated by polymorphism. Hierarchies with no novel methods (7%, 398 in total) appear to be constrained in growth – with most of these (91%, 364) at depth one and none over depth three. Overall, nearly a quarter of hierarchy members (5070) have zero novelty (introduce no new methods).

4.4 Hierarchy Shape

Five distinct hierarchy shapes were found: Line, Fan, Line-Branch, Branch-Line and Subtrees.

- *Line* - hierarchy with no branching (BITMAX=1).
- *Fan* - hierarchy consists solely of one root and its immediate children (DITMAX=1). (Similar to Lanza's 'flying saucer' shape [18].)
- *Line-Branch* - hierarchy root node that has a single child, then branches later (one or more times)
- *Branch-Line* - hierarchy root node branches, and has no later branches, but does have depth of more than one (extended Fan).
- *Subtrees* - hierarchy consists of a branching root with at least one other branching node – see Figure 4.

These categories are based on observations, can be applied to any hierarchy, and include all possible legal hierarchy structures. Figure 3 shows the counts of each shape of hierarchy in each of the systems in the corpus. Most systems show a similar break-down of shapes, with Line, Fan, then Subtrees being the most popular (note the logarithmic scale).

Line dominates – a further indication that much of the inheritance present in systems is uncomplicated. The next most common shape in each system is Fan, which represent many one-deep variations on a single superclass. The Line and Fan shapes account for 74% (1801) of all hierarchies examined – 94% (979) of Line-shaped hierarchies are of depth one, and all but one Line hierarchy is shallow (DITMAX 1-3), indicating also that depth

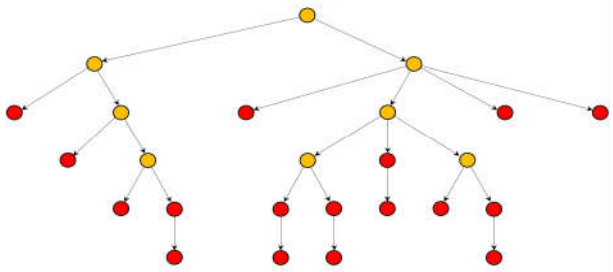


Figure 4: Sub-Trees (yellow nodes abstract / red concrete) - org.eclipse.jdt.apt.core.internal.declaration.EclipseDeclarationI

without branching is rare. Additionally, if a hierarchy does not branch at the root (Line, Line-Branch), it is unlikely to branch subsequently.

The next most common shape is Subtrees, which capture 86% of the hierarchies in medium (DITMAX 5-6) and 96% of deep (DITMAX 7-10) hierarchies – indicating that branching is usually required for depth. Of the 497 (24%) hierarchies with a width (BITMAX) greater than four, 53% (263) of these are Subtrees (31% are Fan), if we consider hierarchies larger than width 19, the proportion that are Subtrees increases to 79% - so not only do wide hierarchies branch, their branches branch.

The Subtree category constitutes 15% of hierarchies examined, *but contains 63% of all hierarchy members* due to the size of these hierarchies. This has implications for the perception of inheritance in a system. With almost two thirds of hierarchy members being part of wider, deeper, overall larger hierarchies, this may be the default impression of inheritance usage, even though these members represent only 15% of hierarchies.

4.5 Abstract Classes

Fifteen percent (3408) of hierarchy members are abstract classes, roughly half of these as hierarchy roots (1760) and half appearing inside a hierarchy (1648). Fifty four percent (1325) of the inheritance hierarchies examined contain at least one abstract class – with 50% (1207) containing one to three abstract classes. However, there is noticeable variation between systems with extremes of none and many – jdk notably has a high number of abstract class hierarchy roots, but many of these implement no interfaces.

Hierarchies are seen to have either a contiguous backbone where all the abstract classes in the hierarchy form a contiguous structure, or a fragmented backbone where gaps appear in the structure of abstract classes. In the hierarchies observed, only 3% (70 of 2440) of the hierarchies have abstract classes that are not in a contiguous sub-graph within the containing hierarchy. When broken down by hierarchy shape, 93% (65) of these fragmented hierarchies have a Subtree shape, with the other shape categories having only a few fragmented hierarchies.

The yellow nodes in Figure 4 represent abstract classes, therefore the hierarchy shown there is an example of a 'contiguous abstract backbone'.

