University of Strathclyde
Department of Computer and Information Sciences

PhD Computer Science 2019

Closing the Gap Between Guidance and Practice

An investigation of the relevance of design guidance to practitioners using object-oriented technologies

Submitted by
Jamie Stevenson

Abstract

This thesis investigates if object oriented guidance is relevant in practice, and how this affects software that is produced.

This is achieved by surveying practitioners and studying how constructs such as interfaces and inheritance are used in open-source systems.

Surveyed practitioners framed ‘good design’ in terms of impact on development and maintenance. Recognition of quality requires practitioner judgement (individually and as a group), and principles are valued over rules. Time constraints heighten sensitivity to the rework cost of poor design decisions.

Examination of open source systems highlights the use of interface and inheritance. There is some evidence of ‘textbook’ use of these structures, and much use is simple. Outliers are widespread indicating a pragmatic approach. Design is found to reflect the pressures of practice – high-level decisions justify ‘designed’ structures and architecture, while uncertainty leads to deferred design decisions – simpler structures, repetition, and unconsolidated design. Sub-populations of structures can be identified which may represent common trade-offs.

Useful insights are gained into practitioner attitude to design guidance. Patterns of use and structure are identified which may aid in assessment and comprehension of object oriented systems.

Acknowledgements

I would like to thank my supervisor, Dr Murray Wood, for his patience and thoughtful advice throughout the course of my studies. His insight and encouragement have been invaluable in completing this thesis.

Other members of staff in the Computer and Information Sciences department have been very supportive – in particular – Dr Marc Roper, Dr John Wilson, and Professor Ian Ruthven.

I would like to thank my partner, family, and friends for putting up with my absence for the last few years – a PhD is another life that is difficult to share.

As noted in previous publications, I would like to thank the anonymous respondents who took time to complete the questionnaire and those who helped to distribute it more widely.

Finally, I would like to thank other members of the academy who helped direct my work:

Prof June Verner – who provided practical advice about the value of empirical software engineering at the EASE '14 doctoral symposium.

Dr Bram Ridder, King's College London - for detailed comments on an early version of the survey study.

My thanks also to the peer reviewers for their very useful feedback – I cannot thank you individually, but your efforts often go unappreciated.

Declaration

This thesis is the result of the author’s original research. It has been composed by the author and has not been previously submitted for examination which has led to the award of a degree.

Signed:

Date:

Publications

Parts of this work have been published previously as noted below:

An early research proposal and initial data was presented at a doctoral symposium:

Stevenson, J., 2014. Research proposal: Objective evaluation of object oriented design quality, *EASE '14 Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, Article No. 53, DOI:10.1145/2601248.2613080

The findings from Chapter 3 as a long-form journal article:

Stevenson, J. & Wood, M. 2017. Recognising object-oriented software design quality: a practitioner-based questionnaire survey *Software Quality Journal*, June 2018, Vol. 26, Issue 2, pp 321-365, Springer, DOI: 10.1007/s11219-017-9364-8

The majority of Chapter 4 as a conference paper to ICSE 2018:

Stevenson, J. and Wood, M. 2018, Inheritance Usage Patterns in Open-Source Systems, *ICSE '18: 40th International Conference on Software Engineering, May 27-June 3, 2018, Gothenburg, Sweden*, DOI: 10.1145/3180155.3180168

Index

1 Introduction

Software is pervasive. Practically every interaction with modern infrastructure – from disposable devices to safety critical systems passes though software at some stage. However, the fast-moving disciplines producing software have not acquired the rigour and bureaucracy of the more traditional engineering disciplines. This is evidenced by the high rate of failure in software projects (Charette, 2005; Gartner Inc., 2010; Verner et al., 2008).

1.1 Context

Software quality is a multi-faceted attribute which can incorporate many aspects such as fitness for purpose, efficiency, maintainability, security, and stability (BSI, 2011; Kitchenham and Pfleeger, 1996). These properties can be traced to different parts of software production, including but not limited to – design methodologies, development or testing processes, static code analysis, tooling, and runtime artefacts.

Quality concerns also arise from different stakeholders such as users, designers, and maintainers. Threats to quality can arise from the nature of software projects - successful software will change (Lehman and Ramil, 2003) and problems are often underspecified, or contain sources of uncertainty (Beck, 2014). This leads inevitably to questions about maintainability, including what kinds of system changes are easy to make, and why.

These concerns about change influence a great deal of the design folklore. Many design guidelines originate from fundamental concerns about 'keeping things separated', 'containing design decisions', or 'reuse of components' (Gamma et al., 1994; Martin, 2000) – satisfying these concerns can incur design–time costs, in the hope that they will lower the cost of (possible) future changes.

1.2 Problem

"Should I use an interface here?" – Novice programmer

This simple question could provoke several responses:

- A simple 'yes' or 'no' based on the learning objectives of the exercise
- A dogmatic response e.g. 'always unless you have a reason not to'
- A consideration of the module structure of the program in question
- Referring 'up' to the system design – an interface may be 'mandated' by architecture

- Consideration of other team members and how system implementation has been split in to deliverable work tasks – the interface may be a 'contract' between contributors
- A more general discussion on modularity, and what constitutes a 'good' interface
- An analysis of the likelihood of change
- Most commonly - 'it depends…'

As design questions become more specific, it can be difficult to determine what general guidance applies, and gauge the relative importance of each guideline. This problem is exacerbated by the fact that there can be 'local' concerns that are just as pressing as adherence to the guidelines enshrined in popular textbooks or introductory programming courses. These conflicts may be resolved with experience – however, this can make decisions difficult to justify or record. Issues can also arise when the structures available to a practitioner do not precisely match requirements. This is inevitable as high-level concepts must be translated down to the specific abstractions and low-level primitive structures offered by programming languages.

Where does this leave novice programmers, or those teaching programming? Must each generation of programmers make the same mistakes to gain 'experience'?

1.3 Response

This research aims to determine which design guidance is useful in practice, and how high-level, generic guidance is balanced with low-level, project-specific concerns. Once this interaction is identified, production software will be examined for evidence of these compromises.

Additionally, it is important to consider this problem from the other direction. Given a software artefact, what can be said about its quality? Specifically, with regards to use of programming constructs such as interfaces and inheritance, and compliance with popular design guidance.

The empirical literature is replete with metrics and modelling which can evaluate code to several decimal places. Metrics will always give an answer about the code they are applied to, even where an experienced practitioner may decide to wait for more information. Thus, it is as important to recognise where the cost of change is not, or cannot be justified, as it is to recognise where a change is required.

Research Objective: To close the gap between object oriented guidance and practice.

Research Goals:

A) Collect information on the usefulness of object-oriented design guidance to practitioners.
B) Analyse object-oriented systems for evidence of compliance with design guidance.
C) Develop advice for guideline application or refine existing guidelines based on actual use and practitioner attitudes (rather than argument from design).

Thesis Question: Is it possible to close the gap between object-oriented design guidance and design practice by surveying practitioners and studying how constructs such as interfaces and inheritance are used in open-source systems?

The contribution made by this thesis is that it helps close the gap between object-oriented design guidance and practice. It achieves this by i) discovering how practitioners approach design quality ii) discovering how interfaces are used in open-source Java systems, iii) discovering how inheritance is used in open-source systems, and iv) relating use back to practitioner sentiment and object-oriented design guidelines.

While guidelines are taught and discussed as aspirational standards for design quality, there is no ‘theory of software design’ to help resolve conflicts *between* guidelines. Indeed, misapplication or over-application of guidance is noted as an expensive mistake by practitioners. Furthermore, many of the outliers examined are legitimate departures from what might be expected if guidance was followed by rote. These appear to be created when there is supporting information e.g. some known problem domain property, external dependency, or pattern.

Guidance appears to fulfil two roles: Firstly, to ground high level design principles into actionable practices, evidenced by the adaption of guidance from older programming paradigms to the new. Secondly, to provide a fall-back in the absence of sufficient information for decision making, much like a least-harm principle.

To bring guidance in line with practice requires a recognition that tolerances and norms vary between systems and developers – which makes a universal Entity Population Model (Kitchenham et al., 1995) seem unlikely, but suggests the possibility of local models for specific design elements. Additionally, when applying guidance it is important to distinguish deliberate structure (essential complexity) from accumulations of ‘bad code’ which it is desirable to improve.

Guiding Research Questions:
The thesis question is broken down further:

RQ1.1: What does it mean for software to be of high quality, or well-designed? Can definition(s) of ‘good design’ be derived from the attitudes and beliefs of experienced practitioners, and the structure of mature software systems?

Survey responses were positive regarding the importance of design quality and recognition of principles, but there was less agreement on the meaning of quality, or the importance of the various guidelines. Practitioners indicated that there are limited resources available to address design quality concerns during development. Indicating that rather than constantly being in a pristine state, we might expect systems to be comprised of ‘designed parts’ where there is enough knowledge or confidence spend effort on more design, and ‘everything else’ which is maintained at some ‘non-committal’ level pending further information or time.

The use of important structures such as interfaces and inheritance varied wildly between systems, indicating that while principles may be universal, recognising deliberate design trade-off requires contextual information which may be missing from source code.

RQ1.2: How are programming language constructs used in practice? Specifically, powerful, mature, and ubiquitous abstractions - such as interfaces and inheritance.

The artefacts available to software practitioners are used to satisfy multiple constraints. This thesis studies interface and inheritance structures in ways that are informed by guidance and the role of these structures to gain context-specific insights into their use. This highlighted the complex and varied use of these structures, including identification of sub-groups and frequent ‘non-standard’ usage, indicating more guidance is required for more controversial or pragmatic use of interfaces or inheritance.

RQ1.3: How does practice impact the structure of software? Specifically – is there any evidence within systems that guidance is being followed? Can (or should) systems be fully compliant with all guidance?

There is evidence of guideline compliance – but this required some refinement of terms to classify structures as compliant or not. It was found that many artefacts are not compliant with well-known guidelines, but that there is often a reason for this, which may be evident, or at least hinted at by the source code. These may represent different compromises between design guidelines – if compliance with one guideline can be sacrificed for compliance with another, assessment of compliance with guidelines cannot treat quality aspects independently.

1.4 Contribution

The contributions from each chapter are summarised here.

Literature Review:
The contribution of the literature review is identifying the key motivations underpinning object-oriented design advice. It surveys the progress that has been made trying to operationalise this advice into paradigm- or language-specific guidance or metrics, and the difficulties faced in detecting compliance with guidance from source code alone.

This chapter begins by reviewing the basic concepts of object-oriented (OO) programming, then the popular design guidance, including the origins of the guidance and how guidelines interact. Similar principles underpin many of the popular design guidelines, these high-level motivations are grounded in process concerns such as maintainability, comprehension, familiarisation, software evolution. The popular OO guidelines can be viewed as high-level motivations applied to specific scenarios or programming language constructs.

Next, there is an overview of some of the simpler software quality metrics that relate to structures studied in this thesis. Attempts to operationalise guidance into metrics rarely (if ever) captures their full richness. Metrics have perceived ‘legitimacy’ despite serious questions being raised in the literature about the soundness of some metrics, as well as the applicability of pre-OO guidance to OO designs. The chapter also covers existing design taxonomies based on ‘experience’ or ‘argument from design’. These have been difficult to validate as they incorporate subjective assessment or semantic information, indicating that the way practitioners classify structures depends on information that is not easily recoverable from source code.

Finally, there is a review of recent surveys of software practitioners highlighting that many of these follow the same best-practices, and have similar formats.

Practitioner Survey:

The motivation for a survey is to 'ground' the research in current practice and to begin to answer RQ1.1 (What does it mean for software to be of high quality, or well-designed?). Since this is a broad question, it was given context by focusing on popular guidance and how it relates to the day-to-day experiences of practitioners. Additionally, the survey responses indicate that design quality is a valid concern among practitioners – indicating that there may be some practical benefit to studying design quality.

RQ3.1: To what extent do practitioners concern themselves with design quality?

Design quality is an important concern for the surveyed practitioners. The survey responses indicate that many well-known guidelines are viewed as relevant and useful. Design quality is a persistent secondary concern (around a third of 'effort') behind functional correctness for many respondents. Responses indicated that the roots of design difficulty include: uncertainty about future change, lack of specification, and time pressure. It is not clear how practitioners decide what design work to do under time constraints and what to leave undone.

This highlights a key tension – design advice is valued, and the best advice from peers is seen to be previous experience in the domain. However, design guidance that is more specific (patterns and inheritance guidance) is more controversial due to the potential cost of misapplication.

This being the case – it might be beneficial to present estimated effort/complexity with design compliance recommendations to assist with the cost-benefit analysis practitioners are doing when deciding which guidance to comply with during development.

RQ3.2: How do practitioners recognise software design quality?

Practitioner responses frame quality in terms of the tasks they must carry out. Good design is characterised as comprehensible design that is easy or obvious to navigate, has a low modification cost (is easy to work with). It is notable that these do not map to specific thresholds and appear to be somewhat subjective, based on the individual practitioner and the task they are carrying out.

Some concerns are about future change requirements – since it is not possible to know what design decisions will make future work difficult. Guidance is useful here, where it provides a means to meet a requirement (capture essential complexity), while avoiding unnecessary complexity.

RQ3.3: What design guidelines do practitioners follow?

Simple, generally applicable, 'fundamental' guidance is valued the most highly – small components, loose coupling, good naming and documentation, avoiding unnecessary complexity. These properties are closely tied to the practical experience of developers and are very similar to the underlying principles identified in the literature. Responses indicate that practitioners are mindful of guidance, but that difficulty can be encountered in knowing how to satisfy the guidance and solution domain constraints quickly. Additionally, reports of 'misapplication' of guidance indicates that the more specific guidance (inheritance and patterns) may be incomplete in terms of application criteria.

Notably, these largely depend on a component of practitioners' judgement in a given scenario (e.g. what is 'too complex'). This is raised again when respondents identify that 'essential complexity' in a solution that should be recognised, rather than 'designed away'.

RQ3.4: What information do practitioners use to make design decisions?

Respondents showed a strong preference for personal experience and people-mediated feedback (personal experience, peer-review), with general design guidance, then tooling and metrics following noticeably further behind. This indicates that a primary component of quality assessment is the knowledge of the assessor, and the interactions between practitioners.

Experience is expressed as – specific previous experience with a problem or technology, or disciplined application of guidance. The main body of design guidance provides a 'fall-back' in unfamiliar situations. Conventions such as 'Clean Code' (Martin, 2009) are seen as valuable, and respondents identified a link between testability and high-quality. Rather than minimum standards – it may be better to say that guidance is rooted in a desire to avoid introducing unnecessary constraints which may make future changes difficult. This explains why more information about the problem or solutions domains can justify departing from the default stance provided by design guidelines.

Interface Study:

This chapter examines interface use in open source systems with respect to the last two guiding questions, RQ1.2 (How are programming language constructs used in practice?) is answered by an investigation (RQ4.1, RQ4.2) of the presence and uses of interfaces. The last question - RQ1.3 (How does practice impact the structure of software?) – is more related to analysis of the data gathered and discusses interface use (RQ4.4) as it is found.

RQ4.1: How are interfaces defined in the systems?

The cross-system averages presented in previous work (Tempero et al., 2008) can be verified across systems – but these averages are not representative of any of the systems examined. Interface definition in some of the examined systems is very low, and is not obviously related to system size. Unusual interface use occurs in all systems, indicating that these require specific guidance rather than simply being 'discouraged'.

RQ4.2: What is the profile of interface size across the systems?

Most of the interfaces examined have only a few methods (50% <= 2, 80% <=8). Some uses of interfaces can be segregated by size such as constant interfaces. Overall, size (number of methods) is a very coarse measure of interfaces, and the (long-tailed) distribution makes it difficult to comment on the size of the many interfaces in the 'usual' range (2 – 8 methods).

Examination of size categories revealed the use of naming conventions. Common prefixes and suffixes, while not strictly binding, are common and give high-level feedback about design intent which can convey a lot of information. Very large interfaces usually documented some system boundary or domain feature, the presence of specific external references which must be captured by these interfaces allows cohesion to be checked easily, which may supersede generic size concerns. Again, there is no generally agreed way to annotate such interfaces to this effect.

RQ4.3: To what extent are interfaces implemented in the systems?

Interface implementation is highly variable among systems with a high tolerance for (concrete) types that cannot be accessed via interfaces, even in the most abstract systems. Much of the interface implementation is via inherited interfaces indicating a complex interaction between interface use and class-inheritance.

Around three quarters of types are defined using some form of inheritance, confirming previous findings, this is achieved using different proportions of interfaces and inheritance across the systems. It is not clear how much of this is due to practitioner preference versus, say, problem domain, but it is notable that the degree of definition-by-abstraction is quite regular between systems. Implementation is also directed by architectural and design patterns with evidence that high level decisions constrain the use or form of many interfaces.

RQ4.4: How are interfaces used in the systems?

Interface definition, interface implementation, and type references to interfaces vary independently and are not obviously related to system size. The number of interface-mediated dependencies is highly variable compared to the number or proportion of interfaces in a system – indicating some systems contain easily recognisable, popular abstractions.

Some interfaces are implementation-level components of high-level or architectural decisions such as module boundaries, or specific architectures. These constraints appear to supersede guidance, as they are specific to the component and the surrounding design – echoing the survey response that guidelines are a 'safe default'. This indicates a risk when applying guidance or using metrics based on guidelines – the most extreme outliers may have the most justification for their status compared to an entity that is a 'little out' (due to lack of time or accumulation of changes). This directly conflicts with the normal distribution proposed by the entity population model approach.

Interfaces appear to be used more in the 'public areas' of types – defined as constructor and method parameters, and method return types. Furthermore, practitioners appear to be more inclined to use interfaces when referring to types they have defined in their own code (where they have the most choice). It may be the case that third party code is perceived as 'abstract enough'.

<u>Inheritance Study</u>:
Following on from Chapter 4 and to further investigate the interaction between inheritance and interfaces, this chapter investigates inheritance in a similar way. This chapter relates to guiding questions RQ1.2 (How are programming language constructs used in practice?), and RQ1.3 (How does practice impact the structure of software?). RQ1.2 maps to RQ5.1 which

surveys the inheritance structures that are present, while RQ5.2 and RQ5.3 relate to RQ1.3, characterising inheritance use at a higher level and determining what can be said about guideline compliance.

RQ5.1: How much inheritance is present? To what extent is inheritance used to define types?

Inheritance is common across the systems examined. Inheritance is proportionally more present in larger systems, indicating more opportunity for code reuse/consolidation in larger systems. Most system examined have around three-quarters of types defined using some form of inheritance, but different systems use different ratios of interface use and class inheritance to achieve this.

RQ5.2: How is inheritance is used? How can inheritance hierarchies be characterised?

Much of the inheritance observed is trivial – simple, shallow hierarchies. There appears to be more freedom for hierarchies to grow horizontally, as wide hierarchies are present in even the smaller systems. Hierarchies can be categorised by shape – with many hierarchies falling into simple categories 'line' and 'fan'.

Just over half of the hierarchies are accessed exclusively via their root type (method signatures) indicating that classical 'strict polymorphism' is a common use case. However, some other hierarchies have directly accessed sub-types and novel behaviour, indicating inheritance-for-reuse is also an important use case.

The population is skewed by the small number of very large hierarchies which contain around half of all hierarchy members across the systems examined. These are often representations of some well-defined external structure such as a domain model, or design pattern. Many casting operations involve hierarchy members – indicating that many abstractions are 'local' rather than of system-wide use.

RQ5.3: To what extent can inheritance usage patterns be related to Design Quality where quality is defined via Guidance, Metrics, Code Smells, Entity Population Models, and Modelling?

The hierarchies observed were either far under or far over the suggested limits on size indicating that these thresholds are of limited use for detection of issues. Use of abstract classes as a hierarchy 'skeleton' is common – though appears to be an all-or-nothing choice in most cases – possibly due to practitioner style. This inner structure is highly likely to be contiguous if present, though larger hierarchies show signs of fragmentation.

1.5 Roadmap

The remaining chapters of this thesis are summarised below. Throughout the chapters, key findings are summarised in the form *[KF section number . finding number]* for ease of reference. This improves traceability from supporting findings to conclusions.

Chapter 2 comprises a review of the literature relating to object-oriented design guidelines, focusing on inheritance and use of interfaces. This includes reviews of both published literature, and practitioner-facing material such as experience reports, advisory textbooks, and advocacy. The role and effectiveness of object-oriented metrics are also discussed – including issues which arise from the use of metrics. Existing models of quality for implementation level structures are summarised and compared. Finally, previous survey work relating to practitioner attitude to quality is discussed.

Chapter 3 describes a survey of software professionals about the relevance of design guidance. Specifically, it highlights which of the 'classical' object-oriented design guidelines are regarded as important in practice.

Chapter 4 is a case study of interface use in a corpus of open-source software systems. With a focus on the guideline 'program to an interface', this work aims to build up a picture of the use of interfaces in the wild.

Chapter 5 is a further case study of open-source with a focus on inheritance use. Chapter 4 highlighted a complex interaction between interface use and class inheritance. This chapter focuses on the tangible aspects of inheritance use and examines if these can be related to the abstract 'principles' motivating inheritance guidelines.

Chapter 6 concludes this thesis, drawing together high-level answers to research questions.

Chapter 7 contains all references for this work.

Chapter 8 comprises appendices. Of note are:

Appendix A – Terminology: Terms are defined when they are introduced, it may be of use to refer to this list of terms rather than locating the initial definition.

Appendix H - Structural Patterns for Inheritance Hierarchies: These are based on the proposals and discussion in Chapter 5. The proposed new guidance is discussed in more detail. An initial check for ‘coverage’ of inheritance use in the corpus by the proposed guidance is also discussed.

2 Literature Review

2.1 Introduction

Software underpins an increasing proportion of technology – from critical infrastructure, to everyday services. Poorly designed software manifests in many ways – poor usability, low reliability, incompatibility between systems, and unexpected behaviour. To the software practitioner, the main consequence of poor design is that systems are difficult to maintain and extend.

Poor design of *software intensive systems* has economic impact – with the estimated cost of software faults running into many millions of dollars per year (Charette, 2005).

The goal of good design in software, like other domains of engineering, is to create reliable systems that are easy to maintain and that are composed of reusable components.

The focus of this research is on object-oriented technology because much of today's software is developed using that technology (Cass 2015). While other programming approaches do exist, support for objects continues to exist in many modern and emerging languages, therefore, object-oriented design will continue to be relevant for the foreseeable future.

Software guidance historically comes from practice in the tradition of early commenters from theory and teaching (Dijkstra, 1968) during the early expansion of computer use, and publishing design and engineering manuals based on practice (Coad and Yourdon, 1991).

This continues today with prominent commentators (Beck, 2014; Bernhardt, 2011; Bloch, 2008; Coplien, 2016; Fowler et al., 1999; Gamma et al., 1994; Joel Spolsky, 2005; Martin, 2000) having a great influence over practitioners through training programs, textbooks, social media advocacy, and importantly, public debate about how software should be developed.

With the explosion in the use of computers, emerged a more empirical approach to software engineering research has led in turn to a great deal of work (Basili et al., 1996; Briand and Wüst, 2002; Kitchenham, 2010; Tempero et al., 2013; Zhang and Budgen, 2012) exploring the properties of software, invention and validation of new metrics, modelling reliability, preparation of software corpora, and attempts to 'validate' the more general guidelines and methodologies which have emerged from practice.

The purpose of this literature review is to understand the current state of the art, and limits of measurement for design quality. Additionally, other sources of guidance are considered, such as guidance from industrial commentators, and popular design aids such as code smells and design patterns.

This review will identify the contribution of each of these elements to ensuring design quality, with the aim of identifying weaknesses, gaps, or conflicts among the approaches. These discontinuities in the literature illustrate that some gaps remain between software guidance from practice and theoretical arguments about software construction.

The rest of this chapter covers the relevant research and methodological themes surrounding the topic of design quality in object oriented systems. This begins with a high-level summary of key topics such as object-oriented programming, a discussion of design quality, and the design guidance available to practitioners. The next topics relate to the specific chapters of the rest of the thesis and cover topics such as survey research, interface use, and class inheritance. Each of these major sections narrow to a research topic picked up by a chapter later in the thesis.

Definitions and brief discussion of basic object-oriented concepts is included in *Appendix B - Object-Oriented Concepts* (section 8.2).

2.2 Object-Oriented Design Guidance

Each theme in this section relates to object-oriented technology and ideas about design quality and recognising high quality designs. These themes relate to all subsequent chapters.

Since the arrival of object-oriented technology in the mainstream of software engineering in the 1980s there has been a vast amount of research (Basili et al., 1996; Briand and Wüst, 2002; Jabangwe et al., 2014) investigating definitions of 'good' object-oriented design and the associated identification of techniques and tools to help achieve the desired quality.

The volume and variety of material discussing design quality and best practices is evidence of a widespread interest in the quality of object-oriented software.

2.2.1 Defining Design Quality

The term design quality covers a range of aspects in software design, in terms of the current international standard on Software Quality Characteristics (BSI, 2011) the focus of this work is maintainability:

- Degree to which a system is composed of discrete components such that a change to one component has minimal impact on other components (Modularity).
- Degree of effectiveness and efficiency with which it is possible to assess the impact on a system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified (Analysability).
- Degree to which a system can be effectively and efficiently modified without introducing defects or degrading existing product quality (Modifiability).
- Degree of effectiveness and efficiency with which test criteria can be established for a system, and tests can be performed to determine whether those criteria have been met (Testability).

These aspects of quality are primarily practitioner facing – they affect the cost and difficulty of day-to-day operations on a code-base. It is clear that high quality software is desirable, the following themes cover how quality can be ensured, and the potential costs incurred by this desire.

2.2.2 The Emergence of Design Guidelines

Engineering is often described as *the art of compromise* – and many design problems can be traced back to the decision making of practitioners. Software construction guidelines are suggested constraints, intended to limit design choices. As Martin notes - "*Structured Programming was in what not to do - don't use unrestrained goto. Functional Programming - don't use assignment. Object Oriented Programming - don't use pointers to functions.*" (Martin, 2016). The preferred choices are those that are likely to be less harmful (or more reversible) in the long run.

Much of the guidance about programming and software design has its roots in advocacy or *practitioner folklore* and is very general or abstract. Additionally, the interpretation of this guidance has evolved with the development of programming languages as practices change. Figure 1 illustrates that much of the popular guidance is decades older than the languages the guidance is now applied to. In fact, these guidelines have informed the design of many modern languages.

Note that in Figure 1, the round cornered boxes indicate the first appearance of a language, while the square cornered boxes indicate the first appearance of guidelines in their popular forms. It is notable that the earlier guidance {1970-1980} originates from pre-object oriented practices, while the later flurry of activity {1987-2004} is more specific to OO design.

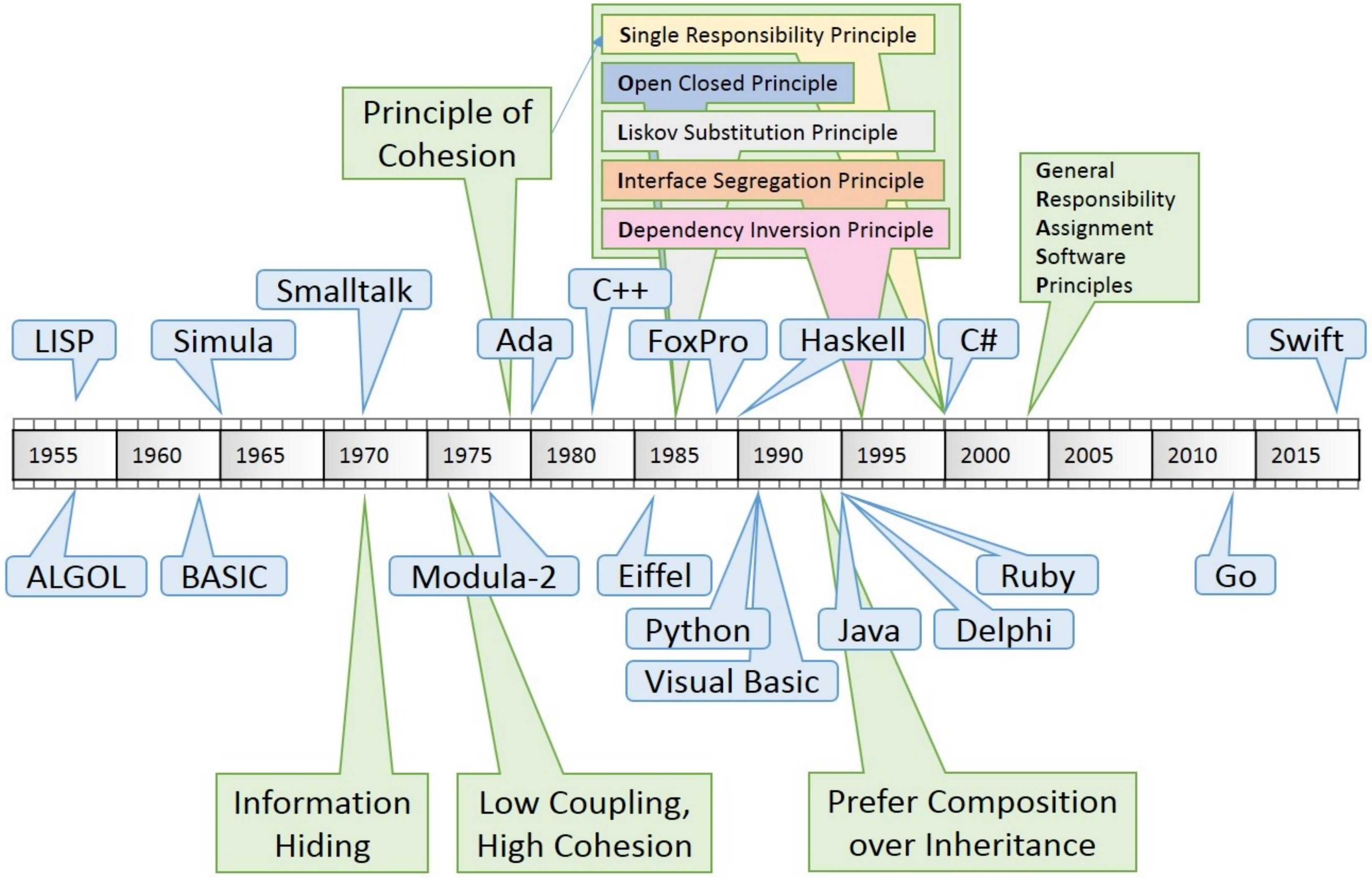
Further refinement of guidance has been taken up by the design patterns community, with many domain-specific patterns being specified since patterns were initially popularised in *Design Patterns* (Gamma et al., 1994). One approach has been the adoption of traditional (pre-object-oriented design) guidelines to the object-oriented domain (Coad and Yourdon, 1991; Riel, 1996).

2.2.3 Dependency Inversion Principle (a.k.a. Program to an Interface)

Martin's Dependency Inversion Principle (DIP) suggests that "*No dependency should target a concrete class.*" (Martin, 2000) i.e. all variables should be defined via interfaces or perhaps, abstract classes. Another prominent commentator, Holub states "*My rule of thumb is that 80 percent of my code at minimum should be written entirely in terms of interfaces*" (Holub, 2003).

If polymorphism is all that is required or there is little or no code re-use to be exploited, then an interface super-type may be preferable to class inheritance. Additionally, the decision to use an interface may be design motivated (top down), rather than code-reuse (bottom up) motivated.

The primary concern of the DIP is that the high-level operations of a system should depend on abstractions, rather than the specific details of implementation. This echoes a more general engineering approach of separating policy from mechanism. DIP creates flexibility by hiding low level details behind abstract types. This guidance sometimes appears as "*depend towards stability*" (R. C. Martin, 1996), where the most stable element is one that has no external dependencies.



Single Responsibility Principle
Open Closed Principle
Liskov Substitution Principle
Interface Segregation Principle
Dependency Inversion Principle
Principle of Cohesion
General Responsibility Assignment Software Principles
Smalltalk
C++
LISP
Simula
Ada
FoxPro
Haskell
C#
Swift
1955
1960
1965
1970
1975
1980
1985
1990
1995
2000
2005
2010
2015
ALGOL
BASIC
Modula-2
Eiffel
Ruby
Go
Python
Java
Delphi
Visual Basic
Information Hiding
Low Coupling, High Cohesion
Prefer Composition over Inheritance

Figure 1 : Timeline: Design Guidance and Languages (dates label the right side of their segment)

When using class-inheritance, the implementation being used (subtype) is hidden from the client; the client is still dependent on another concrete type (the super-type) which can represent a vulnerability to change. This raises the issue of how much behaviour the root of a hierarchy should have and whether hierarchy members should be abstract or concrete.

Some non-abstract dependencies (such as library types) may be perceived as so stable that they will never change, Martin notes that "*Non-volatility is not a replacement for the substitutability of an abstract interface.*" (Martin, 2000). Meaning that *safe* concrete dependencies still reduce the flexibility of dependent code.

2.2.4 Coupling and Cohesion

Coupling and cohesion are two of the most basic principles of software design – these principles are simple to describe, and intuitive to practitioners, but difficult to measure directly.

Coupling is "*the degree of interdependence between modules*" (Yourdon and Constantine, 1975) and is concerned with minimising the complexity of interactions between components. This allows the components to be built, understood, and analysed in isolation from the rest of the system. Liskov discusses this as *locality* (Liskov, 1988).

The cohesion of a module relates to "…*how tightly bound or related its internal elements are to one another.*" (Yourdon and Constantine, 1975). There are many definitions of coupling in modern software engineering literature – an early definition is *"the strength of interconnection"* between the elements of a design – this is not as simple as counting the number of connections between those elements, but must also take account of the *complexity* of the interaction (Yourdon and Constantine, 1975).

Reducing system complexity motivates this property. Grouping highly interacting elements in the same module reduces overall intra-module communication, while exploiting locality in cohesive neighbourhoods. Early work by Briand et al. suggested that long-standing design guidelines by Coad and Yourdon such as coupling and cohesion can have a beneficial effect on the maintainability of object-oriented designs (Briand and Wüst, 2001).

Hall et al. argues that coupling and cohesion are too closely tied to structured programming, and are unsuitable for assessing OO systems. They suggest that OO design has specific mechanisms for defining components and their relationships (Mayer and Hall, 1999a). This may go some way to explain the plethora of cohesion-related metrics – there is a lack of agreement on how cohesion should be defined at the implementation level in OO design.

Coupling and cohesion are often listed together as they are highly related e.g. a system with cohesive modules is likely to have low coupling between modules. However, this is often a compromise and inheritance can complicate matters. As super- and subtypes may have different responsibilities – the case could be made that the new (or modified) behaviour in the subtype does not belong in the super-type (it reduces cohesion). To preserve the cohesion of the parent type, the new functionality is placed in a sub-class, some coupling to the subclass is tolerated. For example – in Figure 54 (8.2.3 Appendix B) the new and overridden methods in the new subtype (*PausableAverage*) do not belong in the super-type – these would bloat the interface and change the behaviour of that type.

It has been argued (Bloch, 2008) that implementation inheritance breaks encapsulation as implementing parts of a sub-class may require knowledge of that super-class's implementation. This is a form of coupling between parent and child. The argument in this case is that this is an inappropriate relationship that *may* cause problems and *can* be avoided, so it *ought* to be avoided. The other side of this trade-off is the effort saved in implementing the subtype using inheritance.

2.2.5 Information Hiding

One of the earliest discussions of object-oriented-style program decomposition is by Parnas, who proposes criteria to decompose the task of constructing a system into *"independent programming of small, manageable programs"* (Parnas, 1972).

The key idea introduced by Parnas is *information hiding* where each component of a system hides one design decision. Since it is not possible to foresee which design decision might be revised, each decision should be isolated from the rest of the system. Parnas contrasted this with more conventional decompositions where each component of a program represents one stage in a process - with implicit dependency on the output of the previous stage(s) and the assumptions of the next stage(s).

Parnas argues that fundamental behavioural units (e.g. parsing, entity representation, algorithms) are likely to persist, so should be in some sense separate or atomic, and thus interchangeable and reusable.

The key role abstract classes and interfaces can play in information hiding is that they allow the definition of a type for use by the rest of the system, without specifying how that type is implemented. Thus, protecting the rest of the system from any change in implementation.

It is implied though not explicit, when information hiding, that the abstraction describes a complete type of object such as a data structure or sub-module. Parnas explains that a module is *"characterized by its knowledge of a design decision which it hides from all others"* (Parnas, 1972).

According to Beck, *"…the costs of changing software are dominated by rippling changes"* (Beck, 2014). This statement captures a persistent issue in the design of software systems – how to maximise the beneficial relationships between elements (their usefulness to each other), while reducing the dependencies between elements (their need to change together).

By following information hiding principles, the implementing module should not hide multiple design decisions. This means that the hiding interface is *complete* in the sense that the implementer of the interface exists only to implement the behaviour described by the interface. The primary motivation for information hiding is to protect the system from the evolution of any one component over time. Information hiding in its original form is more concerned with system evolution (incremental modification) and curbing change propagation than polymorphism.

2.2.6 Favour Object Composition Over Class Inheritance (Gamma et al., 1994)

Object composition is strongly advocated in Design Patterns - "*Object composition is an alternative to class inheritance. Here, new functionality is obtained by assembling or composing objects to get more complex functionality.*" (Gamma et al., 1994) This is sometimes described as *has-a* behaviour in contrast with the *is-a* model of inheritance.

Figure 2 shows an example of composition – in contrast to the inheritance example shown in Figure 54 (8.2.3 Appendix B). *DelegatingPausableAverage* is a class definition which has a *clear()* method, this method is similar to that offered by the delegate *Average* object. The format of the method which passes through to the delegate is similar to an *override-then-upcall* method we may see in a subtype of *Average*. The main differences from class inheritance are that *DelegatingPausableAverage*:

- Is not required to be substitutable for *Average*
- Only uses the delegate behaviour it requires
- May have multiple delegates

```
class DelegatingPausableAverage {
    private Average avg;
    public clear () {
        ...
        avg.clear(); // delegated behaviour via internal reference
}
```

Figure 2 - Delegation

In Design Patterns (Gamma et al., 1994), it is proposed that the use of inheritance is problematic because *"...parent classes often define at least part of their subclasses' physical representation".* This means that, regardless of the amount of useful re-use gained from inheritance, hierarchies can become problematic as they grow. As more members are added to a hierarchy, there may be more super-types involved in the definition of the subtypes – increasing the risk of dispersed code, complex execution paths, and ignored inherited behaviour. The root and other depended-upon types become more expensive to change as they define the behaviour of many other types.

The designer of the Java language has expressed concern about the presence of inheritance – "*It's not so much that class inheritance is particularly bad. It just has problems ... delegation can be a much healthier way to do things. But specific mechanisms for how you would implement that tend to be problematic*" (Venners, 2001). It is not clear if this is a reference to the ease of using the *extends* keyword for class inheritance compared to the lengthier process of implementing a composition relationship. The difficulty of design is perhaps best highlighted by the later comment, regarding when to use inheritance vs. composition – "*I just wish I had some good rules because it always gets kind of vague for me. I personally tend to use inheritance more often than anything else".*

The Design Patterns manual also acknowledges that inheritance has some valid uses and that inheritance and composition are complimentary - "*Almost all patterns use inheritance to some extent ... Structural class patterns use inheritance to compose classes* [e.g. adapter, composite] *... Behavioral* [sic.] *class patterns use inheritance to describe algorithms and flow of control* [e.g. interpreter, template method]..." (Gamma et al., 1994).

'Proper use' is not described in detail, but it is noted that - "*When inheritance is used carefully (some will say properly), all classes derived from an abstract class will share its interface. This implies that a subclass merely adds or overrides operations and does not hide operations of the parent class.*" (Gamma et al., 1994). This prioritisation of substitutability for the parent type and addition (rather than removal) of behaviour are indicative that super-types should be abstract in some sense.

Table 1 illustrates the trade-off between inheritance and composition. Inheritance requires a fixed concrete dependency on another type (the super-type), but what is shared (inherited) can change as the super-type changes. On the other hand, composition allows the target of the dependency (the delegate) to change, but what is shared is fixed by the manual calls to that delegate.

These differences indicate that inheritance and composition are two separate design constructs – but are compared – because both inheritance and composition allow reuse of existing code. While use of inheritance to model *is-a* relationships is encouraged, this is a semantic relationship and there is no way to determine mechanically if an inheritance relationship is *appropriate* in this way.

Feature	Inheritance	Composition	Overall
Implementation	Extend a super-type (a little code)	Composition must be *wired up* (possibly lots of (simple) code)	Modern IDEs support both sets of operations automatically
Benefit	Automatically gain access to all inherited behaviour	Gain access to a specified subset of behaviour	Super-types share new behaviours by default; delegates do not
Flexibility	Stuck in inheritance hierarchy at compile time	Delegate can change at runtime	Intrinsic *is-a* versus transient *has-a*
Dependencies	A leaf may (indirectly) depend on several super-types	Dependency is usually direct from client to delegate	Many dependents may make a delegate/super-type hard to change
Perceived Value	Useful for reuse, incremental modification, polymorphism	Seen as less harmful, may involve more effort, no free polymorphism	Inheritance misuse gets more 'bad press' than composition misuse

Table 1 - Inheritance Vs Composition Trade-offs

2.2.7 SOLID and GRASP

High level OO guidance is proposed in the SOLID principles. Martin argued that "*… dependency management, and therefore these principles, are at the foundation of the '-ilities' that software developers desire*" (Martin, 2005a). The key principles associated with SOLID are:

- Single Responsibility (SRP) - A class should have only one, reason to change.
- Open-Closed (OCP) – Allow for extending a class's behaviour, without modifying it.
- Liskov Substitution (LSP) – A derived class must be substitutable for its base class.
- Interface Segregation (ISP) – Make fine-grained interfaces for client-groups.
- Dependency Inversion (DIP) - Depend on abstractions, not on concrete classes.

As well as the SOLID principles, Martin advocated a wide range of Clean Code principles (Martin, 2009) which include a wider range of advice focussing on both design and code intended to make programs (and their designs) easier to understand and change. This advice covers: package and class design, dependency management, method design with recurring themes of small size (e.g. classes less than 100 lines, methods less than 20 lines and much shorter), simplicity (e.g. methods do one thing), and clarity (e.g. descriptive naming, minimal value-added commenting).

This guidance is unusually prescriptive and makes no claim to be based on anything other than *time served* in industry. It is interesting to note that these constraints synergise, for example, if a class is small, it is less likely to violate SRP.

In a similar vein, Larman identified the GRASP principles (Larman, 2004) which were intended to capture the knowledge of expert designers to help in the construction of better quality object-oriented designs. GRASP consists of nine principles, some of which are closely related to the concepts covered above.

Conley and Sproull used aspects of Martin's SOLID principles to define modularity (Conley and Sproull, 2009). Using static bugs and complexity as proxies for quality, they found a decrease in complexity but also an increase in the number of static bugs detected as their measure of modularity increased.

2.2.8 Interface Segregation Principle

In a similar vein to information hiding, Martin proposes the Interface Segregation Principle (ISP) – namely that "*Many client specific interfaces are better than one general purpose interface.*"(Martin, 2000) The principle notes that interface changes required by one client or group of clients may lead to unnecessary re-compilation of all clients using that interface.

A close reading of the original ISP material indicates that the ISP is concerned with separating sources of demand for change (Martin, 2000). If there are separate groups of clients of an interface, a reasonable concern is that these requirements of these different groups may diverge as a system evolves. Martin argues that introducing *segregated* interfaces avoids having to handle diverging demands in a single interface. This is clarified by the comment that "*If the interface for ClientA needs to change, ClientB and ClientC will remain unaffected.*"(Martin, 2000).

Consequently – a balance must be struck between having an individual interface for *each* client (which the ISP also discourages) and having a single interface for *all* clients. The Interface Segregation Principle (ISP) advocates identifying groups of similar clients of large interfaces or classes and providing interfaces for each group. This ensures that the interface provided for each group of clients can vary independently, *hiding* the fact (from the clients) that there is a common implementer. The ISP is concerned with identifying and segregating different subsets of interactions or protocols.

This is reminiscent of *role* interfaces proposed in the DCI design model (Reenskaug and Coplien, 2009) where roles are derived from object interactions, not as a permanent property of a particular object.

There appears to be a common misconception in the literature that the aim of the ISP is reduce client access to interface methods that they do not invoke (Abdeen et al., 2013a; Romano et al., 2014), or that interfaces should serve a single client (Abdeen et al., 2013b). The belief is that the client will be *broken* if the unused method definition changes. This is incorrect in Java - the dynamic binding in Java means that if an unused method in an interface changes, the client (which does not invoke the interface method) will not be affected (Eisenbach and Drossopoulou, 2002).

2.2.9 Is-a Relationships and the Liskov Substitution Principle

The importance of conceptual similarity (Liskov, 1988) and the difficulty of arranging classes into hierarchies (Dvorak, 1994), mean that the benefits (of re-use) may be out-weighed by the potential costs (e.g. rigidity, complexity).

Brachman discusses *is-a* relationships in the knowledge representation field – he notes that the concept is vaguely defined and often conflated with the mechanism of inheritance. He asserts that inheritance is merely an implementation detail to reduce duplication in implementation which allows designers to - "…*distribute 'properties' so that those being shared were stored in the hierarchy at the place covering the maximal subset of nodes* *sharing them.*" (Brachman, 1983). The author also notes that the use of inheritance may make these systems more efficient, but it does not make them more expressive.

Inheritance alone is not sufficient to capture the many different types of semantic relationship (and proposed metadata) discussed by Brachman. There is great deal of modelling information from a problem domain that may inform the decision to use inheritance, but not be captured in the final design. There is considerable debate in the design literature (Bloch, 2008; Brachman, 1983; Liskov, 1988; Meyer, 1996; Venners, 2002; Weck and Szyperski, 1996) as to whether inheritance use should be used freely for code reuse, or if further constraints should be observed - such as some semantic or conceptual relationship between the elements of a hierarchy. This non-structural aspect may not be easy to reassess each time any change is made to a hierarchy. In practice, the different motivations for use of inheritance may be difficult to distinguish solely from the source code.

The Liskov substitution principle (LSP) is a guideline that is directly relevant to the structure of inheritance hierarchies. "*If for each object O_1 of type S there is an object O_2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when O_1 is substituted for O_2, then S is a subtype of T*" (Liskov, 1988). Informally, it states that all subtypes of a type should be substitutable for that type and for each other. Furthermore, this substitutability should be both logical and semantic. As shown in the above quote, it is one of the few guidelines based on a semi-formal definition.

Logical substitutability can be checked by a compiler and simply requires that a subtype comply with the syntax of the programming language. This does not enforce sematic substitutability, which is discussed further in section 8.2.4.1 (Appendix B). In terms of inheritance this means ensuring that overridden methods have the same method signature.

Semantic substitutability, on the other hand, requires that inherited methods must "*do the same things*" as those defined in the super-type (Liskov, 1988). Taivalsaari recognises that "*is-a is a conceptual specialization relationship: it describes one kind of object as a special kind of another*" (Taivalsaari, 1996). For example – if a module relies on a data structure (say, Collection) and that structure may be freely substituted between a Set and a List (logical compatibility), the program behaviour may be different due to the different treatment of duplicate items by a Set. This is a semantic mismatch, the super-type does not (and cannot) constrain the overridden operations of the subtype, even if a "*using program* *is likely to notice the difference between these two types*" (Liskov, 1988). At best, the designer of the super-type could forbid sub-classing.

Note that there may be applications where Set and List might be used interchangeably without negative effect – semantic substitutability is highly dependent on context and problem domain. Liskov identifies three modes of inheritance use:

- Convenience inheritance - a sub-type is created simply for reuse with no intention of substitutability.
- Relationships identified in advance - characterised by a virtual or abstract super-type in anticipation of a *"family of related types"*. In this case an interface might be defined before any of the implementing types are considered.
- Grouping approach - Represents some common ability or feature in a design, that is may be implemented by many other types e.g. Comparable, Enumerable. Rather than pollute existing type hierarchies, a polymorphic module can *"…use any type that supplies the needed operations. In this case no attempt is made to relate the types."*

These represent different design needs, for example, it is unlikely that convenience inheritance would be a desirable option at design time, where creating or extracting an appropriate abstraction is 'cheap'. Contrast this with the grouping approach, which identifies a role or common interaction in a design which is important enough to be recognised as a type.

2.2.10 Single Responsibility Principle

The Single Responsibility Principle (SRP) maintains that classes ought to have one responsibility – "*A class should have only one reason to change*" (Martin, 2005b) - usually some well-defined domain or implementation detail. There are no definitive criteria for identifying a *single responsibility* other than a single *reason to change*. This is similar to Parnas' oft-cited suggestion that each important design decision should be in its own module (Parnas, 1972).

When using inheritance, there is often a difference in the *responsibility* of the parent from that of the child type. This guideline captures the notion that "*each responsibility is an axis of change*" (Martin, 2005b) – assuming too many responsibilities may lead to conflicting demands on the direction of a class's evolution.

One way to prevent the bloat of a class's responsibility is to have a delegate or sub-class assume the related responsibility. SRP may be used to justify a trade of simplicity (decluttering a super-type) for some coupling (between the super- and subtypes). The main difficulty presented by this guideline is "*living with uncertainty*" (Beck, 2014). In relation to inheritance – it may be unclear when a common super-type should be extracted from similar classes.

2.2.11 Open-Closed Principle

This principle proposes that systems are constructed so that new behaviour can be added, while leaving the original behaviour intact. "*Software entities (classes, modules, functions, etc) should be open for extension, but closed for modification.*" (R. Martin, 1996) paraphrasing Bertrand Meyer. This approach to system evolution attempts to minimise changes to existing components, thus minimising the introduction of new defects.

Inheritance is good candidate mechanism for compliance with this guideline as it allows, not only re-use, but also run-time substitutability of new subtypes for their super-types. While this guideline does not introduce any new mechanism, it does propose that there is a value in class-inheritance-style extension.

2.2.12 Code Smells and Design Patterns

Design patterns and code smells represent recorded experience of practitioners. Design patterns "*capture solutions to specific recurring problems in object-oriented design*" (Gamma et al., 1994). Similarly, anti-patterns are recognised solutions to common problems which are known to be likely to cause further problems i.e. a poor trade-off. These heuristics provide a common vocabulary to discuss design in more detail. Code smells are heuristics which may indicate an underlying problem in a design.

Over the last two decades the idea of design patterns has become widely established as a source of object-oriented design guidance. Design patterns present generalised solutions to common design problems, making use of design techniques such as interfaces, abstract types, inheritance, dependency inversion, polymorphism, and double dispatch to create "*simple and elegant*" (Gamma et al., 1994) design solutions. The original catalogue consisted of 23 patterns whose names and intent are now widely-recognised in the design community. Design patterns are derived from two fundamental design principles:

- *Program to an interface, not an implementation* – Do not declare variables to be instances of concrete classes, commit only to interface types.
- *Favour object composition over class inheritance* – Rather than using inheritance to define a subclass, use the parent class to define a field and delegate to it.

The authors of Design Patterns emphasise that all objects have an interface *("all signatures defined by an object's operations"*) and claim that ignorance of implementation specifics *"greatly reduces implementation dependencies"* to the point where they advocate abstracting all object creation via creational patterns. This is the source of one of the most quoted design principles - *"Program to an interface, not an implementation"* (Gamma et al., 1994).

This use of polymorphism recognises and manages the possibility that a change in implementation may take place at runtime to improve flexibility and reuse. This is reminiscent of early ideas on OO languages where message passing was emphasised (Kay and Ram, 2003).

In contrast to empirical investigation into guidelines such as coupling and cohesion, and their associated metrics, there have been relatively few empirical investigations of Clean Code, GRASP, and design patterns. These principles and patterns capture the experience of recognised experts in the field, such as Fowler (Fowler et al. 1999), the 'Gang of Four' (Gamma et al., 1994), 'Uncle Bob' Martin (Martin, 2009), Brooks (Brooks, 2010), Booch (Booch, 2011), Beck (Beck, 2014). These are derived from decades of experience, both actively working in the field and from challenging and debating other experts within the software engineering community. Much of this literature is largely in the form of advocacy (Zhang and Budgen, 2012) with little in the way of formal evaluation in the published literature.

2.2.13 Interface Related Guidance

While general design guidance has been discussed above, this section discusses how the guidance applies specifically to interface use. Interfaces might be characterised as 'descriptions without form' which can be used to simplify and generalise implementation. Since this is the aim of many design guidelines, interfaces can be used to meet these goals effectively.

*Programming to an interfac*e allows a client to specify the minimum functionality that the implementer must expose for the client to fulfil its specified role. In this mode of use, it would be reasonable for a larger class to implement more than one interface, to facilitate different subsets of interactions without exposing all of its functionality to each client (Riehle, 1996). By compartmentalising functionality in this way, the clients only directly depend on the existence of an interface. This also relates to the ISP and roles where *contracts*, *messages* or *protocols* represent the 'complexity of interaction' between objects.