A hierarchy is considered out-of-order if it contains a sequence of types where one type inherits from a less abstract type – this is direct breach of the DIP - where an abstract class inherits from a concrete class. Only 6% (146) of hierarchies are out-of-order. Subtrees again account for 75% (109) of these – so in larger hierarchies, where a continuous chain of abstractions may be of most use in managing a complex abstraction, there is more chance of a breach of dependency inversion (DIP) occurring.

4.6 Method Invocation

If a hierarchy design is intended for type substitution then it would be expected that most method invocations are to methods defined by the hierarchy root type. On the other hand, non-root type method invocation could suggest that the hierarchy members are making more use of the hierarchy as a reuse mechanism – though it could also be that non-root methods are legitimately being used in different design sub-contexts. Forty-five percent of hierarchy members have no direct method calls (excluding constructors), perhaps suggesting that they are being accessed polymorphically – or perhaps they are 'reuse placemarkers' in the hierarchy. In all but one system (freecol) a high majority of method calls to hierarchy members are either via root types themselves or via root type methods. 31%-93% (median 71%) of all hierarchy member method calls were via the root type. The remainder of method calls are on subtypes split between inherited methods and local, novel, methods.

Each hierarchy member can be classified as one of: never directly accessed, accessed only via root-type method calls, invoked only via local (novel) methods, or invoked using a mixture of root and local. It is possible to automatically classify hierarchies using these different access mechanisms: *Strict Polymorphism* – all accesses to hierarchy members are via root type references; *Common Interface Polymorphism* – all accesses to hierarchy members are via root type methods; *Balanced Reuse* – uses novel methods and at least one invocation of a root method; *Aggressive Reuse* – members only accessed via novel methods; *Fragmented* – a mixture of the previous.

While it is obvious in the case of Strict and Common Interface Polymorphism that substitutability is important, the other categories are less clear. With increasing invocations of non-root methods, the hierarchies in the reuse-categories appear to be less well defined by the root of the hierarchy, indicating that polymorphism is perhaps less important in these cases. With decreasing method invocations to root methods, these have been distinguished by the severity of this 'distance' from the root type. Finally, the fragmented category is assigned where there is no other suitable category.

Fifty-four percent of hierarchies are in the Strict or Common Interface Polymorphism categories indicating that access to just over half of hierarchies is exclusively polymorphic. Of these, 47% are accounted for by Fan and Line hierarchies. Again, it is the larger hierarchies that tend to be non-polymorphic, suggesting they are more likely to be driven by reuse (or a mixture of polymorphism and reuse). It is important to note, again, that these large hierarchies are where most hierarchy members reside – those with 37 or more members are usually 'fragmented'. The larger hierarchies appear to have 'inconsistent' usage patterns. It is important to stress, as above, the skewing effect of large hierarchies on what might be viewed as 'normal'.

4.7 Casting

Closely related to polymorphic access is casting – converting one type to another usually to access methods specifically associated with the destination type. This is necessary if we have access to a subtype via a root type reference and wish to invoke methods specifically defined in the subtype. Casting, or excessive casting, may be viewed as a sign of weakness in the design of a hierarchy [33]. Across the systems, 15-34% (median 24%) of hierarchy members are involved in cast operations. Of these casts, 89% are within hierarchy – from one hierarchy member to another, 6% are into a hierarchy from outside, and 4% are out of a hierarchy from

inside. In terms of average cast per hierarchy member there is a steady decline from root types to leaf types. This may indicate that root abstractions are frequently insufficient for meeting the needs of hierarchy clients in these systems. A limitation of this analysis is the exclusion of interfaces used within hierarchies.

4.8 Large Hierarchies

Forty-eight of the most complex hierarchies were manually inspected. Twelve of these hierarchies were identified as 'generalisation *(is-a)*' hierarchies. Seven were characterized as 'grouping' hierarchies where the root abstraction is small and well-defined but does not define the subclasses and there is lots of semantic variation in the hierarchy. Eight are characterized as 'cross cutting concern' where the hierarchy represents an aspect or trait representing an implementation detail or architectural abstraction e.g. a design pattern. The remaining 21 of these large hierarchies could not be easily classified. It was also found that 25/48 of the large hierarchies were related to recognized architectural features – mainly design patterns. Two other motivations for large hierarchies appeared to be as frameworks hooks [10] – well defined application plugin points – and 'specification ties' where the structure appeared closely bound to an external standard e.g. a document standard.