While Parnas' original proposal to 'hide information' (Parnas, 1972) was language-agnostic, interfaces provide a mechanism to specify module boundaries to hide implementation decisions. This works in two directions - from the client's point of view, a small reasonable interface for a client or group of clients complies with ISP, while an implementer of that interface may define its responsibility separately from the interface(s) it implements, complying with SRP – use of interfaces allows these two factors to vary independently.

If DIP is also considered, and implementers of interfaces are being *injected*, then the implementer of a given interface can be changed at design, implementation, or run time without affecting the client. This shows how interface use enables polymorphism/substitutability of the interface implementer.

While initially devised in relation to class-inheritance, the LSP can be discussed in terms of interfaces. Implementers of an interface must comply with the logical substitutability criteria to compile. However, semantic substitutability cannot be guaranteed by the implemented interface. Of the other uses for inheritance identified by Liskov, "*grouping*" is an excellent fit for interfaces as it relates to supplying needed operations by possibly unrelated types. Relationships "*defined in advance*" may be suitable for interfaces as they do not require an implementation to have been defined (Liskov, 1988). This is the same approach to use which would support DIP compliant designs – by defining high level operations in terms of abstractions.

Design guidance relating to interface use highlights some key concerns. First there are arguments *against* rigidity and coupling. Concrete dependencies allow implementation changes to *ripple* through a system, and allow multiple sources of change to impose conflicting demands on a single element. On the other hand, there are arguments *for* flexibility. Programming with abstract types enables flexibility via composition of objects at runtime.

It is interesting to note that few guidelines mention the properties interfaces should have, rather they capture concerns at the design level - how objects interact, and how much objects *know about each other*, what dependencies should be interface-mediated. Guidance on the specific characteristics of interfaces require information about the purpose of an interface and its use by the surrounding design. While these properties could be documented – it more common that they are reconstructed by practitioners on each inspection.

2.2.13.1 Interface Design Criteria

Hoffman proposed criteria for designing interfaces shown in Table 2 (Hoffman, 1990). It is interesting to note that these properties require different frames of reference to properly assess, some of which are more amenable to automation than others.

Consistency is a view at the level of the entire design, and might be enforced by coding standards to an extent, such as the Clean Code (Martin, 2009) practices related to matters such as number of parameters, use of exceptions, and self-documenting code.

Table 2: Interface Design Criteria (Hoffman, 1990)

Property	**Detail**	**Related Design Principles**
Consistent	Follows local design conventions	Comprehensibility, Principle of Least Astonishment[1]
Essential	Omit needless features	SRP, High Cohesion, Maintainability
General	A complete abstraction	High Cohesion, Reusability
Minimal	Separate different services/features	ISP, Reusability
Opaque	Hides implementation detail	Information Hiding, Low Coupling

Essential and *Minimal* are more related to identifying the main purpose of the interface, than translating this to how the design represents the domain. For example – an interface representing a file system may have relatively few high-level operations such as *open*, *read, write* and *close* while another may have very low-level operations concerning bit-level encoding and disc sectors. These interfaces represent a similar concept, but the specifics of the problem domain mean that the essential behaviour of the low-level interface may be larger while still being considered minimal.

The *General* property, must be considered in terms of needs that may be met elsewhere in the design, or even in a product family if designing a framework, library, or API. The wider

[1] "*Novelty raises the cost of a user's first few interactions with an interface, but poor design will make the interface needlessly painful forever*" (Raymond, 2003)

this net is cast; the more difficult generality may be to achieve or decide upon. This is reminiscent of the issue of architectural mismatch raised by Garlan et al. who found that implicit local assumptions can make reuse difficult (Garlan et al., 1995).

In addition to information hiding, *Opaque* may also relate to what types the interface itself should depend on – ideally abstract types should themselves only depend on primitive types and other abstractions to completely hide implementation choices – this is covered in part by the Dependency Inversion Principle discussed previously.

In summary – there is a large amount of argument from design or experience about how interfaces ought to be used, while there is some discussion of *what* interfaces should be like, much of the commentary relates to *where* and *when* interfaces should be used. This indicates that interfaces may be more relevant to relationships within a design, rather than the actors in those relationships. This fits with the notion that interfaces allow practitioners to program and *design by contract* (Martin, 2000).

2.2.14 Inheritance Related Guidance

While some general guidance relates to inheritance use indirectly, this section looks at guidelines, specifically as they relate to inheritance. The most relevant general guidance for inheritance is Don't Repeat Yourself (DRY), Favour Composition over Inheritance (Program to an interface, PTAI), Liskov Substitution Principle (LSP), and Single Responsibility Principle (SRP).

These principles are relevant to inheritance as inheritance permits reuse of implementation via class inheritance which reduces duplication (DRY) and inheritance hierarchies can be arranged such that each step in the hierarchy represents a specific change (SRP) such as addition or modification of behaviour. LSP is written specifically for inheritance and captures the problem that there must be logical *and* semantic substitutability among members of an inheritance hierarchy. Finally, PTAI reminds practitioners to consider when inheritance may not be a suitable option, although if composition/delegation is chosen instead, the delegate may itself be an inheritance hierarchy.

2.2.14.1 Hierarchy Shape

Inheritance hierarchies are directed acyclic graphs, while there are some construction rules, there is no limit on size. The overall shape of a hierarchy is changed whenever a new sub-class is added.

As inheritance hierarchies grow and the opportunity for reuse increases, the number of classes that may have to be examined to understand the behaviour a *leaf* class increases i.e. all the super-classes of that leaf class. This is sometimes referred to the *yoyo effect* (Taenze et al., 1989) where the practitioner must travel up and down a hierarchy to trace program execution. This may be especially true if the leaf or any of its parents *override* behaviour or self-calls (*this* in Java). It is perhaps for this reason that guidelines suggest that hierarchies remain shallow as practitioners may find larger hierarchies increase *design complexity* and make it *difficult to predict* behaviour (Chidamber and Kemerer, 1994).

Collberg et al. found "*six of our classes are at depth 30–39*" in a study of Java bytecode, which is extraordinary compared to the ranges for DIT reported in other studies (Collberg et al., 2007). Unfortunately, there is no note of what system contained this very deep inheritance structure or what it was used to represent. While extreme, even this very high depth does not automatically mean that this structure is a poor choice – it may be highly dependent on context or problem domain

Liskov notes that the use of inheritance does not excuse poor modularity, noting that "*Locality allows a program to be implemented, understood, or modified one module at a time*" (Liskov, 1988). Other commentators have few reservations about deep hierarchies, Johnson and Foot note that "*Class hierarchies should be deep and narrow*" to maximise reuse, and further that "*A shallow class hierarchy is evidence that change is needed, but does not give any idea how to make that change*" (Johnson and Foote, 1988).

A more balanced proposal by Riel is that "*In theory, inheritance should be deep — the deeper, the better" "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.*" (Riel, 1996). This indicates a recognition that some uses of inheritance cannot meet the ideal case of maximising reuse, because depth has side effects such as maintainability implications.

Binkley and Schach carried out a comparison of high level system metrics with expert assessment of 'cognitive complexity'. Of the inheritance-related metrics assessed, Width of Inheritance Hierarchy (WIH) was more reliably in agreement with the expert assessment. While the structural metrics performed poorly compared to dependency and encapsulation metrics, the authors note that "*the five metrics that performed the worst are also coupling-based metrics"* (Binkley and Schach, 1996).

In summary, hierarchies grow due to the desire to reuse code and program with generalisations, but as hierarchies grow, they may become more complex, which affects maintainability. Complex calls and patterns of overriding within a hierarchy are separate from simple size concerns. It is unclear where the balance between the pressures of utility and comprehension should be – however, most sources of guidance err on the side of comprehensibility (Kelly and Buckley, 2009) over maximising reuse.

2.2.14.2 Fragile Base Class

The fragile base class (FBC) problem occurs when a change to a super-class may change the behaviour of a sub-class in an unexpected way. This is a problem because designers desire structures that can be modified in isolation without affecting the behaviour of other classes or modules – this sentiment underpins much of the design guidance (ISP, DIP, Information Hiding, PTAI). The cause of FBC is the automatic propagation of changes from super-type to subtype(s) – the very mechanism which makes class inheritance useful.

Holub describes FBC as "*the concept of coupling applied to inheritance*" (Holub, 2003) – warning that changes to a base class (super-type) must be verified by checking "*all code that uses both base-class and derived-class objects too, since this code might also be broken by the new behaviour*". This is a concern that the presence of sub-classes violates the encapsulation of a super-class and removes the benefit of modularity/locality. Bloch notes that inheritance poses several design risks, including this close relationship between super- and subtype, advising that practitioners should "*Design and document for inheritance or else prohibit it.*" (Bloch, 2008)

Also known as the ripple effect (co-change, change impact) – "*the extent to which a change to a software component can affect other component of the software*" (Bajeh et al., 2014). This concern touches on many other guidelines which endeavour to limit the likelihood of changes to one part of the system cascading to adjacent modules (the *ripple effect*).

However, when the FBC problem was examined, it was found to be common – but benign where present, in terms of change- and fault-proneness. Additionally, a survey of practitioners (41 responses) indicated that FBC was not responsible for any inheritance-related bugs reported in the systems examined, although some were "*wrongly identified as such*" by practitioners (Sabané et al., 2016). This indicates that awareness of this problem perhaps exceeds the risk presented, once again raising the issue of what guidelines are relevant under which conditions.

2.2.14.3 Use of Abstract Classes

Abstract classes can be used to consolidate common behaviour in hierarchies. This is an example of compliance with the dependency inversion principle (DIP) (R. C. Martin, 1996) where deeper hierarchy members depend on abstract types, rather than concrete classes.

Steimann and Mayer comment that abstract classes should only be used (in place of an interface) if the relationship to the subtype is "*genetic, i.e., if it is (or at least could be) based on the inheritance of internal structure, that is, implementation.*" (Steimann and Mayer, 2005). This advice discourages coincidental or convenience reuse of code, noting that shared implementation should reflect essential common behaviour. If followed, the presence of abstract class root types would indicate a strong semantic relationship between the root type and its sub-types.

Johnson and Foote recommend that extra effort should be taken to find and extract common behaviour: "*An obvious way to make a new superclass is to find some sibling classes that implement the same message and try to migrate the method to a common superclass. Of course, the classes are likely to provide different methods for the message, but it is often possible to break a method into pieces and place some of the pieces in the superclass and some in the subclasses*" (Johnson and Foote, 1988). Note that this approach might be considered aggressive or unnecessary refactoring where the proposed intermediate abstract classes may have no use other than to act as prototypes or repositories of (coincidentally) common behaviour. The GRASP principle 'Pure Fabrication' is a more general form of this advice – "*something made up, in order to support high cohesion, low coupling, and reuse*" (Larman, 2001). This does suggest that the justification for this kind of abstract class may originate entirely from implementation details.

Conversely conventional modes of use may also be 'obvious' during assessment. For example – the Observable class in the Java standard libraries is a common superclass in event-driven systems, however this is a (often maligned) technical decision in the language design. Extending Observable tells us very little about the nature, behaviour, or responsibilities of the subtype – and the scope for further meaningful extension is low. However, there is a clear reason for the extension to exist, which can be a valuable navigational aid when reading a design.

2.2.14.4 Interaction Between Guidelines and Inheritance

Table 3 shows some design guidelines and notes how class inheritance interacts with them. Guidelines may be given different priorities and relative importance by different practitioners in different circumstances – so it is not possible to give a general order of priority or weighting to the guidelines. As such, it may be difficult to determine the relative importance of these aspects as a design changes.

Table 3 - Interaction of inheritance and classical design guidelines

Guideline	Summary	Effect of Inheritance
Modularise (Parnas, 1972)	Hide design decisions with modularised components	Implementation of super-classes may be exposed to their children, FBC problem
Localise Functionality	Allows understanding and construction of a program using local reasoning (Liskov, 1988)	Behaviour may only be discernible by examining super-classes, abstract class intermediates capture common behaviour
Don't Repeat Yourself (Hunt and Thomas, 1999)	Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.	Common code can be collected in (abstract) super-types
Single Responsibility	Objects should do one thing well (Martin, 2005b)	Inheriting type or behaviour, and super-type's 'responsibility', a subtype contains only what varies from its super-type
Depend on Abstractions (R. C. Martin, 1996)	Depend towards more stable, abstract parts of a design	Leaf-types depend on abstract roots and intermediates. Clients depend on root-abstractions.

2.2.14.5 Inheritance Code Smells

Code smells are *"...structures in the design that indicate violation of fundamental design principles and negatively impact design quality."* (Suryanarayana et al., 2015) These were popularised by design guidance in Refactoring (Fowler et al., 1999), and have become part of the body of advice for practitioners.

The motivation behind code smells is that, if a practitioner is unhappy with some aspect of a design, this may reflect some underlying design problem – *"...no set of metrics rivals informed human intuition."* (Fowler et al., 1999) The authors go on to provide a set of smells to guide a concerned practitioner towards candidate re-factorings to address the underlying design issues.

Tufano et al. found that some code smells appear with the addition of new design elements or behaviour modifications and are associated with high workload and release pressure (Tufano et al., 2015), indicating that code smells highlight a trade of quality for speed.

Code smells are discussed in relation to inheritance by Suryanarayana et al., the authors identify *enabling techniques* for inheritance from their experience (Suryanarayana et al., 2015):

- Apply meaningful classification – capture common and variable behaviour
- Apply meaningful generalisation – reduce duplication
- Ensure substitutability – comply with Liskov Substitutability
- Avoid redundant paths – reduces complexity, clarifies relationships, avoids runtime issues
- Ensure proper ordering – comply with Dependency Inversion Principle

The inheritance-specific code smells are also identified:

- Missing Hierarchy – use of conditional logic to manage variation in place of polymorphism
- Unnecessary Hierarchy – a flag or variable could replace the need for subtypes
- Un-factored Hierarchy – redundant code in a hierarchy, duplicated horizontally or vertically
- Wide Hierarchy – a type in the hierarchy has more than 9 subtypes
- Speculative Hierarchy – a type is provided based on an imagined, not actual, need
- Deep Hierarchy – hierarchy is excessively deep (rule of thumb > 6 is too deep)
- Rebellious Hierarchy – overriding methods restrict or cancel inherited behaviour
- Broken Hierarchy – the parent and child types do not share an *is-a* relationship
- Multipath Hierarchy – a subtype inherits from a super-type directly and indirectly
- Cyclic Hierarchy – a super-type depends on its subtype

These smells show how high-level guidance can be made more concrete if it is with respect to a specific structure (in this case, inheritance). A further benefit of this *grounding* is that some cases are identified where action can always be taken - namely Unnecessary Hierarchy and Speculative Hierarchy. However, the requirement for subjective assessment is not eliminated – there is no guarantee that a code smell is caused by underlying design flaw, the nature of code smells is that they may simply indicate a deliberate design trade-off.

2.3 Metrics

Closely related to these guidelines is a large body of work defining metrics that aim to measure properties associated with these design guidelines. Metrics rely on the fact that we can examine the various artefacts produced during the software design process. Metrics are attractive as they require little or no judgement to apply, and can be applied to large code bases cheaply (compared to manual inspection).

Software metrics propose to provide insight into the quality of software using only features of the software. Static metrics examine source code, while dynamic metrics assess runtime behaviour - this approach is attractive for several reasons:

- Numerical summaries present a notionally representative, more manageable data set than manual examination of source code, especially in large systems.
- Expensive manual processes can be targeted via metrics at critical or high-risk areas.
- The properties that concern practitioners are often subjective or non-functional (e.g. complexity, cohesion) - metrics are a best effort to isolate this uncertainty.

A widely-used metrics suite for object-oriented systems is that proposed by Chidamber and Kemerer (C&K), and variants derived from this, covering coupling, cohesion and inheritance relationships (Chidamber and Kemerer, 1994). Chidamber and Kemerer used viewpoints collected from practitioners to drive the development of their metrics. Concerns about the theoretical validity of some of the cohesion and coupling aspects of the original C&K metrics (Hitz and Montazeri, 1996; Kitchenham, 2010; Mayer and Hall, 1999b) have led to refinements such as LCOM4 (lack of cohesion) (Hitz and Montazeri, 1995) and Low level design Similarity-based Class Cohesion metric (LSCC) (Al Dallal and Briand, 2010).

Object-oriented design guidelines and their associated metrics have been subject to a range of empirical evaluation. Most of the investigations explore potential relationships between internal factors, such as coupling and cohesion, and external factors such as fault- or change-proneness, development effort, ripple effect, and expert opinion (Briand et al., 2002). A systematic literature review of 99 studies by Jabangwe et al. explored the potential link between object-oriented measures and external quality attributes (e.g.

reliability and maintainability). In relation to maintainability, the review found that "*Overall, coupling, complexity and size measures seem to have a better relationship with maintainability than inheritance measures*" (Jabangwe et al., 2014). Al Dallal found that most of the cohesion, size, and coupling metrics could predict the more maintainable classes in three open-source Java systems (Al Dallal, 2013). Ferreira et al. identified threshold values for metrics which could help distinguish well-designed classes from less well-designed classes (Ferreira et al., 2012). Dallal and Morasca suggest that coupling and size have a positive impact on reuse-proneness (Al Dallal and Morasca, 2014). Bajeh et al. found a correlation between coupling and cohesion metrics and testability (Palomba et al., 2016). Metrics have also been widely used for fault prediction (Radjenović et al., 2013) and identifying bug-prone code (Palomba et al., 2016).

Kitchenham performed a preliminary mapping survey of empirical research on software metrics (Kitchenham, 2010). The key findings were that there is a large body of research related to metrics but that this is dogged by significance issues: invalid empirical validations (especially related to theoretical limitations of the C&K LCOM metrics and the treatment of code size), contradictory results and too many comparisons in single studies (e.g. 66 different metrics). Other work has raised concerns about the value of such metrics in predicting design quality. Riaz et al.'s systematic review of maintenance prediction and metrics (Riaz et al., 2009) found weaknesses in terms of comparison with expert opinion and external validity, as well as differences in the definition of 'maintenance' used in the studies. Sjøberg et al. found that a range of size, complexity, coupling, cohesion and inheritance metrics were not mutually consistent and that only (large) size and low cohesion were strongly associated with increased maintenance effort (Sjøberg et al., 2012). Cinnéide et al. studied the use of five cohesion metrics to guide refactoring and concluded that the metrics often captured differing notions of cohesion (Ó Cinnéide et al., 2016). Veerappa and Harrison repeated Cinnéide et al.'s study focussing on coupling metrics (Veerappa and Harrison, 2013) finding that coupling metrics appeared less conflicting than cohesion metrics and that improving coupling did not necessarily improve cohesion.

While potentially useful, the literature indicates that a variety of metrics might suffice for tracking maintainability. However, the interaction between these metrics, their *normal ranges*, and overall validity are still poorly understood. It is also notable that while metrics are often inspired by guidance, they appear to capture only narrow aspects of the guidance

- by trying to capture guidance in code, the semantics are lost much like for any other domain.

Mayer and Hall make a case that the original C&K metrics do not accurately reflect the concerns expressed by the viewpoints in their original paper – noting that the viewpoints express design concerns using terms from structured programming. They contend that "*OOP has a number of design fundamentals, or mechanisms not present in structured programming that renders these attributes redundant.*" (Mayer and Hall, 1999a). They have also noted flaws in other OO metrics suites such as the MOOD metrics (Mayer and Hall, 1999b).

Page-Jones proposed that the concepts of coupling and cohesion from structured design should be considered as the same phenomenon in OO design – *connascence* – "*two software elements A and B to be connascent if there is at least one change that could be made to A that would necessitate a change to B in order to preserve overall correctness*" (Page-Jones, 1992).

Sources of information must be presented to the practitioner when they are relevant, thus metrics are often integrated into development environments or build systems to provide timely warnings.

2.3.1 Inheritance Metrics

The C&K (Chidamber and Kemerer, 1994) metrics which relate to inheritance are:

- Class Complexity
  - Weighted Methods per Class (WMC) = Sum of methods in a class, modified by some weighting per method, the default value for each method is one.
- Scope of class properties:
  - Depth of Inheritance Tree (DIT) = Depth of a class in the inheritance tree.
  - Number of Children (NOC) = Number of immediate descendants of a class.

In addition, Response for Class (RFC) and Lack of Cohesion in Methods (LCOM) may be indirectly affected by the structure of inheritance hierarchies.

For example, WMC counts methods defined in a class, and does not account for inherited methods. This appears to only partially account for one of the *viewpoints* which inspired the metric – "*The larger the number of methods in a class the greater the potential* *impact on children, since children will inherit all the methods defined in the class*" (Chidamber and Kemerer, 1994). There is no indication if overridden or inherited methods are weighted differently by the practitioner – even though these affect the final method count of hierarchy leaves in different ways.

Kitchenham notes that it can be difficult to separate code and design faults in the literature, as these are treated similarly, and often measured indirectly using the same metrics (Kitchenham, 2010). For example – DIT is motivated by concerns about complexity (prediction of behaviour, design effort) which is generally avoided, but also reuse potential which is beneficial. Simply knowing the depth of a given hierarchy does not does not reveal whether this is a *good* use of inheritance. At best, this can be used to identify candidate outlier structures for assessment.

Gill and Sikka address the issue of comparing the relative *fitness* of inheritance hierarchies in terms of their core value – (current) reuse and (future) reusability (Gill, 2011). They note that assessing *negative* factors such as magnitude/size and complexity do not shed much light on *desirable* qualities, there remains the issue of defining good ranges for the metrics (Emam et al., 2000). The difference between avoiding risk and the desire for improvement is not as clear in metrics as it is in the v*iewpoints* or design guidance which inspire them. In practice, metrics are used to define bounds and warnings on extreme values.

For example, SonarQube, treats inheritance depth over 5 as a severe quality issue and calculates remediation time at "*4h +30min Number of parents above the defined threshold*". Justification for this is that - "*Most of the time a too deep inheritance tree is due to bad object oriented design*" (SonarQube, 2013). This caveat illustrates that deep hierarchies are, at worst, a *code smell* rather than a fault.

In a retrospective article, an originator of the C&K metrics reflects on the effectiveness of their metrics suite in the field (Darcy and Kemerer, 2005). They note that in industrial practice the C&K metrics are used to detect areas of interest relating to "*managerial-performance variables including effort, reusability, defects and faults, maintainability, and cost savings*". Again, there is an acknowledgement that "*…local model calibration would be important*" between languages and domains (Darcy and Kemerer, 2005).

If local organisation and practice can overwhelm or confound metrics, this may make constructing an *entity population model* (see 2.5 Entity Population Model) very difficult for software in general. Even if it is accepted that metrics address the issues raised by their proposers – these are *predictions about risk factors, applied to code that has already been written*. This offers little in terms of design advice during development.

Bajeh found that poor *average understandability* (many ancestors) and poor *average modifiability* (many descendants) in inheritance hierarchies are good proxies for complexity and correlate well with difficulty in making changes related to closing issues in two projects (junit, elasticsearch) (Bajeh et al., 2014). While this indicates that larger hierarchies may cause more maintenance issues than smaller hierarchies – it is not clear if size is the only factor in play.

Niculescu investigates the "*ratio of the number of inherited methods among all the methods executed in an instance of a class when running some scenario of business value*". (Niculescu et al., 2015) This is based on the notion that *good inheritance* usage results in a balance between new and inherited behaviour. This work represents a rare dynamic metric for inheritance hierarchies which may yield insights into the complexity of the execution path inside hierarchies. This might have some bearing on the complexity of a hierarchy. Complex execution paths may also indicate that a hierarchy exhibits the yoyo problem (Taenze et al., 1989).

In summary, metrics measure aspects of existing code – these are interpreted or combined to relate to aspects of design – but it is not clear how well or completely the code metrics capture these more abstract and subjective concepts. Metrics which *are* defined to be more practitioner facing often include assumptions or simplifications to facilitate interpretation.

Metrics are applied after design decisions have been made – once code has been written. Meaning that they cannot easily inform *what code to write*. Code metrics may be useful for quality assurance activities such as auditing or targeting code review, there is no indication that metrics are used to inform day-to-day design choices. While there appear to be high-level similarities between systems, models of quality do not travel well, and *normal* ranges for metrics have little or no theoretical basis.

2.3.2 Interface Metrics

Interfaces can be difficult to measure using conventional measurements as they contain no executable code, thus they are often excluded from measurement-based studies.

Additionally, many of the design needs met by interfaces satisfy architectural or modelling needs, which may be difficult to detect in source code alone.

Service Interface Usage Cohesion (SIUC) (Perepletchikov et al., 2007) encourages interfaces to be specific to each client – "*an interface has a strong cohesion if each of its client classes actually uses all the methods declared in the interface*". This metric has been used as a measure of interface cohesion in recent work (Abdeen et al., 2013a; Romano and Pinzger, 2011), and is based on the interface segregation principle (ISP).

This deviates subtly, yet significantly, from the original intent of the ISP – which advises to "…*categorise clients* [of an interface] *by their type, and interfaces for each category of client should be created.*"(Martin, 2000). Furthermore, in ISP the advice specifically allows for duplicating methods among different services (interfaces) to permit segregation of coincidentally similar clients.

Paradoxically, the original literature for ISP describes a scenario which would score a perfect rating of 1 using SIUC: "*The ISP does not recommend that every class that uses a service have its own special interface class that the service must inherit from. If that were the case, the service would depend upon each and every client in a bizarre and unhealthy way*" (Martin, 2000). This highlights a risk when associating a specific metric with a guideline in name, without considering the original aim of the advice.

The original formulation of SIUC by Perepletchikov (devised for service oriented architectures) was accompanied by three other metrics for other aspects of cohesion, including *conceptual* cohesion, which covers functional/model cohesion from the OO (object oriented) paradigm (Perepletchikov et al., 2010).

The finding by Shata et al. that "…*interfaces have a strong tendency to be designed according to the PTIP and ISP properties, but also with neglecting the cohesion property.*"(Abdeen et al., 2013a), indicates that decoupling is of higher priority than eliminating 'dead code' represented by uncalled interface methods. This indicates some measure of tolerance among practitioners for the uncalled methods that SIUC is designed to detect.

Loose Program to an Interface (LPTI) measures how many references to interface methods are via an interface vs via a concrete implementation of that interface (Abdeen et al., 2013a). This gives an estimate of how close interface use is to *optimal* (defined as when all methods which can be accessed via an interface, are always accessed via an interface). This metric makes no claim about the number, location, or fitness of interfaces in a system. Since object creation requires access to concrete types in Java, this metric cannot reach a score of 100%. There are differing opinions on how object creation can (or should) be managed among practitioners – so this may factor into the interpretation of this metric.

Interface Implementations Model (IIM) constructs a model by combining all the implementations of an interface. This allows the logical cohesion of an interface to be investigated based on logical cohesion of methods in each implementation. However, as implementation details are irrelevant during interface design, it is not clear if the designer of an interface would necessarily care if the implementations of that interface are cohesive. It could also be argued that the presence of an interface itself lends cohesion to the implemented methods from that interface.

Assessment of interfaces using metrics focuses on the more tangible aspects of interfaces such as cohesion, number of methods and constants, naming, stability, number of implementers, number of clients. In addition to simple counting metrics, some metrics attempt to apply traditional quality aspects such as cohesion to interfaces, which may not be appropriate.

The characteristics of interfaces are in some ways more difficult to interpret than metrics for concrete classes. Concrete classes have a 'justification' in their implementation – implementation can be divided up between different classes, that functionality must exist somewhere in the system.

Unlike concrete classes, interfaces carry little information to justify their existence, and may be added to a design to address concerns above the implementation level. This presents an issue when discussing *good* value ranges for interfaces – it might be argued that an interface can only be assessed in terms of meeting its design goal, regardless of its implementation level properties.

These metrics illustrate that it can be difficult to capture simple high-level rules in a single value. It may be more useful to describe how much, or what aspects of a guideline are being captured by a metric to aid in interpretation.

2.4 Modelling

In this section, modelling is discussed in relation to the use of interfaces and inheritance. While there is little theoretical basis for 'good' metric score ranges, some attempts have been made to provide complete sets of use cases for specific structures. These models describe different ways in which structures *ought* to be used, blending existing guidance and experience to provide comprehensive advice.

2.4.1 Modelling Interface Use

Steimann and Mayer attempt to manually categorise different uses of interfaces based on arguments from design (Steimann and Mayer, 2005), and consideration of whether the caller or called benefit. These are further grouped by categories – general (G), offering (F), context-specific (C), enabling (E).

> Idiosyncratic (F, G) – an interface has a single implementation and the public interface of the implementing class matches the interface. No specific client.
>
> Family (F, G) – heads multiple implementations of a family of classes, often with alternative implementations or differing run-time behaviour, such as data structures. No specific client.
>
> Client/Server (F, C) – the implementer is the server, which offers the services needed by different clients. Different interfaces may be offered to different clients.
>
> Server/Client (C, E) – implementing the interface allows access to services offered by some server(s), generally for the benefit of the client.
>
> Server/Item (C, E) – enables processing of item interface objects for the benefit of some third party e.g. Comparable, Printable. These are typically the '-able' or '-ible' classes in Java.

Even though Steimann et al. were unable to identify "*robust decision rules*" to distinguish between these categories, manual classification was possible - indicating that subjective interpretation was required. Additionally, it is notable that there is no mention of metrics until after classification. This may indicate that the notion of *benefit* from interface mediated interaction, while obvious during practitioner assessment, may not be easily automatable.

Other work by Steimann discusses the difference between *role* and *natural types*. Where roles are added and removed without affecting an object's identity, and may be related to some specific interaction. Natural types on the other hand are not specific to a particular interaction, but are essential to the identity of an object (Steimann et al., 2003). This mirrors some of the UML colour modelling archetypes proposed by Coad – namely Coad's "role" or "description" for Steimann's role, and Coad's "party, place, thing" for the natural type (Palmer and Coad, 2004). The distinct recognition of temporary *roles* is also a central principle in the Data-Context-Interaction method of OO modelling (Reenskaug and Coplien, 2009).

Models of interface use, based on design guidance can offer insight into the benefits of interface-mediated interactions. However, categorising these relationships may be highly subjective. Given a lack of some record of this categorisation in a design, and system evolution, there is no guarantee that the original intention will be retrievable, or useful when the next practitioner examines the design.

2.4.2 Modelling Inheritance

Inheritance is a powerful technique which can create complex structures. In addition to the stand-alone guidelines presented in Section 2.2, some commentators have gone a step further and provided a more comprehensive set of guidelines for all uses of inheritance, including identification of 'improper use'.

Before discussing the LSP in detail, Liskov discounts so called *implementation hierarchies* from further consideration. This use of inheritance exists solely to re-use some functionality and is not intended to make use of polymorphism. While Liskov's work is not presented as a taxonomy – it does address inheritance under several categories (Liskov, 1988). Some of these definitions show the age of the work – Grouping super-types might be better modelled as interfaces in Java, while implementation hierarchy appears to be the situation the Gang of Four (Gamma et al., 1994) had in mind when they suggested that composition is preferred over class-inheritance.

Liskov identifies inheritance related concepts:

Implementation hierarchy – extension solely to reuse code with no expectation of polymorphic use. Liskov notes that this level of reuse can already be achieved by other means (composition) and that the only contribution of this approach is to damage encapsulation. This is essentially the same as sub-classing.

Hierarchy Polymorphism – where a common operation or operations e.g. *sortable* are used to define a super-type. Super-types are not meaningfully related to their extenders as these are generic sets of behaviour.

Multiple implementations – where a common super-type enables one implementation to be replaced by another. Requires only interface compliance with a common super-type. Although with more implementations, 'discovered commonalities' between leaf types – as recommended by Johnson and Foote - "*…it is often possible to break a method into pieces and place some of the pieces in the superclass and some in the subclasses*" (Johnson and Foote, 1988). These layers, are driven by implementation reuse, rather than modelling – Liskov recommends separating sub-typing and implementation reuse. In this case, reuse of common code drives increasing hierarchy depth because code can only be shared if it is moved 'up' in a hierarchy (towards the root), while alternative implementations (which must be in separate classes) drive increasing hierarchy-width.

Incremental Design – a subtype may be an *alternate view* of its super-type, which allows the subtype to provide for the needs of different client objects than its parent type, while reusing functionality. This may mean adding multiple layers of extension to a single type. Intermediate types in the hierarchy may thus represent intermediate *evolutionary steps* between the root of the hierarchy and eventual leaf-types. This allows system extension without changing existing functionality. This may mainly be relevant during the maintenance or evolution stage of a system – where there is no design rationale for, or too much risk in consolidating older parts of a system.

Meyer proposed a taxonomy of *the many faces of inheritance* (Table 4) – while also arguing that a single mechanism (of inheritance) was practical on the basis that "*If we accept classes as both modules and types, then we should accept inheritance as both module accumulation and subtyping*". This contradicts Liskov who preferred that "*These uses should be kept separate*" (Liskov, 1988).

However, it is in line with Meyer's very pragmatic approach - the key points are that "*…inheritance is applicable only if you can seriously argue for the presence of an "is" relation between the instances of the heir and parent…*", and "*…the only goals that count*

- *quality software and fast delivery…*" (Meyer, 1996). In terms of 'good' design, this highlights that what *ought* to be done is a subset of what *can* be done, but that different guidance does not always agree where the boundaries of this subset lie.

Meyer's approach is that anything that is not misleading is potentially useful. This is much more permissive than other guidance, Meyer asserts that advice against implementation inheritance "*lacks the strong theory that should support any such indictment*"[2]. Meyer *does* note that there are categories of "*improper use*" – notably:

- *'Has' relation with no 'is' relation* - reuse without type relationships
- *Taxomania* - Creation of types with no new features, which could be distinguished by flags e.g. TrafficLight having subtypes for RedTrafficLight, GreenTrafficLight, etc.
- *Convenience Inheritance* - Inheriting from a type that is conceptually related, but is not a parent e.g. Apple is not a super-type of ApplePie

Table 4 shows a summary of Meyer's taxonomy – with separate high-level motivations for inheritance – these distinguish between *model* (or problem domain), *software* (implementation details), and *variation* inheritance. The motivations suggest that there should be a clear and singular purpose behind each use of inheritance.

Meyer's taxonomy advocates use of many forms of inheritance that are considered harmful in other areas such as *implementation inheritance*, discouraged by Liskov and the design pattern literature. Some modes of inheritance use have been superseded in modern languages e.g. View inheritance could be achieved even with Java's single inheritance.

Some of these *improper uses* map well onto categories identified by Taivalsaari i.e. flags might be better handled as a class property (classification), conceptual relationships may be better modelled by aggregation (composition) (Taivalsaari, 1996). Considering the *kind* of abstract relationship can highlight how well this relationship maps to the properties of the chosen abstraction mechanism.

When Mayer's taxonomy was applied to a corpus of object oriented programs (English et al., 2005), it was found that the semantic aspects of the classification were not suitable for static analysis, consequently the analysis had to include opinion data from the practitioners and original designers.

[2] Referring to Dijkstra's letter – "Go To statement considered harmful" (Dijkstra, 1968)

External Model	Inheritance Classification	Description	Example	Parent (A)	Child (B)
Model (Domain)	Subtype	A and B represent A' and B' which are external objects, B' is a subset of A', any other subtype heir of A is disjoint from B'	Subtype inheritance is closest to the hierarchical taxonomies of the natural sciences	abstract	abstract or concrete
	View	B describes the same abstraction as A, but viewed from a different angle, recombined via multiple inheritance, *advanced technique*	Access pattern, representation, and traversal aspects of a data structure are selected and combined	-	-
	Restriction	Instances of B are instances of A that satisfy a constraint, any feature added by B should be a result of the new constraint	ELLIPSE <- CIRCLE, where the extra constraint is that the two focuses of an ellipse are merged	Both abstract or both concrete	
	Extension[3]	B introduces features not present in A and not applicable to direct instances of A	Specialisation with the addition of new attributes. MOVING-POINT <- POINT and add speed, for magnitude and direction	concrete	-
Variation	Functional Variation	B redefines some features of A, and some of the redefinitions affect feature bodies, not just signatures, B must not introduce any new features except for the direct needs of the redefined features	Adapt an existing class without affecting the original class and its clients, avoids directly modifying existing software [4]	Both abstract or both concrete	
	Type Variation	B redefines some features of A, and the redefinitions affect only signatures, B must not introduce any new features except for the direct needs of the redefined features	Adapt a class to return a type which is a sub-class of the original return type	Both abstract or both concrete	
	Uneffecting	B redefines some of the concrete features of A into abstract features, *this should not be common*, goes against the normal direction of inheritance	Commonly: Select functionality in multiple inheritance, or reuse a useful class that is too concrete	-	-
Software (Implementation)	Reification	A represents a general data structure and B represents a partial or complete implementation choice for that data structure.	TABLE describes tables of a general nature, reification leads to SEQUENTIAL-TABLE, still abstract, Final reification leads to concrete classes ARRAYED-TABLE, LINKED-TABLE	abstract	abstract or concrete
	Structure	A represents a general structural property, and B represents a certain type of object possessing that property.	COMPARABLE super type	abstract	abstract or concrete
	Implementation	B obtains a set of features from A (other than constant attributes and once functions) necessary to implement the abstraction associated with B	ARRAYED-STACK inherits its specification from STACK and its implementation from ARRAY	concrete	concrete
	Facility	A exists solely to provide a set of logically related features for the benefit of heirs such as B.	Utility class providing facilities for detailed access to some mechanism, is inherited by classes that need those facilities	-	-
	Constant	Variation of Facility	Features of A are all constants or once functions describing shared objects	concrete	concrete
	Machine	Variation of Facility	Features of A are routines, which may be viewed as operations on an abstract machine.	B is at least as concrete as A	

Table 4 : Meyers Taxonomy – abridged form of an inheritance taxonomy (Meyer, 1996)

[3] The presence of both the restriction and extension variants is one of the paradoxes of inheritance. Extension applies to features, and restriction (and more generally specialization) applies to instances, but this does not eliminate the paradox. The problem is that the added features usually include attributes. So if we take the naive interpretation of a type (as given by a class) as the set of its instances, then it seems the subset relation is the wrong way around! (Meyer, 1996)

[4] if you do have access to the source code of the original class, you should examine whether it is not preferable to reorganize the inheritance hierarchy by introducing a more abstract class of which both A (the existing variant) and B (the new one) will both be proper descendants with peer status (Meyer, 1996).

The requirement for semantic comparison makes it difficult to apply the taxonomy *after the fact* as the intent of a developer is often not recorded at implementation time. Furthermore, since designs evolve, the classification of a particular use of inheritance may change over time, which may make the semantics less clear. Finally, English notes the uneven distribution of the various archetypes in the analysis – this suggests that creating a single entity population model for all hierarchies would have to account for the various *norms* within the population, rather than defining a single *usual value*.

Dvorak proposed object-oriented conceptual modelling (OOCM) to make explicitly the hierarchy's conceptual architecture and proposes quantitative sub-classing criteria. This model classifies hierarchy features as *"essential, supportive, and incidental"* to the hierarchy (Dvorak, 1994). Where essential properties "*…capture the essence of a class concept, whereas supportive properties capture important but not crucial elements of the class. Incidental properties capture elements of a class concept that are implementation-oriented.*" These properties are then used to calculate conceptual specificity, consistency, and distance in inheritance hierarchy design.

For sub-classing, Dvorak assigns values for generality from the problem domain to ensure that "*…classes of greater generality are placed higher in the class hierarchy.*" Conceptual consistency is ensured by only allowing sub-classes which:

- Preserve the essential properties of the super-type
- Do not negate any essential properties of the super-type (in Java, this would be refused bequest for an essential property)
- Do not demote an essential property of a super-class to an incidental property

A sub-classing algorithm is proposed to create one or more hierarchies from a pool of hierarchy members based on conceptual consistency values and specificity. While this can be done automatically, it requires a great deal of manual documentation and that the practitioner requires sufficient (semantic) information to properly categorise the hierarchy features. These requirements indicate that this model may be useful at the initial design or review stages where this detailed analysis is more likely to be carried out.

These taxonomies of inheritance illustrate that design demands and opinions on acceptable practice are diverse, while programming languages offer only a few structures to capture these differences. These taxonomies focus on the motivations for use of inheritance, but do not extrapolate what inheritance use would look like in practice at the system level if these models were followed. To the practitioner, the existence of a well-defined and meaningful *is-a* relationship is a valuable insight into a design. If inheritance is used as a *mere* implementation detail, then the information encoded in these structures may offer less insight. Additionally, inheritance structures may be more expensive to assess, refactor, and maintain as they become more complex – which may offset perceived benefits. The taxonomies presented above do not directly address these trade-offs. Notably, the models all deal with individual steps in a hierarchy – indicating that different extensions within a hierarchy may meet different purposes. While not explicit, much of the guidance and teaching material imply that hierarchy growth means *more of the same*.

Finally, the original design motivation for a specific structure may be absent from a design, or out-of-date after some system evolution. The same structure (contrary to some guidance) may satisfy multiple design motivations. Taking these issues into account, it may be more realistic to assess inheritance structures 'as they are found'. So, then is there any useful assessment that *can* be done, without recourse to manual (subjective) classification?

2.5 Entity Population Model

An entity population model, as defined by Kitchenham, captures the normal values for a measurement. "*When we measure an attribute on a particular entity, we assume our measurement can be interpreted by reference to the 'normal values' of specific attributes for specific entity classes under specific conditions. An entity population model allows us to define the normal values for an attribute*." (Kitchenham et al., 1995) There has been some effort to capture the current state of practice by examining large corpora of software in the wild.

The creation of an entity population model relies on the assumption that measurements fall into a normal distribution across systems, and that the desirable part of the distribution can be identified.

2.5.1 Inheritance in Practice

Tempero et al. investigated the use of inheritance in 93 applications within the Qualitas Corpus - the focus of their work was inheritance in Java *as it is used in practice* – acknowledging that there may be a discontinuity between how inheritance is used in practice from what would be expected from following the guidance. One objective of the study was to enable discussion of "*typical or extreme"* systems with respect to the rest of the corpus. The study found that "*around three-quarters of user-defined classes use some form of inheritance in at least half the applications in our corpus*" – with most classes extending other user-defined classes. They also found that *"… applications generally have a lower proportion of classes implementing interfaces than classes extending classes*" (Tempero et al., 2008).

Tempero also notes that most types appear in the shallow part of hierarchies. Emphasising that the population is skewed – so a comparison with a population norm may serve to highlight outliers, but not explain or advise on use of these less common structures. It is also noted that a few classes have many children and that these classes tend to scale up with program size, all other types have few children. Again, if advice is to be given or assessment made of these structures, it seems necessary first to identify the kind of structure that is being observed, then assess it *on its own terms*.

Tempero et al. carried out a further study investigating what programmers do with inheritance in Java systems (Tempero et al., 2013). This study is distinct from the previous study in that it was more focused on practitioner choice, than simply characterising a corpus. The study was an assessment of byte-code and excluded references to Java library and third-party types.

The study found that two-thirds of inheritance is used for sub-typing (substitutability), specifically where a subtype is used in place of a super type e.g. variable assignment or as a parameter. They also identify that approximately a fifth of inheritance might be refactored to composition – though the appropriate criteria to make this decision or the design impact of this are unknown. Overall, they conclude that "*…there was a considerable amount of use of inheritance … most of that use is justified…*". However, this is based exclusively on an *inheritance required to compile* criteria - a third of subclasses rely on down-calls to customise superclass behaviour, and two thirds of inheritance is used for sub-typing. The authors do not infer anything about quality from this analysis. The study also raises the possibility of different *kinds of hierarchies* are hinted at when discussing the effect of inheritance hierarchy depth on maintainability may be more related to the *kind* of inheritance – "*different uses of overriding could explain the variation*" (Tempero et al., 2013).

Collberg carried out a census-style style survey of 1132 Java-jar files sourced from internet software repositories, in this case, the analysis targeted byte code. The authors note that "*most of our data have sharp 'spikes' and long 'tails'. That is, one or a few (typically small) values are very common, but there are a small number of large outliers which by themselves are also interesting*" (Collberg et al., 2007). This illustrates that outliers are common in many of the measurements.

Again, it was found that there are many shallow hierarchies with a few deep outliers. It is interesting to note that Collberg also reported that 89% of classes have no subtypes – this is a view of the entire corpus. It is not clear how many of these non-extended types are leaves in inheritance trees or simply classes that are not in hierarchies (Collberg et al., 2007).

Overall the *census* approach relies on the notion that some useful comparison might be drawn between aspects of an individual system and the same aspect of the systems over a large *representative* corpus. Attempts to build reusable models between systems have not been successful, other than between very similar systems, and after controlling for differences in process and data collection (Murphy, 2011).

Once inheritance is present in a system, there is an implicit cost to removing it - refactoring out an inheritance hierarchy may be difficult and resource intensive (redesign/testing). Kegel and Steimann note that although they found that 63% of inheritance structures could be refactored to composition, *"…no insights can be derived from them as to whether or when applying the refactoring leads to better design."* (Kegel and Steimann, 2008).

Inheritance is in common use and appears to be integral to the designs where it is present. Variations in expertise may account for different modes of use – however is it clear that inheritance is not 'factored out' of designs even where the change is feasible. It is not clear if this staying power is due simply to the cost of removal, or if it is not clear *when* inheritance should be removed.

Many concerns about inheritance are about system evolution. Though the relationship between inheritance and evolution problems, particularly maintenance effort, is inconsistent - suggesting that other factors than those measured are in play.

2.5.2 Interfaces in Practice

There have been a few studies that explore interface use in practice.

Gößner et al. focussed on the use of interfaces in JDK (J2SE 1.4.1_02) (Gößner et al., 2004). They found that the interfaces most frequently implemented were special purpose 'marker' interfaces (with no abstract methods) such as Serializable and Cloneable. The next most popular interfaces were a range of 'listener' interfaces, again fulfilling quite a specialised role but clearly supporting programming to an interface through the Observer interface - 50% of all interface implementations examined were of these 'enabling' interface types. In terms of use of interfaces, they found that the head of families of objects, iterators and collection types to be the most widely used, and that most classes implement either 0 or 1 interface only, only a small number implementing more than 8 interfaces and these being "*rather exotic classes*". They found that although only 20% of the total types available in the JDK are interfaces, 31% of classes are accessed exclusively through interfaces, which they saw as a positive use of program to an interface. Finally, they showed that the class Vector is only accessed via its interfaces (e.g. List) 46% of the time when it could have been used via interfaces 85% of the time.

While examining interface use in the JDK, Steimann et al. manually classified a sample of interfaces based on the 100 most implemented and 100 most referenced (by number of variables). These were distinguished by comparing variables of interface type and interface implementations. Categories of interface were identified – *enabling* interfaces are implemented by classes that are interchangeable 'service providers' who take part in the activities of one or few callers, while *context-specific* interfaces are often partial interfaces and are usually specific to a narrow interaction with many callers. It was found that *enabling* (a sub-group of context-specific) interfaces are implemented more often than they are referenced – the opposite being true for *offering*. Similarly *offering* interfaces tend to have a high number of methods (which the authors call a *weak contract*), with a mean of 19 while *enabling* interfaces have fewer methods, with a mean of 5 (Steimann and Mayer, 2005).

These findings indicate that small interfaces are implemented many times because they capture roles in popular interactions, while larger interfaces capture general services interfaces with fewer implementations. Steimann and Mayer also found a consistent ratio of 5.5:1 of classes:interfaces across 5 releases of the JDK from 1.0.2 to 1.4.1. They also found that the ratio of interface-typed:class-typed variables nearly doubled with the change to Java 2 (JDK 1.2) from 1:9 to 1:5. At the same time, the average number of interfaces implemented per class jumped from 0.8 to 1.8.

This finding agrees with Tempero's comment that "*we can expect that as programs get larger, the numbers of implementations of popular interfaces and the number of descendants of popular classes will grow without limit.*" (Tempero et al., 2008). However, this does not resolve the issue of assessing interfaces as these findings are the result of corpus studies, where the *popular* interfaces are only detectable because they are popular – given the presence of many non-popular interfaces, popularity cannot be the sole measure of quality, if at all (Stevenson, 2014).

In a further study Steimann et al. investigated the use of interfaces in JDK 1.4, jboss and eclipse (Steimann et al., 2003). They found a ratio of class:interface definition ranging from 3.7:1 to 6.6:1, ratio of implementations:classes ranging from 0.5:1 to 0.8:1, and the ratio of types defined by classes:defined by interface ranging from 4.5:1 to 2:1. In 2004, they found that the "*use of classes in variable definitions still clearly dominates the use of interfaces*" and speculated that this was partly due to the effort involved in introducing and maintaining interfaces and that there was a lack of "*intuitive conceptualisation*" of interfaces compared to classes.

Abdeen and Shata, and their co-workers, have explored a wide range of interface usage in multiple studies using between three and twelve open source systems (Abdeen et al., 2013a; Abdeen and Shata, 2014, 2012). Some of their key findings that are related to this study include:

- Many interfaces were implemented by only one class
- Many interfaces were identical to the public interfaces of their implementing classes. Only about 25% of interfaces were really "partial interfaces" for their implementing types
- The ratio of classes to interfaces ranged from 3.3:1 to 27:1
- The percentage of classes that "implement directly" interfaces range from 15% to 40%
- Evidence of overlapping between interfaces (8% to 44% of redundant method declarations) and even interface clones
- 7% to 35% of interface methods unused in their applications

- Significant amounts of redundancy in interface hierarchies

Attempts to model interfaces and inheritance-focused studies often ignore marker and constant interfaces (Abdeen et al., 2013a, 2013b; Amálio and Glodt, 2014; Tempero et al., 2008). This is despite the fact that super-types that contain only constants are found to be "*fairly common*" in a different study by Tempero – with interfaces being preferred over classes for constant-carrying super types (Tempero et al., 2013). Tempero also found that marker interfaces (with no method definitions) are common in larger systems (Tempero et al., 2013). There is also evidence of cases where interfaces are a *standard* practice, for example for accessing an API, or interacting with Java serialisation. In these cases, the practitioner has no decision to make, but it may be difficult to distinguish these cases with automated tools.

2.6 Survey Research and Practitioner Attitude

This section provides a background for survey research, covering both survey methodologies and insights from surveys performed relating to software design practices.

Surveys have been used to discover the state of industrial software engineering practices for more than three decades (Beck and Perkins, 1983). In recent years they have become particularly popular, taking advantage of online distribution of questionnaires, email and social media for contacting potential participants, and using highly-regarded survey guidance (Pfleeger and Kitchenham, 2001).

One of the most similar design-related surveys to this work was on design patterns (Zhang and Budgen, 2013). The goal was to discover which of the Gang of Four patterns were considered useful by experienced users. They approached all authors of papers relating to design patterns in the mainstream software engineering literature. They received 206 usable responses (response rate of 19%). More than half the respondents had over ten years of experience in object-oriented development. The survey was administered using the SurveyMonkey commercial site. It used a combination of Likert-item questions and open-ended questions but received "*relatively few comments*" – 338 in total. Their main finding was that only three design patterns – Observer, Composite, and Abstract Factory – were highly regarded.

Simons et al. surveyed 50 professional developers comparing their opinion of design quality to that of metrics based on design size, coupling, and inheritance (Simons et al., 2015). They found no correlation between the metrics used and professional opinion when assessing UML class diagrams. One of their main findings was the need to involve humans within the software design quality assessment process; metrics alone were insufficient. Yamashita and Moonen surveyed 85 professional developers exploring their knowledge and use of code smells (Yamashita and Moonen, 2013). 32% of respondents did not know of code smells. Only 26 were concerned about code smells and associated them with product evolvability. Respondents highlighted that they often had to make trade-offs between code quality and delivering a product on time. At Microsoft, Devanbu et al. found that practitioners have strong beliefs largely based on personal experience rather than empirical evidence (Devanbu et al., 2016). They call for empirical research to take practitioner beliefs into account when designing new empirical studies.

Other recent, but less closely related, software engineering surveys include: the practices used in the Turkish software industry (Garousi et al., 2015), the use of modelling in the Italian software industry (Torchiano et al., 2013), the use of UML modelling for the design of embedded software in Brazil (Agner et al., 2013), testing practices used in the Canadian software industry (Garousi and Zhi, 2013), the data and analysis needs of developers and managers at Microsoft (Buse and Zimmermann, 2012), and the use of software design models in practice, in particular UML (Gorschek et al., 2010).

A number of common themes emerge from these surveys. All surveys tend to use an online, web-based delivery mechanism such as SurveyMonkey. Many closely follow the guidance of Pfleeger and Kitchenham on survey methodology (Pfleeger and Kitchenham, 2001). Most focus on the use of Likert-item or fixed-response questions with associated (quite straightforward) quantitative analysis. There is some use of free-text questioning and routine qualitative analysis. Sampling is challenging, with a tendency to use personal contacts, public records of relevant organisations, mailing lists, and social media. The number of respondents typically ranges from about 50 to 250 (excepting one very large outlier (Gorschek et al., 2010)), in most of the studies respondents typically had 5-10 years of experience. Most of these survey papers tend to identify high level findings or guidelines, focusing on raising issues to be investigated further in more rigorous studies.