Forty-five of these large hierarchies show a Fragmented or Balanced Reuse invocation profile, with only the most generic hierarchies having Common Interface Polymorphism. Most (36) also show a mixture of overriding and new methods. Both reuse and substitutability appear to be strong design factors for some large hierarchies but not all.

There is no indication that external motivations for large hierarchies correspond to empirical properties, indicating that understanding of both is necessary to assess these hierarchies.

The deepest hierarchy encountered (eclipse EventManager, DITMAX=10) is an efficient replacement for a common Java library type (java.util.Observable). However, it appears that many of the sub-classes of EventManager are "*implementation inheritance*" [19] seeking to re-use EventManager public methods and functionality while, in many cases, ignoring the accumulated functions of the superclasses. The next two deepest (eclipse Viewer and eclipse Window) have 85 and 361 members respectively. Their characteristics include being both wide and deep, having some continuous abstract backbones, having fragmented method invocation – some subtypes manipulated directly (particularly as depth increases) and some via root type methods.

5 ANSWERS TO RESEARCH QUESTIONS

5.1 RQ1: How is inheritance used in the corpus?

Inheritance is present in significant amounts in the corpus – typically over 50% of all types are defined using inheritance, over 80% if interface implementation is included, and 64% of concrete classes are defined using inheritance (median values across the 14 systems). Inheritance is therefore clearly very important to software design and construction using Java.

However, most of these hierarchies are very simple in structure – 80% of hierarchies contain seven members or less, most inheritance is shallow (71% depth one, 19% depth two), most hierarchies (64%) are either width one or two. Most hierarchy members are only extended once or twice. Depth rarely goes beyond four. Width is less constrained, often reaching 10 and often going beyond this. The simplicity of most hierarchies may mean that they are relatively easy to understand and maintain and that there is little need to be concerned about their overall design quality.

Large hierarchies were rare, only 2% were depth four or more (and most of these were in two systems – jdk and eclipse). Only nine hierarchies out of 2440 were of depth greater than six. Wider hierarchies are more common – 229 hierarchies of width ten or more, 44 hierarchies with a width of 44 or more (again eclipse contributed 50% of these). Although larger hierarchies form a small percentage of the overall population, these are where the majority of hierarchy members reside (63% of hierarchy members). Therefore, developers are just as likely, or more likely, to encounter a class sitting in a large hierarchy as a small hierarchy. In terms of design quality, it seems that it is the small number of larger hierarchies that warrant close design consideration.

5.2 RQ2: How can inheritance hierarchies be characterised?

Hierarchies can be categorised by shape (Line, Fan, Line-Branch, Branch-Line, Subtrees) – which are present in similar proportions across the systems analysed here. Many hierarchies are simple (Line and Fan) reflecting again that much inheritance seems trivial. Hierarchies can grow arbitrarily wide, indeed, deeper hierarchies without branching are rare and very deep hierarchies always branch.

Over half of the hierarchies examined were categorised as Strict or Common Interface Polymorphism where access is exclusively polymorphic – via the root type or the root type defined methods only. Of these, 47% are accounted for by Fan and Line hierarchies. Hierarchy members with no-novel methods are usually in shallow hierarchies.

It is the larger hierarchies that tend to be non-polymorphic, suggesting they are more likely to be driven by reuse. Overriding declines with depth in a hierarchy, while novel methods continue to be added, indicating that initial subclasses tend to refine root behaviour, and deeper subclasses tend to add behaviour.

The most obvious and significant influence on large hierarchies are elements from the problem or solution domain such as a protocol, document format, or design pattern. These provide regularity in the hierarchies which allows faster comprehension and identification of inconsistent structure. These are specific to each hierarchy however, and do not necessarily inform a general inheritance assessment strategy.

The use of abstract classes to form structures within hierarchies is common, but not universal. This abstract structure is generally continuous within a hierarchy or not present at all, though in very larger hierarchies this structure becomes fragmented.

Casting is common in the examined systems, including a significant number of casts involving inheritance hierarchy members. Root abstractions are commonly being bypassed, which may mean that reuse has come at the cost of poor abstraction. Alternatively, it may just be that clean, versatile abstractions are hard to define in some cases.