In terms of practitioner attitude to inheritance, there have also been a few relevant surveys. In a wide-ranging survey (3785 responses) Gorshek et al. found that there is a distinction between awareness of guidelines on matters such as class size/inheritance depth (which is widespread), and conformance to these guidelines (which varies) (Gorschek et al., 2010). This may indicate that compliance is being achieved in some other way, or at least to the satisfaction of the practitioner.

In a survey of OO practitioners (275 valid responses), it was found that around half of respondents think inheritance depth is a problem. Though more experienced programmers may have less of a problem with it. Inheritance also occasionally caused difficulty of understanding for many (80%) of respondents (Daly et al., 1995).

Dvorak found that deciding on the order and relationships within a hierarchy creates disagreement among practitioners, and even shallow hierarchies can make it difficult to place new members (Dvorak, 1994). A related concern is that complex hierarchies (using size as a proxy for complexity) have been shown to require stronger cognitive model-building or de-centralised task handling skills to understand, and may require more effort to modify, especially for less experienced developers (Kelly and Buckley, 2009).

2.7 Conclusions

Inheritance has the potential to reduce system implementation time, eliminate duplicate code, and encourage reuse. It can also be used to capture important hierarchical relationships within a problem or solution domain. Like interface implementations, subtypes can be freely substituted at runtime, creating flexibility. The extra facility of code reuse introduces potential complications in design – redundancy in hierarchies, complex method execution paths, fragile base class – which make the design difficult to understand and thus prone to errors during modification.

In contrast with inheritance are interfaces, which are often used to model roles – which may be transitory or incidental to the implementing type. Advice promotes that inheritance relationships are true *is-a* relationships, which places a greater burden of choice on the implementer. This may be made more difficult by Java's single inheritance constraint – this only allows for a single, most important *is-a* relationship for each object. Studies of the inheritance population indicate that there are no clear norms – though most inheritance is shallow. In addition, there are clear outliers in the population independent of what is measured – methods, hierarchy depth, width.

Guidance becomes more specific when applied to a specific structure, though this also highlights opposing points of view. A central conflict is between *optimal* and *practical*.

Optimal inheritance is as deep as possible to maximise code reuse, and has small intermediate steps to promote common behaviour up to the highest shared point requiring frequent refactoring and good understanding of the whole hierarchy. Practical inheritance tempers indirection, depth, and overriding with the need to read and maintain the code. Unfortunately, there is no consensus on what constitute reasonable practical limits.

Many of the guidelines address overlapping concerns, but often at different levels of abstraction, or in relation to different data structures. Some effort has been made to incorporate local context, or intuition into more 'locally aware' metrics. This, however, introduces the risk that these new metrics, which begin as *rules of thumb* much like other guidance, find their way into metrics and coding tools – what was rare or unusual becomes 'wrong' or 'to be avoided' – introducing unexplained and out of context limits on designs. Much like reusing software modules – the design assumptions used to construct tools or models may limit their portability unless these assumptions are well documented.

When corpora of object oriented programs are examined, inheritance is still an important design feature of many systems. There has been some investigation of the effect of the presence of inheritance on undesirable design properties, which overall, have proved inconclusive – there is no consistent relationship between inheritance markers such as hierarchy depth, and poor maintainability scores, or that 'known issues' such as the fragile base class problem is as problematic as the guidance suggests. It has been suggested there are perhaps other non-obvious properties such as hierarchy complexity which may need to be considered.

Models of inheritance attempt to distil the concerns and opportunities for inheritance use into all-encompassing schemes or taxonomies, however these are based on the proposer's interests, and make no claims about completeness or soundness. There is consensus that identifying an 'is a' relationship is key, but again there is difference of opinion on how this is recognised. So, even if compliance with one or more of the models could be demonstrated for a given design, it is not clear what assertions could then be made about that design.

A recurring theme in software design is *pragmatism*, these findings indicate that practitioners are conscious of guidance, but that *satisfactory* solutions may be achieved before full compliance with guidance. In addition, the definition of 'acceptable solution' appears to vary with experience and concerns of the moment. So, it is perhaps not the case that important and relevant aspects of software designs cannot be measured – but more accurate to say that design decisions can be difficult to reconstruct. Especially if the variable experience of the practitioner must be factored in to the definition of *acceptable* thresholds.

2.7.1 Research Questions

There is broad agreement that *bad design* is undesirable, evidenced by the volume and variety of design-guidance related material available to practitioners. However, there is also a noticeable gap between the popular guidance, which focuses mainly on low-level interactions between types and *ought*-arguments, and empirical measurement which focuses on prediction and population modelling.

<u>Thesis Question</u>: Is it possible to close the gap between object-oriented design guidance and design practice by surveying practitioners and studying how constructs such as interfaces and inheritance are used in open-source systems?

Consequently, it would be desirable to gather information from the environment where the guidance and metrics are used, under the constraints of time and production deadlines – from software practitioners. This may yield insight into what remains useful and important in a production environment.

<u>RQ1.1</u>: What does it mean for software to be of high quality, or well-designed? Can definition(s) of 'good design' be derived from the attitudes and beliefs of experienced practitioners, and the structure of mature software systems?

It has been several decades since the publication of the Design Patterns (Gamma et al., 1994) and the appearance of *Program to an Interface* – which is now a core principle in many object-oriented curricula, and is closely tied to more general programming approaches such as dependency inversion. Given this simple directive and the proposed benefits of programming with interfaces, it is surprising that recent code surveys have revealed that interface use remains relatively low. This raises the question of what good interface-based programming looks like at the system level.

<u>RQ1.2</u>: How are programming language constructs used in practice? Specifically, powerful, mature, and ubiquitous abstractions - such as interfaces and inheritance.

In contrast, despite warnings of detrimental effects, inheritance is still widely used – indicating that class inheritance is perceived as useful, or at the very least unavoidable by many practitioners.   Previous research in characterising inheritance has focused on either counting, or proposing principle- or experience-driven models. Despite extensive examination, there is no good model of how inheritance should be used – much of the design advice for inheritance use predates modern language features (and indeed some modern languages). It may be more useful to characterise inheritance use *as is* – to examine how use of inheritance reflects guidance and if any modes of use appear inconsistent with guidance.

RQ1.3: How does practice impact the structure of software? Specifically – is there any evidence within systems that guidance is being followed? Can (or should) systems be fully compliant with all guidance?

The following chapters to address these questions by investigating practitioner attitude and examining the source code of mature software projects.

3 Survey of Practitioner Attitude to Design Guidance

3.1 Introduction

The literature review highlighted that software quality is a broad area and that there are different approaches to defining and assessing quality. It also highlighted that early work on quality measurement (Chidamber and Kemerer, 1994) used practitioner viewpoints to determine what aspects of software were considered important. This chapter follows on from this work, investigating how practitioners view software quality, with a specific focus on the perceived value of object-oriented guidelines and the importance of software quality in day-to-day practice.

This survey aims to determine current attitudes to accepted object-oriented design guidance (as it is taught and promoted) among practitioners. There is much activity in the empirical software engineering field to validate the (supposed) popular guidance – underpinned by the assumption that this guidance is useful in practice. Similarly, the teaching of object-oriented best practice has become increasingly standardised in computer science courses, based on the teachings of popular textbooks and mantra-like principles. This initial study aims to determine what guidance is relevant to practitioners 'at the coalface', and how such guidance is applied.

3.1.1 Motivation

This research was motivated by the lack of previous work that explores the techniques used by software developers in their day-to-day practice and which of these techniques are perceived as most useful. It is important for the software engineering research community to know what practitioners are really doing in terms of design, what is useful and what is less so - to help guide the direction of future research. It is also of potential interest to developers working in industry to know the techniques and practices that their colleagues are using and find useful. It is important for educators to base their teaching on techniques and practices that are used in industry, and known to be useful in practice.

3.2 Study

3.2.1 Problem Statement

This research contributes to the problem identified by Baker et al. in their foreword to an IEEE Software special issue on Studying Professional Software Design (Baker et al. 2012): *"Although we have quite a few high-level design methodologies we don't have a* *sufficient understanding of what effective professional developers do when they design … To produce insights that are relevant to practice, researchers need to relate to practice and be informed by it".*

The research is further motivated by evidence-based software engineering (EBSE) (Dyba et al. 2005): "*A common goal for individual researchers and research groups to ensure that their research is directed to the requirements of industry and other stakeholder groups*". Finally, Carver et al. recently explored practitioners' views of empirical software engineering research (Carver et al. 2016) and one topic practitioners were keen to see more research on was "*Define good qualities of software and help developers to implement software with great quality*".

3.2.2 Research Objectives

The goal of the survey was to discover the extent to which practitioners concern themselves with software design quality and the approaches practitioners use when considering design quality in practice.

3.2.3 Context

The key practices and techniques used in the design of software are long established and widely known. Multi-edition software engineering textbooks (Sommerville 2010; Pressman 2014) have included chapters on software design for decades, there is a large collection of textbooks that specifically address software design (Coad and Yourdon 1991; Gamma et al. 1994; Riel 1996; McLaughlin et al. 2006), and a large body of empirical research literature constantly proposing and evaluating new approaches to design. However, there does not appear to be much research exploring the techniques that software developers use in their day-to-day practice (Weyuker, 2011) and which of these techniques they find most useful.

3.2.4 Experimental Design

To address the goal of discovering how practitioners approach design quality, an online questionnaire consisting of 37 questions, in three sections, was developed. The first section explored high level views on design quality: how it is ensured by practitioners, its relative importance compared to functional correctness and the development methodologies used. The second, main section, explored opinion on well-established design guidelines and practices. The third section captured demographic data. Questions consisted of fixed Likert-scale options followed by free-text fields intended to gather more detailed insights.

3.2.4.1 Subjects

The target population for the survey was software developers with industrial experience using object-oriented technology. Obtaining access to a representative sample of this population was a major challenge in performing the study. Industrial software development is a worldwide activity currently involving perhaps as many as eleven million professional developers[5]. As Zhang and Budgen recognise *"… this particular community is defined in terms of their specific skills and knowledge, rather than by location or through membership of any formal organisation, there is no way of knowing the size of the target population…"* (Zhang and Budgen 2013).

3.2.4.2 Objects

Using the template suggested by Linaker et al. (Linaker et al. 2015), the *Target Audience*, *Unit of Analysis*, *Unit of Observation*, and *Search Unit* for this study are all: *Software developers with industrial experience using object-oriented technology*. The *Source of Sampling* are the authors' contacts in the UK software industry and software engineering papers published in the period 2009-2014 by authors who held an industrial affiliation.

3.2.4.3 Instrumentation

The design of this survey was based on empirical advice popular in the related studies (Pfleeger and Kitchenham 2001). The questionnaire was based on a mixture of closed questions and open free-text questions which were intended to gain deeper insights into the participants' responses.

The questionnaire was developed, distributed and analysed using the Qualtrics Survey Software[6]. Qualtrics contains professional-grade survey building tools, including support for questions paths and customisation by scripting and stylesheets. Further analysis of the results was carried out using nVivo[7].

[5] http://www.infoq.com/news/2014/01/IDC-software-developers
[6] http://www.qualtrics.com/
[7] http://www.qsrinternational.com/

The questionnaire consisted of a mixture of Likert-item questions, mostly with five items including a neutral response in the middle. Including a neutral option avoids forcing a positive or negative choice (Shull et al. 2008), which seemed appropriate given the exploratory nature of the survey instrument. To increase respondent speed and comprehension, a similar question layout was used throughout the questionnaire. Each question topic finished with a free-text box intended to gather more detailed insights into the participants' responses.

The questionnaire landing page stated the study aim, identified its authors and gave their contact details. It confirmed that the survey had university ethics approval (obtained 12/6/2014). It stated that questionnaire was only concerned with object-oriented design and defined design in the context of the survey: *"The term design is used in this survey to discuss the organisation of packages and classes, class identification and interaction, and interface use."* This was followed by a section on privacy which explained how responses were to be kept anonymous, how the data was to be analysed, how long data would be retained, how to quit the questionnaire, if required, and how to contact the survey authors. There was also a consent section which blocked access to the rest of the questionnaire until consent was given.

The first section of the survey explored the participant's high-level views of software design quality: how they assess quality, their confidence in design decisions that they make, the relative importance of design quality compared to functional correctness, and the development methodologies that they use.

The next section explored how design guidelines for each of the following were used in practice (a questionnaire page dedicated to each in turn): program to an interface, inheritance, coupling, cohesion, size, complexity and design patterns. Again, the questions were a mixture of Likert-item followed by free-text questions to explore each topic further.

Demographic questions were included at the end in order that these questions did not deter potential respondents from completing the questionnaire; as Kitchenham and Pfleeger note this is suggested by Bourke and Fielder (Kitchenham and Pfleeger 2002b). These questions were used to screen participants, and to provide a general summary of participants' backgrounds.

3.2.4.4 Data Collection Procedures

The questionnaire was initially distributed by email in July 2014 to 48 of the authors' university contacts who worked in the UK software industry. At the same time, the questionnaire was published to various relevant social network communities and 34 email invitations were sent to business incubators affiliated with the Scottish Informatics and Computer Science Alliance (SICSA). The survey therefore used non-probabilistic sampling, convenience and snowballing (Kitchenham and Pfleeger 2002a).

Two months after the initial distribution the authors then targeted a more global population using industrial authors who had recently published design-related research in key software engineering destinations. A range of software engineering journals and conferences (EMSE, ICSE, ICSM, IEEE Software, TSE, IST, SPE, SQJ, OOPSLA, SPLASH, MOBILESoft) were searched for the period 2009-2014 for software design-related papers with an author who appeared to have an industrial affiliation. Many of the publications printed the author's email address. In cases where the email address was missing a web-search was performed to discover whether the author's email address was publicly available in another source e.g. the individual's web page. This resulted in the identification of a further 275 contacts.

Each contact was then sent a short email. This email introduced the authors and the purpose of the survey. It explained why they were being approached e.g. as an author of a technical paper in an identified journal or conference and their affiliation. The email stated that the questionnaire should be completed by programmers, designers or architects based on how they approach object-oriented design quality in practice. The email also stated that the questionnaire took between 15 and 30 minutes to complete (depending on how much text was written in the open-ended questions). The email also included an anonymised, reusable link to the questionnaire. (The Qualtrics survey system uses a machine's IP address to identify unique participants. This information is not available for inspection, and is destroyed on completion of the survey. Duplicate responses are prevented using cookies.) The invitation email also included a request that the email be passed on to other relevant parties e.g. within the recipient's organisation, if appropriate (snowball recruitment).

The survey was closed at the end of December 2014, having run for six months.

3.2.4.5 Analysis Procedure

For each of the closed questions the data is summarised and presented *as is*. The questionnaire contained 14 free-text questions that yielded over 25k words of feedback. The approach to qualitative analysis of free-text responses was similar to that adopted by Garousi and Zhi – synthesis, aggregation and clustering of responses to identify key categories (Garousi and Zhi 2013).

There are many approaches available for coding such qualitative data (Saldaña 2015), here an iterative approach was used to allow the data to inform the coding schemes:

- Holistic coding – collecting all the relevant sources for each basic theme or issue together for further analysis.
- In vivo / verbatim coding – short words or phrases from the material – this may reveal major subjects, topics or themes that are explicit in the material.
- Descriptive / topic coding – identifying the topic, not necessarily the content of the material.
- Versus coding – any discussion of mutually exclusive activities, policies etc.

Each of these types of coding was performed in its own pass over the free-text material. The coding process was iterative – with topics and relationships becoming more refined as more data was analysed. This inevitably carries the risk of bias – as coding is heavily dependent on the researcher. In this case, there was no pre-determined coding scheme and the aim was to let the responses *speak for themselves*.

Coding was used to identify key topics in the responses for each individual free-text question. In the topic tables that follow, the first column identifies the topic code, the second column lists illustrative quotes for that topic, and the third column shows a count of how many respondents mentioned the topic for that question. It should be noted that this count of the number of mentions is not viewed as a sample from any population, but rather only an indication within this sample about popularity.

To save space, only the most popular topics for each question are included in this thesis along with one or two illustrative quotes. Occasionally, where an illustrative quote was judged to be particularly interesting, some less popular topics have also been retained. As a result, some detail is inevitably lost to the reader - the complete set of raw data and full results tables are available in the online experimental package (Stevenson and Wood, 2017).

3.2.4.6 Evaluation of Validity

In this survey, it is recognised that there are numerous important threats to validity. In terms of construct validity, perhaps the biggest concern is the choice of design topics explicitly covered by the survey. It may be that other design topics should also have been explored, however the length of the questionnaire, and the associated time taken to complete it, were major concerns. Most questions had an 'other' option and associated free-text fields - there was little indication from respondents that important topics had been omitted.

The pilot study was intended to help identify any major concerns with the design of the questionnaire. As a result, the original questionnaire was reduced in length, some potentially confusing terminology was clarified, and questions exploring the Law of Demeter (Lieberherr et al. 1988) were removed. An estimation of the time to complete the questionnaire was also included in the introduction.

The most pertinent threat to internal validity is the methodology used to derive the key findings and associated conclusions. The key findings were derived using a mixture of quantitative and qualitative analyses, in attempt to distil the main messages. In a similar manner, these key findings were then merged together to form the main conclusions and answer the initial research questions. This procedure is open to researcher bias and others may have identified different findings. To address this, all the original questionnaire data has been made available publicly. In the analyses, there are traceable paths from the questionnaire data to the key findings (KF1-15) and then to the corresponding high-level conclusions and answers to the original research questions.

The selection of participants for this study is also a concern when considering internal validity. Initially, participants were identified using industrial contacts of the authors and their university and, thereafter, by searching for industrial authors in the academic literature. This is clearly a limited selection procedure. Furthermore, the respondents who completed the questionnaire have been self-selected and may have biases and opinions which they were keen to express, quite possibly views that are not representative of the mainstream. Also, there may be a tendency for subjects to give known, *best practice* responses. Lastly, there was very limited control of the questionnaire once distributed, it is possible that the questionnaire could be completed carelessly or even dishonestly. It is difficult to address such concerns, though, given the level of consensus and quality of free-text responses, it is believed that these issues have not had a major impact on the findings.

The ordering of questions is another potential threat, again, particularly, the ordering of specific design topics covered. There is the potential for *rating fatigue* (Zhang and Budgen 2013) where respondents lose interest as they progress. There is little indication of this, with generally positive responses in initial questions, then more mixed views on inheritance, then being more positive again on coupling etc. and finishing with more mixed responses on design patterns and refactoring.

There are recognised concerns regarding external validity and the generalisability of the findings. The target population was the millions of industrial software developers working around the globe. A key question is whether the actual population sampled is a representative subset of that target population (Kitchenham and Pfleeger 2002c). The respondents in this survey appear to have more industrial experience compared to related work (45% have over 10 years (Figure 3)), work in a wide range of locations around the world, are mostly active programmers (93/102) and have experience of multiple software development activities. The textual feedback provided in the questionnaires showed many respondents responding thoughtfully and knowledgeably which increases confidence in the relevance of the results.

Finally, the number of respondents is relatively small (102), certainly in comparison to the potential size of the target population. However, this size is in keeping with that of recent, comparable studies.

3.2.4.7 Pilot Study

Prior to formal and widespread distribution, the initial questionnaire was evaluated in a pilot study. The questionnaire was sent to ten individuals asking them to complete it and provide feedback on the content and form of the questionnaire. The ten included six individuals from the target population, two researchers with software development experience and two academics with experience of survey research. The pilot was distributed on 23/06/2014, a reminder was sent on 4/7/2014 and the pilot study closed on 11/7/2014.

Four completed and two partially completed questionnaires were received. Only one of the pilot respondents appeared to complete the questionnaire in a single session, taking 36 minutes. Four of the respondents indicated their number of years working in industry: 1 had 2-5 years, 2 had 6-10 years, and 1 had 11-20 years. The respondents used the (then) generous free-text spaces in the questionnaire to provide feedback. As a result of feedback on the questionnaire structure the following changes were made:

- Some of the optional free-text boxes were removed as these had been perceived as mandatory or excessive despite being marked as optional.
- A question about the Law of Demeter was removed as it seemed some respondents were not familiar with this concept and therefore it did not appear to meet the requirement for a commonly known guideline.
- Some terms that appeared to cause confusion (e.g. framework, pattern) were clarified.
- The options in some questions were simplified.

The final version of the questionnaire consisted of an initial consent question and then 37 questions: 29 topic questions (15 multiple-choice, 1 numerical, 13 free-text), 7 demographic questions, and the final general free-text feedback question. A summary of the final questionnaire is available in Appendix D - Survey Questions and the complete questionnaire is available in the experimental package available at: http://dx.doi.org/10.15129/879fb7cf-7f98-49f5-bfa0-183dc5f7d85f.

3.3 Research Questions

The design of the questionnaire was guided by the following research questions:

RQ3.1: Design Priorities - To what extent do practitioners concern themselves with design quality?

This research question was motivated by the fact that there is that much of the research literature and empirical studies (discussed in Chapter 2) that investigate software design quality but there is not much research that actually investigates the importance of design quality to practitioners in their day-to-day work.

RQ3.2: Recognising Quality - How do practitioners recognise software design quality?

This question was motivated by the need to discover how practitioners distinguish *good* (or better) design quality from *bad* (or worse). Again, there seems to be little prior research that investigates how practitioners recognise quality in their day-to-day work.

RQ3.3: Guidance - What design guidelines do practitioners follow?

This question was motivated by the need to discover whether well-known guidelines and practices from the literature are used, and are useful, in practice. Also, which of these are the most useful?

RQ3.4: Decision Making - What information do practitioners use to make design decisions?

This question was motivated by the need to discover the design information that practitioners use to make decisions about quality? Is it information from designs, from code, from metrics, from tools …?

The answers to these research questions are important to researchers to help guide future research; to practitioners to see how other practitioners are addressing design quality; and, to educators to teach students the principles that underpin industry-relevant techniques.

The Goal Question Metric approach (Basili, 1992) was used to derive the research questions from the original research goal. Prior to the distribution of the questionnaire, the research questions were mapped to survey questions, see Table 5.

3.4 Results

3.4.1 Descriptive Statistics

The survey was open for 6 months and received 222 responses, 109 of which were considered valid. No response rate can be calculated as sampling was non-probabilistic. The mean completion time for the questionnaire was 31 minutes, though many responses were completed over multiple days.

Table 5 - Original GQM mapping of survey goal to research questions to answerable survey questions

High level goal: *To what extent do practitioners concern themselves with design quality?*
RQ3.1: Design Priorities – *To what extent do practitioners concern themselves with design quality?* Questions about time spent improving quality, responses where poor quality is the cause of a problem.
Q1, Q3, Q5, Q17, Q19, Q27, Q37
RQ3.2: Recognising Quality - *How do practitioners recognise software design quality?* Questions about recognising quality and responses where following a particular guideline or practice is considered positive/reassuring.
Q3, Q5, Q6, Q7, Q9, Q11, Q14, Q28
RQ3.3: Guidance - *What design guidelines do practitioners follow?* Direct questions about selected guidelines, general question about any other useful guidelines. Responses where a tool, guideline or methodology is mentioned as useful.
Q4, Q8, Q10, Q12, Q13, Q15, Q16, Q18, Q20, Q22, Q24, Q26
RQ3.4: Decision Making - *What information do practitioners use to make design decisions?* Direct questions about design decision making, confidence, and what makes decisions difficult. Responses that mention choice, design decisions, doubt or uncertainty.
Q1, Q4, Q9, Q20, Q11, Q14, Q15, Q16, Q28

3.4.2 Data Set Reduction

222 responses were received in total. Of these:

- 96 questionnaires were complete.
- 93 questionnaires were blank indicating that candidates had seen the landing page and not continued or had answered *no* to the consent question.
- 23 questionnaires were partially completed – these respondents had completed a variety of multiple choice questions, omitting some free-text questions or some of the demographic questions.
- 6 completed questionnaires from the pilot study were included, excluding the data from questions that were removed between the pilot and the main study. This didn't include the two academics as it was known they didn't respond - email tracking was used in the pilot so it was possible to tell who responded but not which response corresponded to which email.

An examination of the partially completed questionnaires found that they appeared to contribute very little – mostly a few questions answered and small comments in the free-text sections. A decision was therefore made to exclude these from the analysis. This resulted in a total of 102 complete questionnaires being used for the main analysis. All 102 respondents stated that they were currently working in industry or had previously done so.

3.4.3 Respondents

One hundred and two respondents completed the questionnaire, 56 from the UK, 40 from all over the world outside of the UK, and 6 who did not provide a location. The self-reported years of experience for respondents is shown in Figure 3. Compared to the experience levels recorded in the surveys discussed in the Related Work section the experience level is high, 45% of respondents in this study have over 10 years of industry experience.

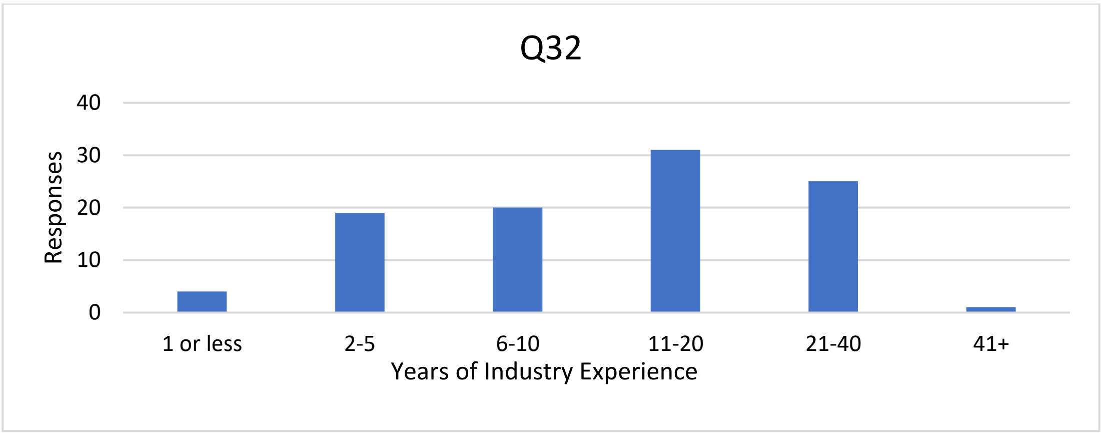


Figure 3 - Q32 - How many years have you been involved in the software industry as a designer/developer? (102 responses)[8]

Figure 4 summarises the roles that the respondents fill in software development (they could fill more than one role). 93 of the respondents have some programming duties. Only two respondents were solely managers – however, these individual's responses in other areas indicate that this is a technical lead, rather than an administrative role.