5.3 RQ3: To what extent can inheritance usage patterns be related to design quality?

Much of the design quality guidance relating to inheritance is structural advice. For example, metrics such as C&K, code smells and tools such as SonarQube focus on depth of hierarchy. SonarQube by default highlights a depth of six or more. Others have debated various depths at which potential maintenance issues can arise. Previous census studies have found the majority of inheritance is at depth two or less [9]. The results here are consistent with that finding – here 98% of hierarchies are smaller than depth four. Wide hierarchies are more common. Arguably, width is less of a design challenge than depth since subclasses at the same level can often be understood independently of their neighbours. As a simplistic indicator, a depth of more than three or four seems a good indicator that a hierarchy might warrant closer examination – especially given the finding that deeper hierarchies are more likely to add novel methods that cannot be accessed directly through the hierarchy root type.

The majority of classes defined by inheritance reside in 'large' hierarchies – hierarchies of depth four or greater and with tens or hundreds of members. Design guidance that relates to such hierarchies includes LSP - polymorphic rather than reuse hierarchies [20, 22, 28] and DIP – depend on abstractions [21]. The findings here suggest that analysis of hierarchy method definition, hierarchy method invocation, use of abstract classes and casting, together, offer potential insights into the nature and quality of a hierarchy – in particular the extent to which it is being used polymorphically or for reuse. Such analyses offer potential insights into areas within larger hierarchies that are starting to drift from the root purpose and could be better broken into separate hierarchies or achieved using an alternative mechanism such as interface implementation and composition.

Only 15% of hierarchy members are abstract, a lower proportion than expected if DIP was being pursued rigorously. Part of the explanation for this is the small size of the majority of hierarchies. When abstract classes are present they usually occur in a 'continuous backbone'. There also appears to be an 'all-or-nothing' approach to the use of abstract classes – possibly indicating different practitioner styles [6]. As hierarchies grow very large, the abstract backbone tends to become fragmented. Ninety three percent of 'fragmented hierarchies' and 75% 'out of order' hierarchies were Subtrees – again an indicator that it is these hierarchies that warrant closer design attention.

Meyer's taxonomy for inheritance [22] describes three broad categories of inheritance use, depending on the model inspiring the hierarchy: model inheritance (is-a relationships in the problem domain), software inheritance (solution domain relationships), and variation inheritance (similarity or difference relationships). There is evidence that model and software inheritance are present in significant amounts in the large hierarchies examined, and that these supersede the need for some lower level guidance. However, these were detected via manual inspection and do not have consistent indicators in the counting metrics. This is consistent with previous attempts to validate this taxonomy [8].

6. DISCUSSION AND IMPLICATIONS

It is clear from this study and related studies [31] that inheritance is widely used in real-world objected-oriented systems written in Java. Typically, over 60% of concrete classes were defined using inheritance in the systems analysed here. Inheritance is therefore a key topic for object-oriented practitioners and for students studying software development. Given both its subtyping and reuse benefits, but also the risks of associated complexity, it is therefore important to investigate whether inheritance usage and teaching could or should be limited to 'best practice patterns'.

This study confirms findings from previous work that much inheritance usage is in the form of very simple structures – structures that can provide subtyping and reuse benefits with little danger of adding unnecessary complexity to software. The large majority of hierarchies found in this study are depth one or two and take the form of Line (width one) or Fan (depth one) hierarchies. The majority of hierarchies had less than seven or less members. These shallow depth findings are consistent with Ferreira et al. who found a 'typical' value of depth two [9] and Tempero et al. who observed that most classes appear in the shallow parts of hierarchies [31].

The simplicity of this majority of hierarchies is likely to mean that they are relatively easy to understand and maintain, and that there may be little need to be concerned about their overall design quality while they remain so simple.

A typical example of such a 'simple' hierarchy is *org.eclipse.swt.widgets.Dialog*. This is a 'Fan' hierarchy with seven leaves. There is no overriding in this hierarchy – the shared behaviour from the abstract root type is a 'minimal' set of behaviour/functionality. The lack of overriding makes each leaf in the hierarchy a simple extension of the root type.

Although large hierarchies are rare, because of their size, these are where many members that are defined reside. It is this small number of large hierarchies that warrant close design attention. How can they be identified and what criteria might be used to assess their design quality?

A simple source of identification is depth. In the analysed systems, almost all hierarchies were depth one or two, 98% were depth four or less. Certainly hierarchies beyond a depth of four warrant close design attention. Again these depth findings are reasonably consistent with previous work. In a survey of practitioners, Gorsheck et al. obtained preferred thresholds of three or five for inheritance depth [14] (although 20% of respondents did not care about inheritance maximum depth, and 31% responded "*indicating an awareness of depth but a tendency not to act on this information*"). Riel [25], code smells [28] and SonarQube [3] have set higher thresholds of depth of six. This work indicates that such depth are rare, and only occur in the largest systems.

One issue to be aware of is that, in this study, the largest systems contained the deepest hierarchies. Although deep hierarchies were very rare, if the study had included more of these large systems (those with thousands of user-defined types) then more large hierarchies may have been found. There is no reason to expect the overall percentage of large hierarchies to alter, however. Also, the two largest systems studied here are mature frameworks which have been subject to intense design effort over the last decade or longer.

Almost all the systems in the study corpus had 'wide' hierarchies - exceeding the 'bad smell' advice of width nine [28]. Although, most hierarchies were also narrower than Johnson and Foote's maximum width of 27 [17]. Again, many of these wider hierarchies are very shallow, and as such are considered unlikely to cause significant maintenance challenge. Wider hierarchies may create more paths from root to leaf, but sibling classes at the same depth do not increase complexity in the way ancestors do. It is suggested that width, particularly shallow width, is more likely to be associated with type substitutability rather than reuse – as demonstrated by the eclipse *Dialog* example described above.

Other than depth, what other factors may indicate problems and how can these larger hierarchies be assessed for quality? This research identified five different hierarchy shapes. Two of these shapes are the simplest of hierarchy structures Line and Fan, and again they are the most common. Two related shapes, Branch-Line and Line-Branch also appear relatively unproblematic, due to their inherent lack of branch-points. It is the Subtree hierarchies, with multiple branch-points – see Figure 4, that tended to be the largest and most complicated. So as well as depth, number of branch-points may be an indicator that a hierarchy warrants close design attention.

Examining how a hierarchy defines methods and how those methods are invoked throughout the system may provide deeper insights into the fundamental nature and quality of an inheritance hierarchy. If a hierarchy is being used polymorphically, in keeping with LSP, it would be expected that access would mainly be through the hierarchy root type or at least through methods defined by the root type. Characterisation of hierarchies in this way may help distinguish hierarchies that are designed with polymorphism in mind as opposed to those that are designed for reuse (or both).

For example, a Fan with Strict Polymorphism and uniform overriding in the leaves is a comprehensible, regular structure regardless of how wide it becomes. While a structure with inconsistent patterns of overriding may indicate a poorly defined abstraction. Assessing local regularity may be more useful than concern over comparing numerical values between hierarchies. This may provide guidance for interpreting the 'fitness for purpose' of a hierarchy. If this is generally the case, this may indicate that once overriding stops in a hierarchy, the deeper portions which are merely adding new behaviour may instead be 'composed' with the shallower parts of the hierarchy. On the other hand, it seems quite valid to add novel methods which are only to be used in a localized context within a design while general usage of the class is restricted to root type method access.

Another finding is that many of the larger complicated hierarchies are strongly influenced by a problem or solution domain model. If a hierarchy satisfies other design criteria (usefulness, domain modelling, low complexity), a depth cut-off seems arbitrary. These relationships appear harder to detect automatically but are often quite visible under manual analysis. A problem is that some of the largest hierarchies appeared initially motivated by such factors but had then grown out of control and become major legacy issues for their containing designs. This is where it seems the biggest inheritance design challenges reside – being able to detect that a hierarchy is losing its original design motivation and quality, and being able to recognize that an alternative design structure would be more appropriate.

7. THREATS TO VALIDITY

Using open-source systems as a proxy for real-world development is a threat to the external validity of this work. Given the difficulty of analysing propriety program code, open-source is often used as a substitute for closed source software. However, it is important to be aware that there are real risks in doing so, these open-source systems are unlikely to be subject to the same design and review practices associated with commercial software.

There are also external validity dangers in the selection of the corpus used in this study. Care was taken to select a range of system sizes, problem domains and system maturity levels. It is argued that the corpus shares similarities with corpora used in comparable studies. Some of the high level findings such as the prevalence of inheritance suggest that the corpus properties are in keeping with related research.

Although many of the findings appear consistent across the systems, the deeper and more complex inheritance hierarchies occur in the two largest systems (eclipse and jdk). Tempero et al. noted in their work that "*the depth of classes in the inheritance tree, does not increase with program size*" [31], but here it is the two largest systems that contain the majority of these rare deep hierarchies. It is perhaps the case that there is more opportunity to accommodate deeper hierarchies in larger systems.