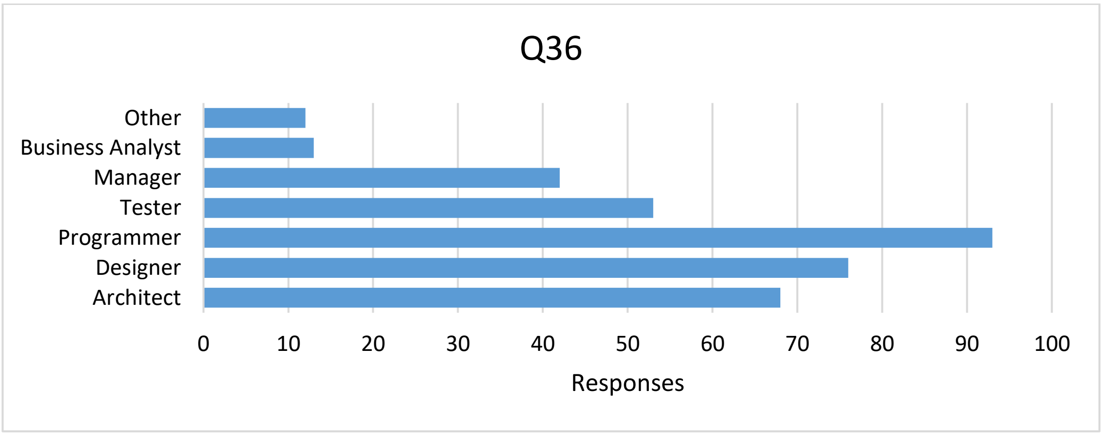


Figure 4 – Q36 - What role(s) have you carried out in a software development environment? (102 responses, respondents could choose more than one option)

[8] Unless shown otherwise, all figures consist of 102 responses.

Figure 5 summarises the programming languages used by respondents (again respondents could choose more than one response). There is a good correspondence with the top programming languages identified by IEEE Spectrum (Cass 2015) which ranked 1: Java, 2: C, 3: C++, 4: Python, 5: C#, 6: R, 7: JavaScript.

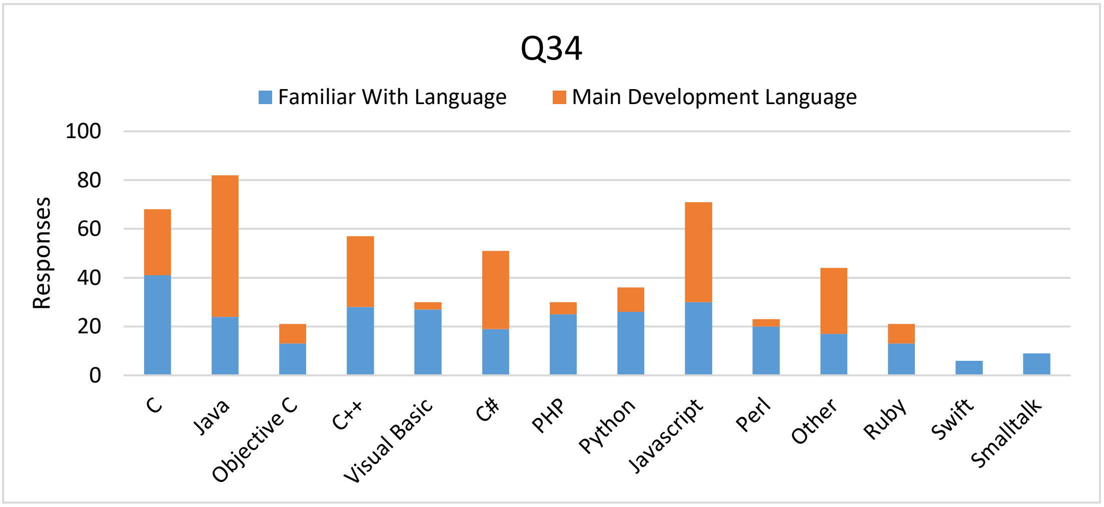


Figure 5 – Q34 - What programming language(s) do you use when developing software? (102 responses, respondents could choose more than one)

As the study used non-probabilistic sampling, the findings cannot be generalised to a wider population (Stavru 2014). However, the respondents do appear to represent a diverse group of experienced practitioners and the common findings should therefore provide a useful indication of widely used approaches to design quality. Overall, it does appear that the respondents are all from the target population. Figure 6 shows the application domains that respondents work in.

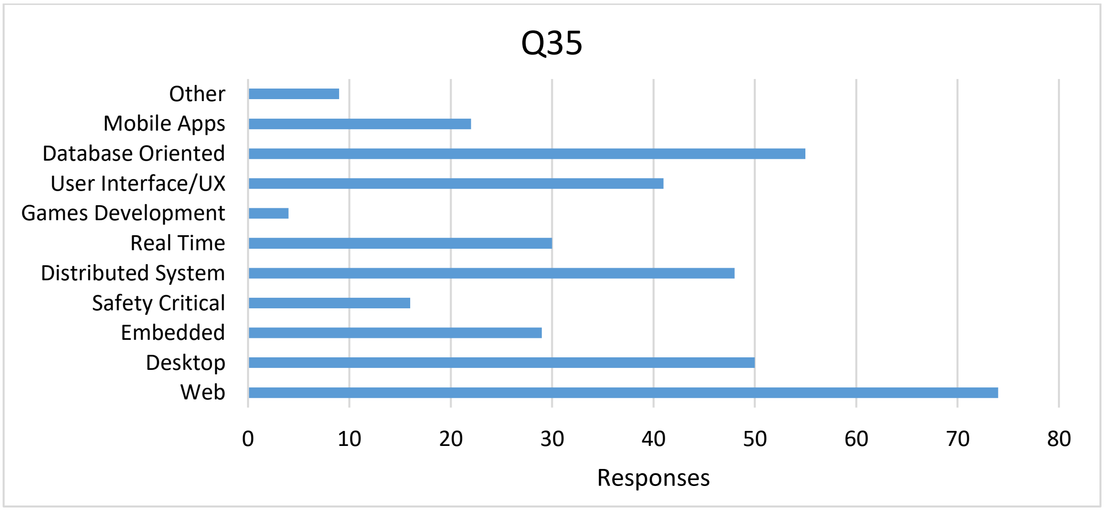


Figure 6 – Q35 - What kinds of problems are you most familiar with? (102 responses, respondents could choose more than one)

3.4.4 General Approaches to Design Quality

Question 7, the first Likert-item question that focussed on design quality, asked respondents about their general approach to ensuring quality in their development process. The results are summarised in Figure 7. Note that diverging stacked bar charts (Figure 7, Figure 9, Figure 11, Figure 12, Figure 13, Figure 14, Figure 15) show frequency data aligned along the *neutral* category. This emphasises the relative overall *swing* in the data, while allowing comparison between the relative proportions of each bar (Robbins et al., 2011). Note that the grey bars adjacent to the vertical axis indicate *non-use* and are thus set aside.

Q7. How important is each of the following in ensuring the quality of software design in your development process?

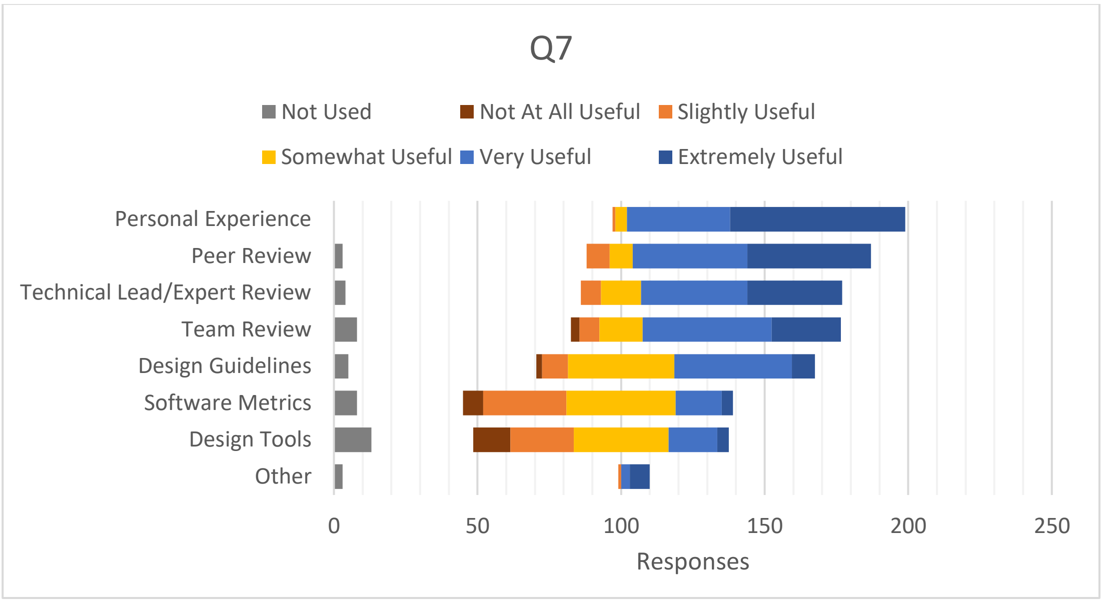


Figure 7 – Q7 - Activities ensuring software quality. Note that the categories have been ordered from most to least positive (total of Very and Extremely Useful). Counts for 'not used' are shown as grey bars for clarity.

Responses show a clear bias towards personal experience – this is the strongest response for each of the categories, followed closely by peer review, team review, and expert review – all people-mediated inputs. This leads to the first key finding (KF) from the study.

Table 6 shows the free-text responses associated with Q2 – further details on ensuring quality in the development process. The topic code for each category of response is shown in the first column, illustrative examples are listed in middle column, and the total count of respondents who made responses in that category is provided in the third column. The complete set of response data is available in the online experimental package.

The free-text responses in Table 6 might be summarised as *knowing what to do* – whether this is from previous experience, a review process, guideline, process, specification, or domain knowledge. The feedback on experience might be separated into two further categories:

- General conscientiousness, following guidelines and design practices (discipline)
- Prior experience with a specific technology or problem domain (specific knowledge), allowing the avoidance of common pitfalls

Table 6 - Q2 responses – Further details on how software quality is ensured in the development process (example free-text responses)

Topic Code	**Illustrative Quotes**	**Mentions**
Review	"…tight review by 1 or 2 technical experts."	10
Personal Experience	"I think personal/peer experience is the starting point, as you can't consider using what you/your team don't know about."	9
Tools	"…with our build system, Sonar by SonarQube is ran and this runs Checkstyle, EclEmma, ..."	9
Peer Experience	"…we are much more reliant on personal experience, team experience and access to domain experts than design guidelines. The latter can only take us so far..."	7
Design Space Problem	"…there is no perfect choice but there are always different ways to reach your goal."	5
Teamwork	"…a general consideration and acknowledgement that others will have to use your code once you're done is IMO the single most important factor. There are experienced people who will produce lower quality designs than less experienced but conscientious designers…"	4
Processes	"Separating design from implementation is itself an impediment to good design. Good design depends on feedback."	4
Guidelines	"…professional guidelines are very helpful."	4
Feedback	"…feedback that matters is that which comes from end users and those concerned about the long-term evolution of the system."	3
Design for change	"…Re-architecting an existing design is rarely a simple task and so is often not a desired option."	3
Domain Familiarity	"Generally our team deals with a certain business domain… we can have a good idea whether or not a design proposal will work…"	3

The next general design question was concerned with confidence in design decisions, see Figure 8 (Q3). The quantitative results suggest generally high confidence, however analysis of the free-text responses (Table 7) revealed a range of situations where there is less certainty. Many responses highlighted uncertainty in terms of the problem domain, requirements, choices amongst design options, lack of control, and design complexity.

Q3. When making design decisions, how confident are you that you have chosen the best option from the alternatives you have considered?

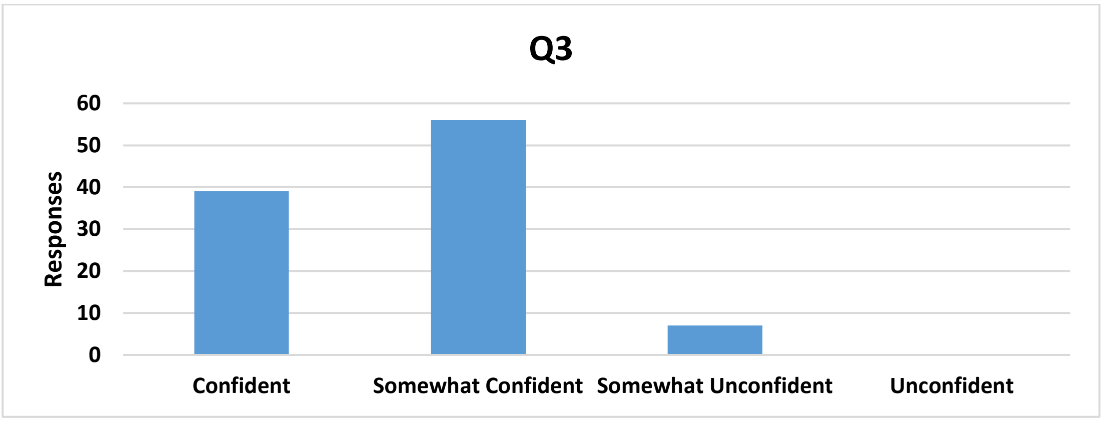


Figure 8 – Q3 - Confidence level when making design decisions

Table 7 – Q4 responses - What type of problem makes it difficult to have confidence in a design decision? (sample free-text responses)

Topic Code	**Illustrative Quotes**	**Mentions**
Knowledge Gap	"…lack of familiarity with the problem generally leads to more iterations over the designs as the project moves forward." "...inexperience with any of the technologies involved and the feasibility of what can be done with them."	20
Unclear Requirements	"Customers who don't understand their own problem."	18
Choice Among Options	"Many options may be good, and weighing their advantages and disadvantages is an inexact science"	17
Changing Requirements	"It is far more time and cost efficient to be able to refactor code when we know how it is really being used, rather than try and concoct a perfect plan first time around."	15
Lack of control	"Confidence levels can drop when the use of external resources … is used as part of an overall project."	14
Unknown Future Use	"The best you can do is design for the requirements you were given, and with a bit of foresight…"	13
Complexity	"Convoluted designs almost always suggest there's something wrong with design, regardless of system complexity"	6

The next question (Q5) asked how designers recognise design quality. Table 8 summarises the main responses. Many topics are associated with well-defined practices such as Clean Code (Martin 2009): clarity of design including naming, small size, simplicity, modularity, documented, full set of tests.

The other group of topics are much less concrete and relate back to the emphasis placed on experience in responses to Q1, intuition and even design hindsight: maintainable, clarity, complete, subjective (“*design is an art*”), reflects the problem domain.

Table 8 – Q5 - What are the key features that allow you to recognize good design in your own work and in the work of others? (sample free-text responses)

Topic Code	**Illustrative Quotes**	**Mentions**
Maintainable / Extensible	“…well designed components are easy to maintain over time though it takes longer to recognise this…”	31
Clarity	“A good design should make it as simple as possible to understand the responsibility of each piece of code, and how they interact.”	28
Modular / Separation	“Partitioning of functionality across interfaces i.e. a target balance of loosely coupled components. I tear my hair out on a regular basis when I see components poisoned with dependencies between logical boundaries (this is something you always see with graduate programmers but never with older hands).”	18
Smallness / Simplicity	“…no genius code where a task is compacted into an unmaintainable mess.”	18
Tested / Testable	“…code that doesn't have a decent set of tests is badly designed…”	14
Complete	“Easy to understand, reuse, alter and maintain without needing to seek external explanation” “…you should be able to read the code as if it were written in English, without any comments…”	13
Documented	“…programmers who think their code is 'self-documenting' or who give other excuses for failing to document are usually incompetent, in my experience.”	12
Quality is Subjective	“There's a quote from R. Buckminster Fuller ‘When I am working on a problem, I never think about beauty, but when I have finished, if the solution is not beautiful, I know it is wrong.’” “…mostly gut feeling, I think design is an art.”	10
Reflects Problem	“How well the model describes the domain and how little the technical platform has influence on the model”	10

There then followed a pair of questions that explored the relationship between functional correctness and design quality. The first of these (Q6) asked about the importance of functional correctness and design quality in the respondent’s work – see Figure 9. The second (Q7) asked about the split of effort between focusing on correctness and design quality – see Figure 9.

Functional correctness was defined in the questionnaire as "*Ensuring that the software being created will meet the specified requirements*". Design quality was defined as "*The suitability of the structural organisation of packages and classes, class identification and interaction, and interface use*".

Most respondents (83) stated that functional correctness is *Very Important* in their work, with most of the rest (15) stating that it is *Important*. The responses for design quality (39 *Very Important* and 53 *Important*) suggest that respondents, while not viewing design quality as important as correctness, still view design quality as an important consideration in their development work.

Q6. How important are the following elements in your work?

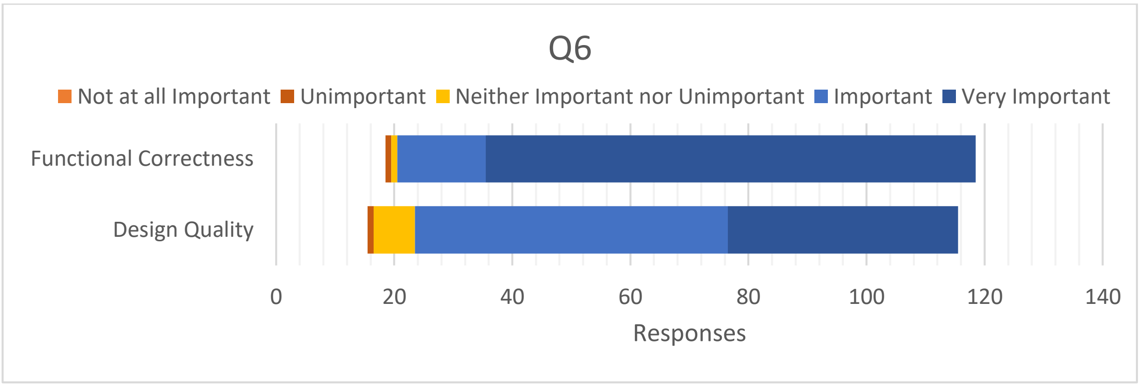


Figure 9 – Q6 - Importance of Functional Correctness vs. Design Quality
(99 respondents answered both questions, 2 answered one each, 1 answered neither question)

Figure 10 shows the distribution of effort spent on design quality compared to functional correctness, where each point is an anonymous respondent. Note that respondents could choose *Other Tasks* which is why many of the respondents are not on the diagonal representing a time split totalling 100%. The focus of this question was the split between functional correctness and design quality so no further questions were asked about other tasks.

The chart has been scaled to 100% on each axis and the data points have also been jittered by <=2. (Jittering is the process whereby a small random amount is added or subtracted from each point to avoid overlapping points in scatter charts (Kitchenham et al. 2002)). The tendency towards the top left in Figure 10 shows respondents favouring functional correctness over design quality.

Q7. In the course of your software development work, approximately what percentage of your effort is split between the following tasks? (Ensuring Functional Correctness; Ensuring Design Quality; Other Tasks)

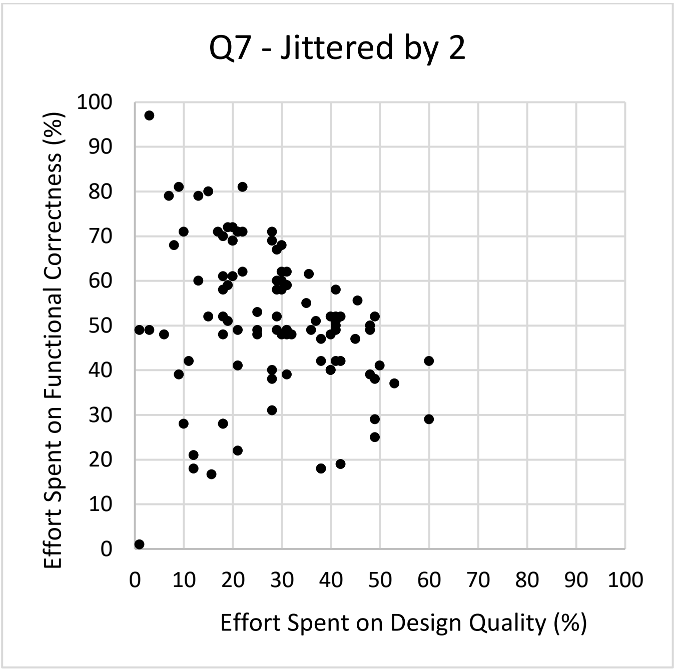


Figure 10 - Q7 - Effort split between correctness and quality
(Note that the data points have been jittered by <=2 on each axis to avoid overlapping points)

Overall, Figure 10 shows most responses around two-thirds correctness and one-third quality (or about twice the effort spent on correctness compared to design quality), but also highlights a large spread in responses. Almost all of the respondents spend some time ensuring functional correctness, with a few respondents almost exclusively working on this aspect. Some respondents work more on design quality (those to the right of the diagram) – but this aspect has no dedicated full-time practitioners. Finally, one respondent pointed out the importance of both: "*These are not mutually exclusive activities, and unless you're doing both all the time you aren't doing your job.*"

The final general design question (Q8) explored the development methodology that most influenced how practitioners do design. The key themes that emerged from this question are generally well known – the importance of an iterative approach and design for change, and the pressures of time and commercial factors. A more general finding was the need to adapt the methodology to each new project rather than always adopting the same set process.

As noted in the introduction, key findings will be listed as *{KF 'chapter number'.'finding number'}* for ease of reference.

*Approaches to Design Summary: [KF 3.1] People-mediated feedback, including personal experience, is the primary means of ensuring software design quality***.** *[KF 3.2]: Respondents were generally confident about design decisions that they make. Issues that reduce confidence include: lack of knowledge about the problem domain or technologies used; having a range of design options; imprecise or changing requirements; lack of control over all aspects of the system; design complexity. [KF 3.3] Respondents identified many well-defined practices that help improve design quality: clarity of design especially naming, small size, simplicity, modularity, documented, full set of tests. [KF 3.4] Respondents identified a range of desirable properties for designs that are hard to define and acquire and which seem derived from experience, intuition and hindsight. [KF 3.5] Almost all respondents see both functional correctness and design quality as important or very important in their practice. Functional correctness is more important than design quality with respondents typical spending as much as twice the effort on functional correctness as design quality.*

3.4.5 Design Practices

Each page regarding a particular guideline or aspect of design featured an initial question about the importance of that topic in the course of design work. The results of these initial questions are compared in Figure 11.

Figure 11 shows that, overall, there is positive support (blue segments) for all of the guidelines. Coupling received the most positive responses, followed by method complexity. There is a notable drift towards ambivalence or disregard for some aspects – with controversial topics receiving a more mixed response.

There then followed a series of questions that explored well known design practices.

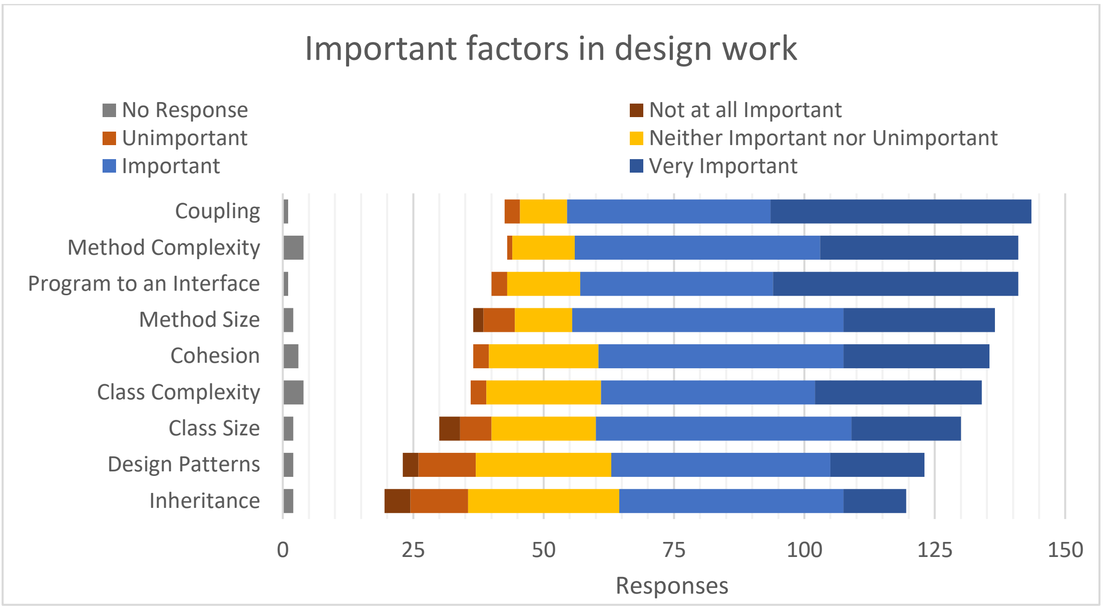


Figure 11 – Q10, 13, 18, 20, 22, 24, 26 – Importance of various aspects of guidance in design work, note that the vertical categories are sorted by total positive responses (Important + Very Important)

3.4.5.1 Design Practice – Program to an Interface

The first of these (Q10) asked about *program to an interface* (Gamma et al. 1994). A follow-up question (Q11) asked about specific factors that help to decide when to use an interface in a design, rather than a concrete (or abstract) class.

Figure 11 shows that there is positive support for the guideline program to an interface, with 84/102 rating it *Very Important* or *Important* and no respondents rating it *Not at all Important*.

Figure 12 illustrates that there is strong support for many of the program to an interface factors with over 60% of respondents seeing avoiding dependencies, multiple implementations, dependency injection (Fowler 2004) and containing change as *Very Important* or *Important*. On the other hand, there is still a notable group of respondents (at least 25%) who view these criteria neutrally or as unimportant. It is possible that program to an interface is not important in the particular development context in which these respondents work, rather than them viewing the practice itself as unimportant.

Q11. How important are the following factors when deciding if an interface should be used?