In terms of internal validity, the size of the corpus was on the small side compared to earlier census-style research. The reason for the relatively small corpus size, 14 systems, was the depth of analysis performed. Each system had its own particular package configuration which required understanding. Extracting and fully analysing the source code for each individual system was a significant effort.

In terms of construct validity, creating a new tool carries with it inherent risks. At each stage in the building process the results from the tool were compared with reference hierarchies in the testing suite. In addition, spot checks were carried out on the actual corpus results to confirm that the tool had correctly recorded inheritance structures. Finally, any easily reachable measures that could serve as a checksum were used for data validation e.g. depth of inheritance (DIT) and number of children (NOC).

8. CONCLUSIONS

It was found that inheritance was heavily used in the corpus. In most of the systems at least 50% of the concrete classes that are defined are involved in an inheritance relationship.

Most hierarchies are of depth one, or two maximum. Many hierarchies are of width one. Width one was described as a Line shape and depth one as a Fan. Together, these account for 74% of all hierarchies in the analysed systems. Over 80% of hierarchies contained seven or less members. It is suggested that most of these hierarchies are unlikely to cause major comprehension or maintenance issues - even if they are being used for reuse rather than type substitutability.

Despite the relatively small number of larger hierarchies, this is where the majority of classes defined via inheritance reside – as much as 63% of classes sit in the larger hierarchies. Developers are therefore just as likely to encounter a class defined in one of these hierarchies. It is these hierarchies that should be the focus of most design effort.

How can the design quality of these hierarchies be assessed in a practical way? It was found that many of the large, apparently complex hierarchies, were modelling a problem domain or solution concept such as a document standard or a design pattern. However, these required close manual examination to determine this.

Depth of hierarchy remains a simple warning of potential hierarchy design issues. As stated above many hierarchies are only depth one or two, 98% are depth four or less. Based on this data, hierarchies that are of depth four and beyond warrant closer design attention.

A novel contribution of this research is the characterization of hierarchies by shape. Most hierarchies are Line or Fan in shape. Many others are Line-Branch or Branch-Line. The deepest, most complex hierarchies are Subtree in shape, with multiple branch-points. Along with depth, multiple branch points seems another practical indication that the design of the hierarchy should be examined in more detail.

A further new contribution is the characterization of hierarchies by method definition and method invocation. These analyses enable determination of the extent to which a hierarchy member is being accessed polymorphically – through its root type or through root type methods. In practice, this would seem to indicate good quality hierarchy design in keeping with the Liskov Substitution Principle.

Another new finding is that many hierarchies added novel methods that could not be accessed directly via the root type, and that novelty increased noticeably as hierarchy depth increased. This is another indicator that increased hierarchy depth is more likely to be a concern. The addition of novel methods may be an indication that a hierarchy is being used for reuse rather than type substitution – not necessarily a poor practice, but again a potential design warning sign. Type casting to access novelty is likely to be another hierarchy design danger sign.

A topic for further research is the study of inheritance hierarchy evolution. Some of the key findings here on shape, particularly branching, and hierarchy characterization in terms of root method access and novel method addition, could be used as warning signs as a hierarchy evolves. There is potential to study the complexity of the identified inheritance shapes and also any possible relationship to defect occurrence. Refactoring advice might be given based on the identified usage patterns and structures.

Finally, this study did not include interfaces and their implementation in much of the analysis. Interfaces are often found at the roots of hierarchies and, potentially, much of the access to hierarchy members is through interface-defined types. Future studies should therefore integrate interface analysis with inheritance analysis.

9. REFERENCES

[1] Baxter, G., M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero. *Understanding the Shape of Java Software.* in *ACM Sigplan Notices.* 2006. ACM.

[2] Bloch, J., *Effective Java.* 2008: Pearson Education India.

[3] Campbell, G. and P.P. Papapetrou, *Sonarqube in Action.* 2013: Manning Publications Co.

[4] Cass, S., 2017. *The 2017 Top Ten Programming Languages.* IEEE Spectrum, July.

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

[6] Chow, J. and E. Tempero. *Stability of Java Interfaces: A Preliminary Investigation.* in *Proceedings of the 2nd International Workshop on Emerging Trends in Software Metrics.* 2011. ACM.

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

[8] English, M., J. Buckley, and T. Cahill. *Applying Meyer's Taxonomy to Object-Oriented Software Systems.* in *Source Code Analysis and Manipulation, 2003. Proceedings. Third IEEE International Workshop on.* 2003. IEEE.

[9] Ferreira, K.A., M.A. Bigonha, R.S. Bigonha, L.F. Mendes, and H.C. Almeida, 2012. *Identifying Thresholds for Object-Oriented Software Metrics.* Journal of Systems and Software. 85(2): p. 244-257.

[10] Froehlich, G., H.J. Hoover, L. Liu, and P. Sorenson. *Hooking into Object-Oriented Application Frameworks.* in *Proceedings of the 19th international conference on Software engineering.* 1997. ACM.

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

[12] Gill, N.S. and S. Sikka, 2011. *Inheritance Hierarchy Based Reuse & Reusability Metrics in Oosd.* International Journal on Computer Science and Engineering. 3(6): p. 2300-2309.

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

[14] Gorschek, T., E. Tempero, and L. Angelis. *A Large-Scale Empirical Study of Practitioners' Use of Object-Oriented Concepts.* in *32nd International Conference on Software Engineering.* 2010. IEEE.

[15] Harrison, R., S. Counsell, and R. Nithi, 2000. *Experimental Assessment of the Effect of Inheritance on the Maintainability of Object-Oriented Systems.* Journal of Systems and Software. 52(2): p. 173-179.

[16] ISO, *Iec/Ieee 24765: 2010 Systems and Software Engineering-Vocabulary.* 2010, Technical report, Institute of Electrical and Electronics Engineers, Inc.

[17] Johnson, R.E. and B. Foote, 1988. *Designing Reusable Classes.* Journal of object-oriented programming. 1(2): p. 22-35.

[18] Lanza, M., 2003. *Object-Oriented Reverse Engineering Coarse-Grained, Fine-Grained, and Evolutionary Software Visualization.*

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

[20] Liskov, B.H. and J.M. Wing, 1994. *A Behavioral Notion of Subtyping*. ACM Transactions on Programming Languages and Systems (TOPLAS). 16(6): p. 1811-1841.
[21] Martin, R.C., *Agile Software Development: Principles, Patterns, and Practices*. 2003: Prentice Hall PTR.
[22] Meyer, B., 1996. *The Many Faces of Inheritance: A Taxonomy of Taxonomy*. Computer. 29(5): p. 105-108.
[23] Mikhajlov, L. and E. Sekerinski, 1998. *A Study of the Fragile Base Class Problem*. ECOOP'98—Object-Oriented Programming. p. 355-382.
[24] Radjenović, D., 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.
[25] Riel, A.J., *Object-Oriented Design Heuristics*. 1996: Addison-Wesley Longman Publishing Co., Inc.
[26] Sabané, A., Y.-G. Guéhéneuc, V. Arnaoudova, and G. Antoniol, 2016. *Fragile Base-Class Problem, Problem?* Empirical Software Engineering. 22(5): p. 2612–2657.
[27] Stevenson, J. and M.I. Wood, 2017. *Recognising Object-Oriented Software Design Quality: A Practitioner-Based Questionnaire Survey*. Software Quality Journal.
[28] Suryanarayana, G., G. Samarthyam, and T. Sharma, *Refactoring for Software Design Smells: Managing Technical Debt*. 2014: Morgan Kaufmann.
[29] Taenzer, D.H., M. Ganti, and S. Podar. *Problems in Object-Oriented Software Reuse*. in *ECOOP*. 1989.
[30] Tempero, E., C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. *The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies*. in *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*. 2010. IEEE.
[31] Tempero, E., J. Noble, and H. Melton, *How Do Java Programs Use Inheritance? An Empirical Study of Inheritance in Java Software*, in *Ecoop 2008–Object-Oriented Programming*. 2008, Springer. p. 667-691.
[32] Tempero, E., H.Y. Yang, and J. Noble. *What Programmers Do with Inheritance in Java*. in *European Conference on Object-Oriented Programming*. 2013. Springer.
[33] Van Emden, E. and L. Moonen. *Java Quality Assurance by Detecting Code Smells*. in *Ninth Working Conference on Reverse Engineering*. 2002. IEEE.
[34] Yue, K.-B., T.A. Yang, W. Ding, and P. Chen, 2004. *Open Courseware and Computer Science Education*. Journal of Computing Sciences in Colleges. 20(1): p. 178-186.