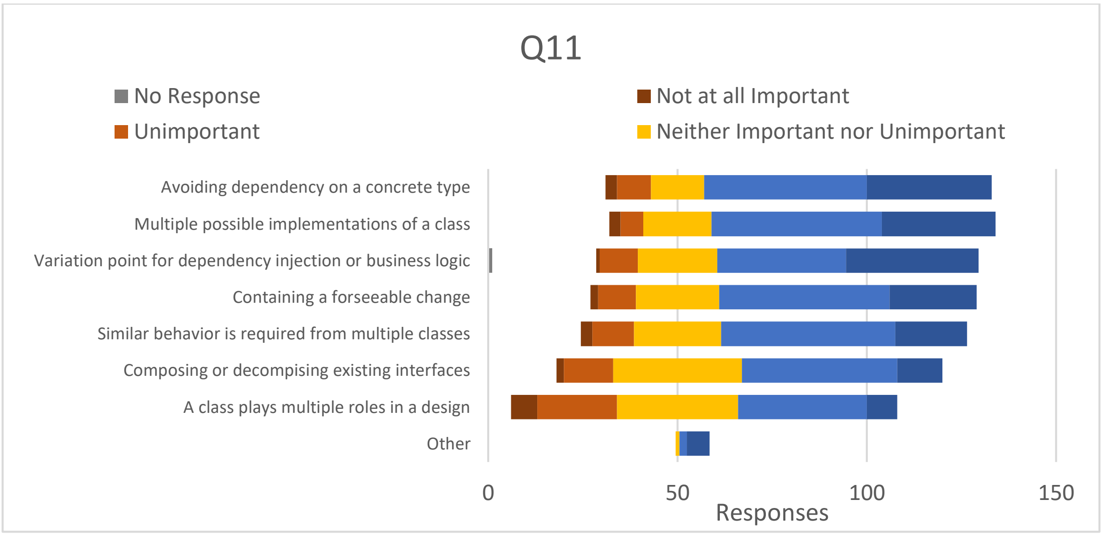


Figure 12 – Q11 - Factors for deciding on Interface use
Note that the vertical order is in most-to-least positive (total of the 'important' responses in each category)

Table 9 shows the free-text responses for *program to an interface*. From these responses, the key factors that are identified include isolating systems, dependency injection, single responsibility, and aiding testing. It is also important to note the warnings regarding potential overuse of interfaces potentially "*cluttering the code base*".

Table 9 – Q12 - Feedback for Use of Interfaces (sample free-text responses)

Factor	**Examples**	**Count**
API / Isolating Systems	"Abstractions for core infrastructure classes such as repositories, data storage, messaging etc." "…layer boundaries in application design."	7
Dependency Injection	"Depending on a concrete type becomes acceptable with dynamic or configurable composition."	6
Testability	"As that module inherits that interface and we can run tests against that interface out of the system we can hot-swap a module in a live system without worry of error."	5
Composability / Abstraction	"Reducing each class to a single role gives better compartmentalisation and composability…"	5
Overuse of Interfaces	"Often the answer is: it depends. Overuse of interfaces can clutter the code base and make changes tedious."	5
One Role per Class	"Multiple interface inheritance (beyond obvious ones like IDisposable) is a code smell"	4

3.4.5.2 Design Practice – Inheritance

The next section of the questionnaire explored inheritance – its importance (Q13), factors influencing its use (Q14), and the design of hierarchies (Q15).

Figure 11 shows that there was more ambivalence regarding inheritance use than *program to an interface*, although overall there were still more respondents rating it *Very Important* or *Important* (55) than *Unimportant* or *Not at all Important* (16).

Figure 13 shows that Code Reuse, Type Substitution and Future Flexibility are all regarded positively (*Very Important* or *Important*) by over half of the respondents (with very few identifying *Other* reasons for inheritance usage). It is notable, however, that some respondents consider these inheritance factors *Not at all Important*.

Q14. What factors or indicators do you consider to be most important when deciding to use inheritance?

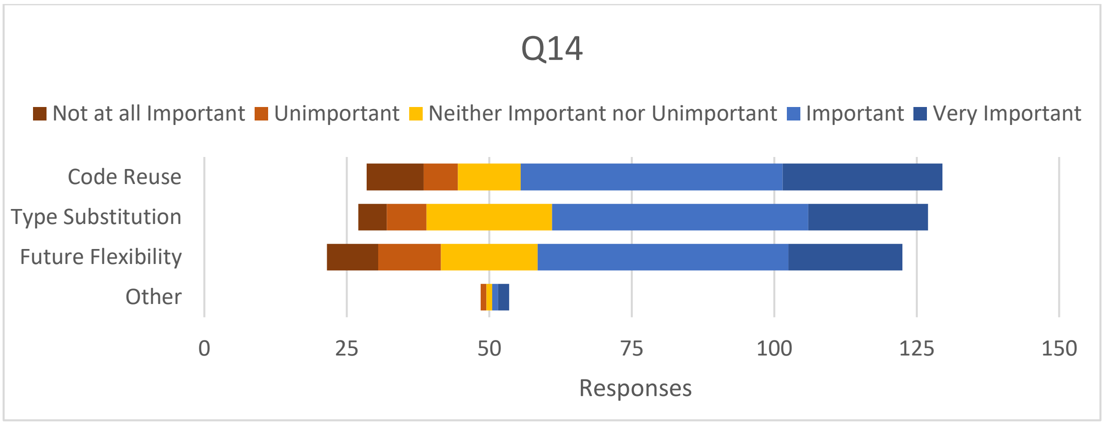


Figure 13 – Q14 - Importance of selected factors when using inheritance

The free-text responses for Q15 (Table 10) indicate that code-reuse is the most commonly mentioned motivation for using inheritance, followed by reuse of concepts. However, in keeping with the responses to Q14, there are many negative responses regarding inheritance: not used (code smell), prefer composition (Gamma et al. 1994), as well as the view that inheritance hierarchies should be kept shallow (Daly et al. 1996), and that inheritance should follow a type substitution relationship (Liskov, 1988).

Figure 14 shows the factors considered important when designing inheritance hierarchies. There is relatively strong support for consideration of inheritance depth, more modest support for consideration of dependency inversion, and more mixed opinions on inheritance width.

Table 10 – Q15 – Deciding on abstract or concrete inheritance

Factor	**Examples**	**Count**
Code Reuse	"…very important so as not to use the same code over and over…"	12
Behaviour / Concept Reuse	"… identify concepts which are core and likely to evolve on the same path and refactor those to use a common base."	9
Not used	"…avoid inheritance where possible … it's always ended up biting me." "Creates code which is hard to read, difficult to maintain."	9
Prefer Composition	"Composition of behaviour has proven to be far more flexible and reusable."	7
Flexibility	"Future flexibility … Abstract Inheritance over Object Inheritance."	5
Dependency Injection /	"…it's easier and generally considered safer and more testable/scalable, to inject interface dependency and use composition over inheritance."	4
Framework / API	"…abstract inheritance which allows us to share code and API between very similar classes	4
Shallow Hierarchy	"…very shallow hierarchies (probably just one (maybe abstract) parent with few direct children), every child class 100% 'is a' [parent class]."	4

Q16. When designing inheritance hierarchies, how important are the following features?

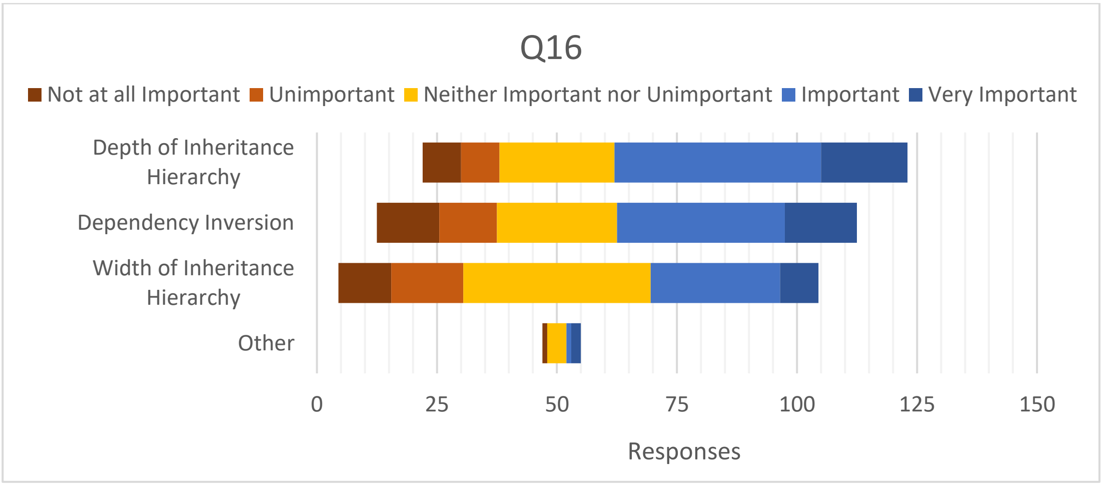


Figure 14 – Q16 - Importance of selected features when designing inheritance hierarchies

The free-text responses for Q17 (Table 11) provide useful insight into respondents' thoughts on the use of inheritance in object-oriented designs. Again, there are strong views that inheritance should be avoided, and composition preferred, or, when used, it should be limited to inheritance from abstract classes only. In terms of hierarchy design, the most common views suggest that inheritance hierarchies should be shallow, model the problem domain, and adhere to is-a relationships. Good tool support was recognised as important in understanding inheritance hierarchies.

Table 11 – Q17 - Dealing with inheritance

Factor	Examples	Count
Hierarchy Structure	"If a design calls for a deep inheritance hierarchy, so be it. Ditto for the width…" "You don't want your ears to pop when traversing down the inheritance hierarchy."	13
Avoidance	"…we very rarely find ourselves using inheritance."	12
Side Effects	"…derived types must satisfy the Liskov Substitution Principle … very difficult to achieve, so we try to use composition…"	7
Tools	"…modern IDEs help, but they only help traverse the hierarchy, they don't necessarily aid in understanding the structure."	5

3.4.5.3 Design Practice – Coupling

Respondents were very positive about the role of coupling in considering design quality with 89/102 rating it at least *Important* and none rating it *Not at all Important* – see Figure 11. Coupling received the most positive responses of all the design considerations presented in the questionnaire.

The free-text responses in Table 12 emphasise the popularity of coupling as an indicator of design quality with many practical approaches to its detection. These approaches include the use of tools (SonarQube) and methods such as dependency graphs. Recurring themes of experience and ease of testing were again highlighted, along with guidelines on keeping class interfaces small and simple, being able to understand classes in isolation, and avoiding access to, and minimising knowledge of, the state within other classes. One reason why coupling is popular may be the range of practical techniques for detecting it.

Table 12 – Q19 - Dealing with coupling

Factor	Examples	Count
Tools and Methods	"SonarQube, JUnit and the overall TDD approach." "Dependency graphs and a well understood domain model…"	19
Guidelines	"'Keep It Simple', 'Tell don't Ask' and 'Open/Closed' (from SOLID) …" "Essential coupling in the business should show up as essential coupling in the code."	10
Technique	"Explicit API between classes and modules, and prohibiting classes from accessing others' internal state."	10
Smells / Problems	"…if you find yourself having to mock too many other objects … your subject under test probably has too many dependencies."	5
No issue	"We don't lose sleep over coupling"	1

3.4.5.4 *Design Practices – Cohesion*

The summary of responses to Q20, included in Figure 11 shows a strong positive response (75/102 rating it at least *Important*).

A theme that emerges from the free-text responses on cohesion (Q21), Table 13, is a difficulty in providing clear guidelines for detecting good or bad cohesion – many of the guidelines were statements of other design principles e.g. *Low Coupling* or small size. Indeed, one of the responses acknowledges this *"It's quite hard to explain and measure cohesion…"*.

One of strongest positive guidelines was the Single Responsibility Principle (SRP) from SOLID again (Martin 2003). There was some mention of tool use (SonarQube again), metrics and dependency analysis. Once again, ease of testing was highlighted as an indirect indicator of cohesion (difficulty in achieving high coverage).

Table 13 – Q21 - Dealing with cohesion

Factor	**Examples**	**Count**
Guidelines	"…programming to interfaces…should encourage splitting classes…" "Code size is important." "…few dependencies are better…"	9
Tools and Methods	"SonarQube uses the LCOM2 metric." "Dependency Structure Matrix" "…low cohesion…often difficult to write tests for to get high coverage."	7
SRP	"One class should do one thing well and nothing else."	7
Middle Ground	"…aim for high cohesion in models but certain things like logging libraries, utility classes and other things drag it down a bit…"	3
Not an issue	"Not having cohesion is an indication of a bad design, but I never strive to get it for its own sake."	3

3.4.5.5 *Design Practices – Size*

The next question, Q22, explored both class size and method size. Figure 11 includes these responses showing that a clear majority of respondents see both class size and method size as important indicators of design quality. There is a slight indication (about 10% of respondents) that method size is more important across the response categories. Around 10% of respondents suggest that size is not important to design quality.

In the free-text responses (Q23) many respondents expressed rules of thumb when considering class and method size. For example: *"A method should rarely exceed the size* *of one page on the screen"*. What constitutes a *screen* varies amongst working environments - Table 14 summarises the specific size recommendations received**.**

In contrast to the view that size should generally be *small*, another common view was to avoid excessive decomposition - *"the kind of abstraction that discards essential complexity"*. Like some views expressed under the coupling and inheritance questions, respondents suggested that some *substantial* (large, complex, coupled) pieces of code can be central to the design in terms of managing complexity and/or representing the domain. One of the challenges of design appears to be the recognition of such *essential complexity*.

Table 14 – Q23 - Summary of responses on Size issues (LOC = Lines of Code)

Attribute	**Response**										
LOC in a Method	1	"a few"	<= 5	< 8	5 - 10	<= 20 non-comment	<= 30	< 50	half page	page / screen	short, no fixed limit
Agreement	2	1	1	1	2	1	1	1	1	6	6

Class Size	<= 50 LOC	<= 100 LOC	<= 500 LOC	<= 7 non-get/set methods	<= 12 methods	<= 15 methods	< page / screen	< 2 pages / screens
Agreement	1	1	2	1	1	1	3	1

Method Parameters	2-3	3-4	<= 5	< 7		*Inheritance Depth*	<=3
Agreement	1	1	1	1		*Agreement*	3

Practices Summary 1: [KF 3.6] Program to an Interface was identified as an important design guideline by a large majority of respondents (83%). The most important motivators for interface use are: depending on abstractions (avoid direct concrete dependency, multiple implementations, dependency injection), isolating (sub-) systems and aiding testability. Respondents warned against overuse of interfaces and stressed the Single Responsibility Principle in interface design. [KF 3.7] There was a clear spread of opinion on the role of inheritance in design quality with some practitioners considering it a design flaw. Code reuse, type substitution and future flexibility were all seen as positive reasons for using inheritance. Hierarchies should model the problem domain, adhere to is-a relationships and be kept shallow. [KF 3.8] Respondents viewed coupling as an important consideration in design quality (87% rated at least Important), with the most positive responses out of all the guidelines explored. Many different approaches to minimising coupling were identified including: use of tools, dependency analysis, experience, design guidelines, simple and small class interfaces, understanding classes in isolation, avoiding access to state within other classes and ease of testing as an indicator. [KF 3.9] Cohesion was identified as an important indicator of design quality (70% at least Important). Respondents suggest that cohesion is harder to assess than other design properties. Guidance on assessing cohesion included SRP, *use of tools and metrics, experience, and as a by-product of other design guidelines (coupling, size, interface use) and the ease with which classes can be tested. [KF 3.10] Both class and method size were viewed as important design factors (70-80% respondents), though about 10% viewed size as unimportant. Many respondents provided explicit comments on size which tended to constrain classes and methods to be 'small'.*

3.4.5.6 Design Practices – Complexity

The next question (Q24) explored the role of complexity in design quality. Like size there were separate questions on class and method complexity. Figure 11 shows that there was a strong view that complexity is an important consideration in design quality. 70-80% of respondents identified both class and method complexity as at least *Important* and less than 5% viewed it as *Unimportant.*

There were many comments made in the free-text section on complexity, see Table 15. One of the main themes that arose was the relationship with size, keeping classes and methods *small* helps reduce complexity: "*This is the same as size, just measured with different metrics*". Related to this was the recurring theme of striving for simplicity – "*simplify, simplify, simplify*" – and the emphasis once again of SOLID/Clean Code-like principles (Martin 2003, 2009) of separating conditionals into separate methods, one abstraction per method etc. There was some identification of some useful tools (SonarQube) and some metrics: cyclomatic complexity, npath (Nejmeh 1988). Once again, the recurring themes of review, experience and the relationship with the ease of testing were highlighted.

Table 15 – Q25 - Dealing with complexity

Factor	**Examples**	**Count**
Guidelines	"…making sure each class only has a single responsibility… (although at the cost of some increase in the complexity of arranging objects)…"	16
Tools	"Using Sonar is like having a very knowledgeable Java programmer looking over your shoulder…"	13
Definition	"Complexity is everything… the complexity of the solution should match the complexity of the problem."	8
Techniques	"… logical chunks of code are often moved to separate methods…"	5
Experience	"All personal experience, try to keep things as simple as possible."	3
Cause	"… the problem is not sufficiently understood but it can also be an indicator of poor coding and design skills."	2
Problems	"Method complexity is something we consider as it affects our Unit Testing and Code Coverage."	2

The majority of responses suggested that complexity was avoidable and addressable by breaking down classes and methods into ever simpler components, there was also, (again) the recognition that some complexity is *essential* and is better explicitly modelled as such rather than artificially broken down. Pushing complexity out of a class may increase overall systemic complexity - which is less visible: *"I'd rather have a more complex class than one that removes some of the real complexity".* This raises the fundamental question of what constitutes a *good* decomposition for a particular problem: *"We don't know what should be the aspects that need separating, and we don't know when it is worth separating them and when it is not"* (Fowler 2003).

3.4.5.7 Design Practices – Design Patterns

The second last question on design practices, Q26, explored the role and importance of design patterns, see Figure 11. Although there is a similar shape to the distribution of responses (~60% *Important)* there was more ambivalence on the role design patterns than the previous design practices. The free-text responses in Table 16 provide more detailed insight into this.

Table 16 – Q27 - Dealing with design patterns

Factor	**Examples**	**Count**
Positive	"General awareness of design patterns is essential for well written code."	15
Usage	"…applied, but very pragmatically. It's not about using DPs for the sake of using them."	13
Common Language	"…makes it instantly clear what something does and how it should be used, as we all understand the pattern"	10
Specific Pattern	Builder, Factory, Observer, Template, Strategy, MVP, ViewHolder, Singleton, MVC, Composite, Command, Façade	9
Resistance / Controversy	"Don't know of any developer I work with that makes use of design patterns."	8
Guidelines	"Simpler rules like inheritance, compositions, SOLID, etc. are better to master than design patterns."	3
Negative	"…younger programmers tend to shoehorn into every single problem… results in a lot of technical debt that other developers have to maintain"	3
Framework	"…framework we use dictates what design patterns we use to interact with the hooks."	2

Responses were, in the majority, positive, with comments about their use improving the quality of design and also the *common language* and sense of familiarity that patterns provide. However, there were also strong opinions expressed about overuse or blind application of patterns. Another view is that fundamental design principles such as low coupling and SOLID are more important than design patterns. There was also an issue of how patterns should be used in design, up front – *"…identify up front possible design patterns…"* - or emerging as appropriate - *"They emerge, they are not the starting point"* - which seemed to be the more common view and more in line with the original proposal for design patterns (Gamma et al. 1994). Nine design patterns were explicitly mentioned by respondents (five of these corresponding to the six identified by Zhang and Budgen as the most favoured patterns in their survey results (Zhang and Budgen 2013)).

3.4.5.8 Design Practice – Refactoring

The final section explored the use of refactoring to address each of the previously covered design concerns: complexity, coupling, size etc. Figure 15 summarises the responses, again ordered by popularity. Complexity and size lead the *Most Often* category for refactoring, with 62 respondents saying they *Very Often* or *Quite Often* refactor due to complexity issues.

Q28. How often to you alter or refactor a design due to the following considerations?

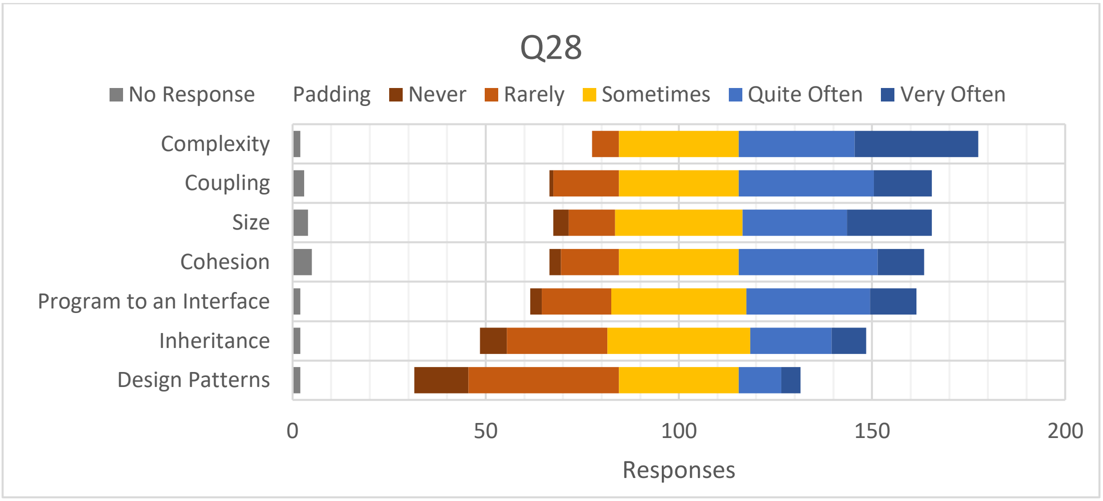


Figure 15 – Q28 - Causes of refactoring

Coupling and cohesion are ranked next. It is striking that the refactoring ranking has a broad similarity to the overall importance respondents gave to the design practices in the previous section (Figure 11). It may also be the case that complexity and size degrade more easily, or are easiest to detect, and are also closely related: *"Size and complexity are often related and are much more likely to get out of hand as time goes by"*. Notably, Program to and Interface (PtaI) is the most mobile factor – being important at the design stage (in Figure 11), but ranking low (by order) for refactoring.

Table 17 – Q29 – Final thoughts on design and assessing quality

Factor	Examples	Count
Methods	"By adopting BDD and TDD methods in order to follow the high-level Red-Green-Refactor cycle you will find that refactoring happens often…" "Refactoring just means that when you touch code, you should strive to leave it in better shape than when you found it." "The Mikado Method"	9
Indicators / Drivers	"Annoyingly, I find a lot of it comes down to experience - sometimes a solution just feels wrong. Once you've refactored it it's easy to show that it's better, but I can't always articulate exactly what's wrong beforehand." "… Bad names allow responsibilities to be assigned incorrectly. Good names make that a little bit harder."	7
Tools / Metrics	"Sonar keeps everyone honest on a day-to-day basis." "Using introspection tools and metrics is important."	5

The free-text responses on design in general (Q29), are summarised in Table 17. Although, for this question, there was a lack of commonality in the responses, they do identify a range of techniques for determining when to refactor: when changes are made code should always be improved, peer review, experience, formal and informal metrics, tools, and as part of test driven development. Finally, there was an illuminating comment on why refactoring is perhaps not practiced as widely as it should be: "*Refactoring is rare, not because it isn't important but because it is rarely the most important thing you need to do next*".

Design Practices Summary 2: [KF 3.11] Both class and method complexity were viewed as important design factors (70-80% respondents), with less than 5% viewing them as unimportant. Practical approaches to managing complexity include the use of tools, metrics and the relationship to small size. More informal approaches include keeping classes and methods 'simple', using experience and reviews, and the ease of testing. There was recognition that 'essential' complexity in the problem or design domain should be kept together rather than artificially broken up. [KF 3.12] There were mixed views on the role of design patterns as a contributor to design quality, though, overall, most respondents viewed them as at least Important (~60%). They contribute positively to design structure and common understanding. They can also be over-used. There was mixed opinion on whether patterns should be used up-front during design or emerge where necessary and appropriate. [KF 3.13] Strong reasons for refactoring were complexity, coupling, size and cohesion, in that order. The relative importance of reasons to refactor broadly matched the overall *importance of the corresponding design practice. Developers use experience, peer review, formal and informal metrics, tools and test-driven development to determine when to refactor. Ideally, code should be refactored whenever it is changed but other pressures can make this unrealistic.*

3.4.6 Experience, Programming Languages, and Development Role

The questionnaire data was also analysed to investigate whether there was any noticeable difference in responses based on experience, main programming language used, and software development role. Rather than *fishing* for statistically significant results the response data was explored visually using the Qualtrics software. Given the relatively small number of respondents overall, and the even smaller number in each sub-category, the aim was to check if there were any differences amongst responses based on subject background. (Only 97 of the subjects identified their years of experience and so the total numbers shown below are less than the 102 shown elsewhere in the thesis.)

For each of the main results reported in this thesis (17 in total), a series of graphs was constructed which enabled visualisation of any major differences in response based on sub-category (experience, language or role). For almost every result there did not appear to be a major difference amongst sub-categories. Figure 16 and Figure 17 shows the typical shape of the graphs, in these cases for Main Programming Language Used versus Inheritance and Experience versus Inheritance.

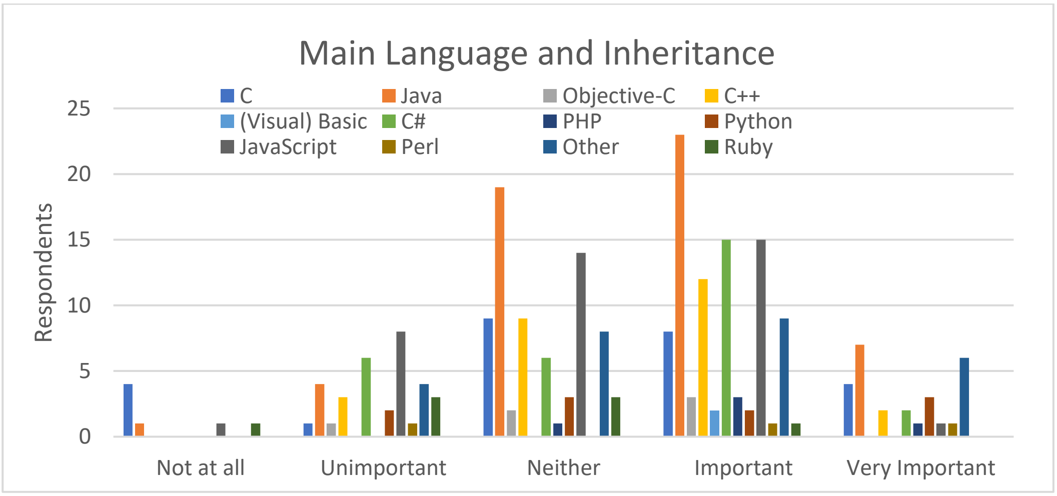


Figure 16 - Response trend for importance of inheritance and main programming language, % of respondents in each category shown in brackets

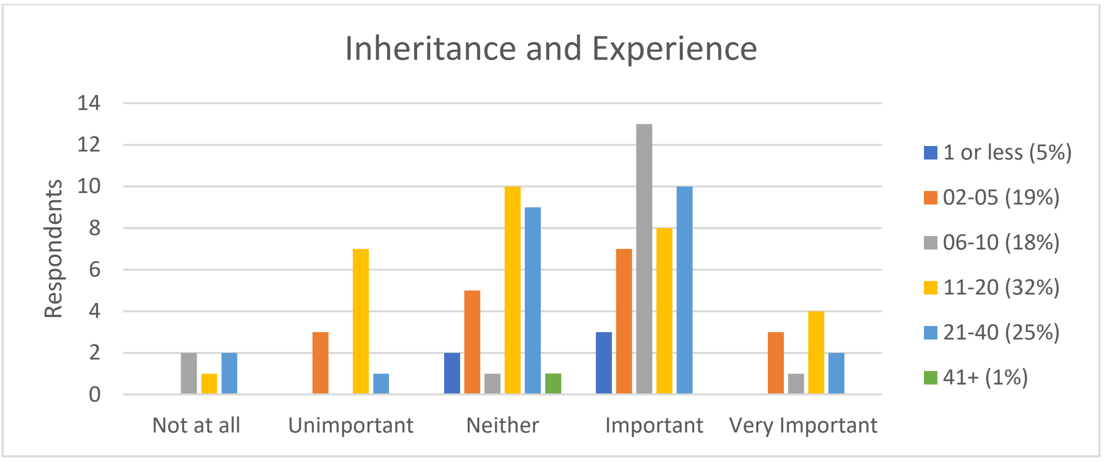


Figure 17 - Response Trends for Importance of Inheritance and Experience, % of respondents in each category shown in brackets

Both graphs show the same general trend as the overall Inheritance result shown in Figure 13. All the sub-categories which have more than a few respondents in them peak at *Important*, with *Neither Important or Unimportant* the second most popular option. Almost all the other graphs followed a similar breakdown pattern, with each major sub-category following the same trend as the corresponding overall result for the question. This suggests a broadly similar view amongst respondents regardless of experience, main programming language used or development role.

There were three cases where some difference in response based on sub-category could be seen. Figure 18 shows a plot of Experience versus Confidence in Design Decision making (see also Figure 8). It is interesting to compare those with 21 to 40 years of experience (blue) against those with 2 to 20 (orange, grey, and yellow) years. (The number of respondents with less than 1 year and over 40 is very small.) 13 (50%) of the respondents with more than 20 years of experience were *Somewhat Confident* or less in their design decisions while the other 13 (50%) were *Confident*. On the other hand, for those with 20 years or less experience, 45 (63%) were *Somewhat Confident* or less and 26 (37%) were *Confident*. So, perhaps understandably, there is an indication that those with much more experience tend to be more confident in their design decisions.

Figure 19 shows a similar plot to Figure 18 when comparing Experience against the Usefulness of Technical Lead / Expert Review in ensuring software design quality. The trend towards *Extremely Useful* can be seen in those with 21-40 years of experience (blue), whereas the peak is *Very Useful* for those with less experience. Again, this finding is not too surprising - those respondents with the most experience, who might have technical lead responsibilities themselves, were more inclined to see the value of expert reviews in assuring software design quality.

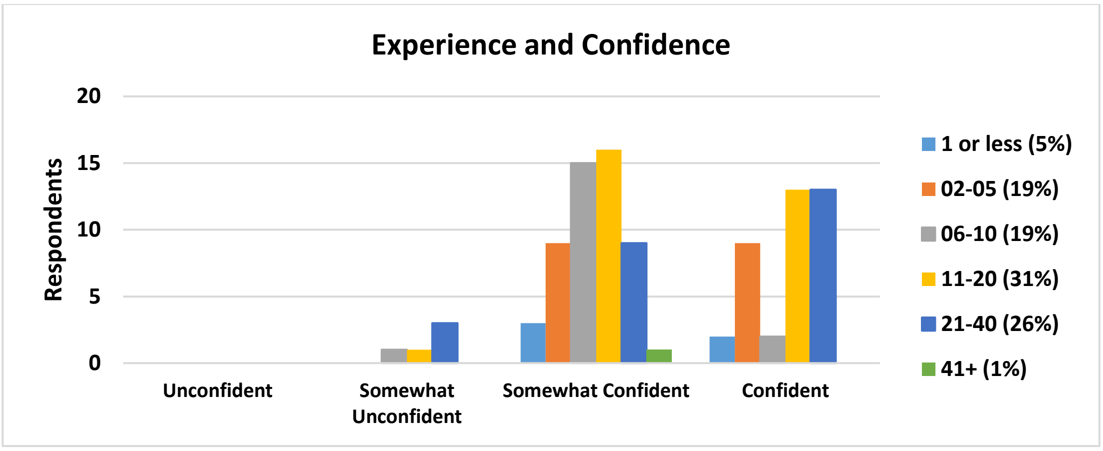


Figure 18 -Experience versus Confidence in Design Decision Making
% of respondents in each category shown in brackets. (See also Figure 8)

The third plot that showed some variation in response was Experience versus the Importance of Design Quality – see Figure 20. Again, the variation is based on those with the most experience in the 21-40 years' experience category. Overall, respondents suggest that design quality was *Important* rather than *Very Important* (in contrast to Functional Correctness which is consistently *Very Important* – see Figure 9), the experienced respondents show more tendency to view design quality also as *Very Important* (blue). This would suggest that those respondents with most experience are more inclined to value design quality.

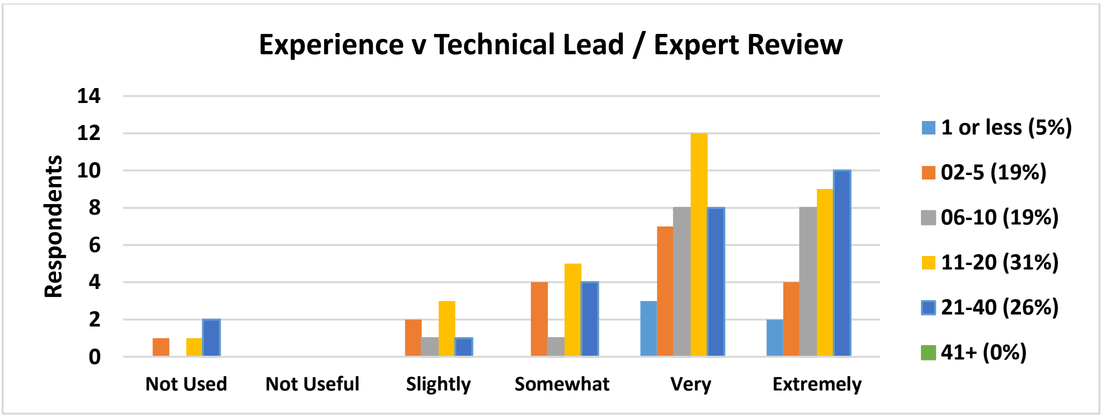


Figure 19 - Experience versus role of Technical Lead in quality assurance
% of respondents in each category shown in brackets

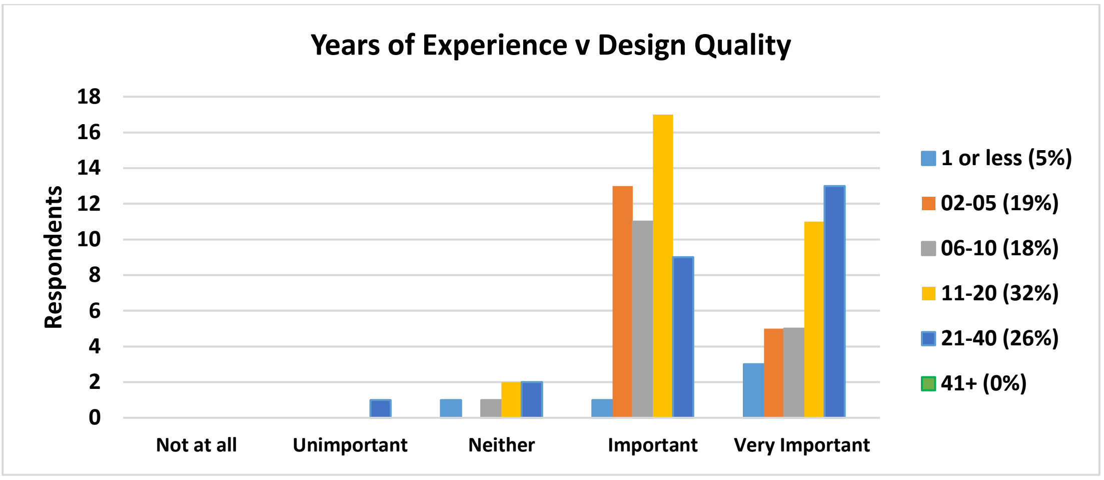


Figure 20 - Experience versus the Importance of Design Quality
% of respondents in each category shown in brackets

Overall, these findings show that there was a consistency in response for the design topics explored, regardless of years of experience, main programming language used and development role. This consistency across all question topics may help strengthen confidence in the general findings. Only 3 out of the 51 graphs showed any distinctive variation (Figure 18, Figure 19 and Figure 20). The patterns observed in those three figures suggest that experienced respondents may have more confidence in their design decisions, place more value on the role of technical leads in reviewing design quality, and place more importance on the role of design quality in the development process.

Demographic data was collected primarily to verify that respondents were in the target population. However, the population of respondents can be broken along broad lines. Figure 21 and Figure 22 show the responses for Q13 (importance of inheritance) with Q34 (programming language familiarity).

For *familiar* languages (Figure 21), respondents give a broadly similar response regarding the importance of inheritance. When the same breakdown is done for *main development* language in Figure 22, the proportions change indicating that familiarity with a language may have some relationship with the perceived value of design guidelines.

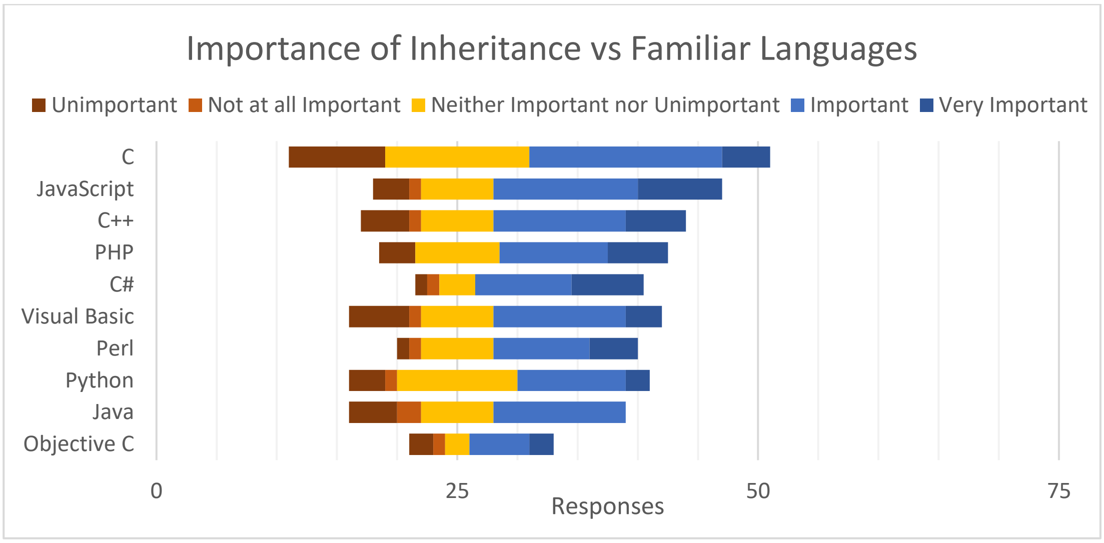


Figure 21 – Importance of inheritance in design work, against programming languages respondents rated as 'familiar with'

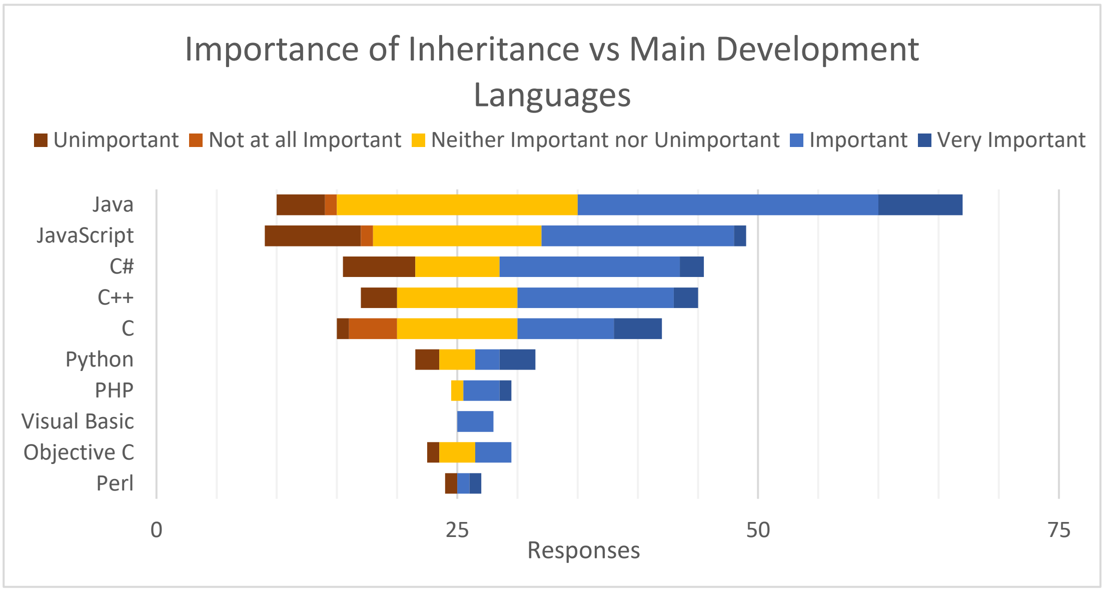


Figure 22 – Importance of inheritance in design work, against programming languages respondents rated as 'main language'

Experience Summary: [KF 3.14] Findings did not appear to be influenced by experience, programming language or development role, though there was a suggestion that those with most experience are more confident in their design decisions, place more value on reviews by team leads and are more likely to rate design quality as very important. [KF 3.15] Degree of familiarity with a language appears to affect the perceived importance of guidelines. It is not clear if a practitioners' main languages shape their overall view, or if perceived guideline importance varies on a per-language basis.

3.5 Answers to Research Questions

3.5.1 Evaluation of Results and Implications

This section attempts to answer each of the research questions in turn. Some answers to research questions follow directly from the survey questions. For the rest, the Key Findings [KF] are the main source of answers to the research questions. Section 3.3 lists the original research questions and the survey questions that were designed with the intention of providing responses that could help answer these research questions.

The first research question (RQ3.1) was: *To what extent do practitioners concern themselves with design quality?* This question is mainly addressed by answers to the survey questions that explicitly explored functional correctness and design quality (Q6 and Q7). It was clear that that practitioner priority is almost always functional correctness over design quality [KF 3.5]. However, most respondents said that design quality was still important or very important in their work. As a rough estimate, participants spend around twice as much effort on functional correctness than design quality. Commercial and other time pressures impact on the time spent on design quality.

The second research question (RQ3.2) was: *How do practitioners recognise software design quality?* This was mainly addressed by answers to survey questions that explored possible approaches to quality (Q1, Q5), the follow-up free text questions (Q2) and questions which explored specific design practices (most of the questions Q10 to Q25). The key findings identified a range of mechanisms that practitioners use to determine software design quality: personal experience [KF 3.1], peer and manager review [KF 3.1], design practices: naming practices, *simplicity* – keep things simple, modularity, documentation, full set of tests [KF 3.3], coupling [KF 3.8], program to an interface [KF 3.6], class and method size [KF 3.10], cohesion [KF 3.9], class and method complexity [KF 3.11]. Respondents also identified the role that tools (e.g. SonarQube), some metrics and standard design guidelines can play in determining design quality.

Overall, it is concluded that designers use a combination of experience, peer review, tools, design guidelines and some metrics to recognise design quality. There appears to be a core of important design guidelines based on small size, minimal complexity, low coupling, good naming, high cohesion, ease of testing etc. It should be acknowledged, however, that a high-level questionnaire such as the one used here is only identifying indicators used by practitioners, indicators which *may* suggest a design problem. The survey has not revealed exactly how they are linked to design quality or the extent to which there is any explicit relationship. It seems that experience prevents the blind application of guidelines, helps deal with new problem domains and technologies, helps to address changing and incomplete specifications, and in recognising essential complexity where it arises.

The third research question (RQ3.3) was: *What design guidelines do practitioners follow?* This was mainly answered by survey questions which explicitly concerned guidelines (Q10 through to Q27). There was clear, strong support for many of the long-standing design guidelines amongst the respondents. 87% of respondents said that coupling was an important or very important design guideline [KF 3.8]. 83% stated that programming to an interface was at least important [KF 3.6]. 70% said that cohesion was important. Over 70% said that both class and method size were at least important [KF 3.10]. Over 70% also said that both class and method complexity were at least important [KF 3.11]. There were more mixed views on inheritance [KF 3.7] and design patterns [KF 3.12], though both were recognised as potentially important when they are used appropriately. There are indications that the perceived relevance of specific guidance changes with programming language familiarity [KF 3.15].

The fourth and final research question (RQ3.4) was: *What information do practitioners use to make design decisions?* To answer this research question, responses to a range of survey questions were used, often from the free-text responses. As highlighted under RQ2 above, it seems clear that respondents believe that experience is a key factor in decision making – designers recognising situations similar to those they have encountered before and reusing design solutions [KF 3.1]. As such, unfamiliar problems can cause difficulty in making good design choices. Beyond experience, it is clear that practitioners do make use of the main, established design guidelines (see RQ3) and other well-established design practices such as Clean Code and SOLID. There was evidence of the use of tools (e.g. SonarQube) and some metrics (e.g. LCOM for complexity) in supporting decision making. An interesting finding was the regular mention of the close link between testing and design, higher quality designs tend to be easier to test, and difficulty in writing test cases and achieving test coverage can indicate poor design quality.

3.5.2 Lessons Learnt

The final question (Q37) asked whether the respondents had any comments on the questionnaire itself. The actual feedback here was quite sparse, with only a few issues raised.

- One respondent questioned the focus on object-oriented technology.
- One respondent pointed out that *agile* and *iterative* were not mutually exclusive development methodologies (Q8). (The questionnaire text did define what was intended by these two terms.)
- One respondent commented on the lack of questions on testing. As the results of this survey have shown, design quality and testing are closely related. It may have been better to explore this relationship further.
- The questionnaire presented design quality and functional correctness as separate topics. One respondent commented that these should not be viewed separately.

Overall, respondents had very few issues with the wording of the questionnaire, the quality and variety of responses suggest that most had a clear understanding of what was being asked.

If the questionnaire was to be used again there are a few ways it might be improved. Some of those would be to address the concerns raised in the previous section – being clearer regarding *agile* and *iterative* and perhaps introducing a few questions on the role of testing. Removing the focus on object-oriented technology would completely change the purpose of the questionnaire. The issue of separate consideration of design quality and functional correctness may require some change of wording (though this issue was only raised by one respondent).

With hindsight, the main omission was some exploration of thoughts from respondents on where researchers could mostly usefully contribute to future improvements in software design quality.

3.6 Discussion

A major contribution from this research is confirmation of the importance of personal and peer experience to practitioners, the experience that comes from seeing similar problems and knowing what works in those situations. Practitioner experience has long been recognised as important in software development, going back to the days of chief programmer teams (Mills 1971) and the emphasis placed on retaining experience in Lister and DeMarco's Peopleware (Lister and DeMarco 1987). These findings confirm that the practitioners surveyed here see experience as the most important contribution to design quality. Figure 7, based on Question 7, emphasises the relative importance of experience and peer review compared to guidelines tools and metrics (which were still seen as useful). In discussing Lister and DeMarco's Peopleware, Boehm and Turner say: "*The agilists have it right in valuing individuals and interactions over processes and tools … Good people and teams trump other factors*" (Boehm and Turner 2003). In their recent work on design smells and technical debt, Suryanarayana et al. argue that developers lacking in experience are often responsible for poor quality designs (Suryanarayana et al. 2014). Simons et al. also highlighted the need involve humans within the software design quality assessment process, metrics alone were insufficient (Simons et al. 2015). Wu et al. have also recently found a strong link between developer quality and software quality (Wu et al. 2014).

How can inexperienced designers and novices gain such experience? One answer is learning *on the job* especially from experienced peers (Begel and Simon 2008). Can education and training do more? A solution might be to ensure that students are exposed to many examples of design, especially high-quality design. A useful contribution would therefore be the identification of a range of widely-recognised, high-quality software design case studies to be used in education and training. JHotdraw is often presented as one such example (Ferreira et al. 2012). Ducasse also argues that *"… patterns (and smells) are really effective ways to capture such [design] experience*" in her introduction to a refactoring manual by Suryanarayana et al. (Suryanarayana et al. 2014). Design patterns may therefore have a role to play, being small examples of high quality software design – though previous work (Zhang and Budgen 2012) and the findings from the current survey suggest mixed views on the value of patterns as a learning mechanism.

Respondents said that functional correctness was more important than design quality, but that design quality was still important. There was an indication that practitioners spend about twice as much effort on functional correctness as design quality. Commercial pressures reduce the time afforded to design quality and for refactoring designs. In Yamashita and Moonen's survey of practitioners exploring their knowledge and use of code smells (A. F. Yamashita and L. Moonen 2013), respondents highlighted that they often had to make trade-offs between code quality and delivering a product on time. Again, Demarco and Lister recognised the commercial pressure of time to market (Lister and DeMarco 1987). The implications of this finding suggest the need for approaches to design quality that fit efficiently and effectively into the development process – this is one reason why experience is so important, but also it also suggests the need for efficient and effective tools and metrics.

It is clear that in this study that respondents viewed many of the long-standing design guidelines as helpful to identify and improve design quality. A large majority of respondents said that coupling was an *important* or *very important* indicator of design quality, similarly for programming to an interface, cohesion, class and method size, and complexity. The same key design factors were identified as important motivators for refactoring designs. The importance of inheritance and design patterns were more contentious. That respondents see coupling, complexity and size measures as important, and inheritance less so, is in keeping with the findings of Jabangwe et al.'s major systematic literature review (Jabangwe et al. 2015). These findings are also consistent with Al Dallal's finding that most of the cohesion, size and coupling metrics could predict class maintainability (Al Dallal 2013) and also Yamashita and Moonen's finding that the code smells most closely associated with maintenance problems were associated with size, complexity and coupling (A. Yamashita and L. Moonen 2013).

Tools were identified as useful, particularly for checking coupling, cohesion and complexity (with SonarQube receiving many mentions). The results suggest that there is a role for the right tools and that some metrics are useful, at least as indicators of minimum quality or to flag-up areas requiring careful attention. There is therefore a need to identify what tools and metrics are particularly useful, what aspects of these are helpful, and to what extent they can be improved. There is an ongoing research effort to convert general guidance on design guidelines into theoretically sound and empirically demonstrated metrics (Kitchenham 2010). It would appear, though, that despite theoretically weaknesses and inconsistences, metrics are used in practice - perhaps only as indicators, with additional expert judgement being used as the final arbiter.

As an example of the importance of expert judgement being used to temper the blind application of metrics, there was a recurring theme of careful management of essential complexity from the problem domain. Respondents suggested that blocks of essential complexity should be kept together, and understood as a whole rather, than thoughtlessly following design rules that would encourage breaking them down into less understandable, less cohesive, smaller units.

An unanticipated but very important contribution of this study was the repeated emphasis that respondents put on Clean Code and the related SOLID principles. Respondents consistently identified a core set of these practices as positively contributing to software design quality. These included *good* naming practices, *small* size of methods and classes, simplicity (or reduced complexity), modularity and documentation, single responsibility, interface segregation, and Liskov substitution. This finding raises an important future research question: how much design quality can be achieved just by carefully following a set of such principles (perhaps along with the previous higher-level guidelines and some tool and metrics support). This relates to Becks view that *"... a certain amount of flexibility just comes from doing a clean job, doing a workman-like job, acting like an engineer..."* (K. Beck 2014).

Another unanticipated and important contribution was the recurring theme of a potential relationship between ease of testing and design quality. The claim is that well-designed software systems are easier to test and, importantly, vice-versa, that using practices such as Test Driven Development (K. Beck 2003) can encourage better quality designs. Respondents noted that reduced coupling, increased cohesion and reduced complexity all made testing easier. This appears to be consistent with Bajeh's recent work suggesting a correlation between coupling and cohesion metrics and *testability* (Bajeh et al. 2015). Again, some of this finding may be related to experience, knowing how to construct a design to make it easier to build test cases and achieve coverage. There would appear to be further valuable research to be done exploring the relationship between testing practices and software design quality. Some of this experience may reside in processes and organisations rather than with individuals.

There was also some identification of practices that can have a negative impact on design quality: overuse of interfaces, inheritance not modelling the problem domain, inheritance not adhering to *is-a* relationships (Liskov substitution), *non-shallow* hierarchies and overuse of design patterns, particularly by novices. There were also negative comments about "*striving for cohesion for its own sake*" and warnings against excessive decomposition that *"discards essential complexity*". Again, this may be where experience is used to make the final decision rather than the unquestioning application of design guidelines. It is interesting that many of these issues are related to the application of interfaces and particularly inheritance. Chapter 4 investigates the relationship between interface practices and design quality – addressing the question: To what extent good interface practices can be distinguished from bad?

Finally, these research findings did not appear to be influenced by respondent experience, programming language used or development role, though there was a suggestion that those with most experience are more confident in their design decisions, place more value on reviews by team leads and are more likely to rate design quality as very important.

3.6.1 Limitations

One major limitation of the survey is the focus on object-oriented technologies. While many modern languages are object-oriented (or incorporate OO elements), subtle differences in language implementation may mean that guidance does not travel between languages. However, the design principles underpinning OO are strongly tied to the guidance for using this technology (e.g. separation of concerns, polymorphism), which are the focus of this survey. So, it might be argued that, while the details differ, wherever there is OO design, the concerns addressed by the questions in the survey remain relevant.

A further limitation is the representativeness of the respondents, which may prevent incorporation of the presented findings into the larger picture of the software practitioners. Accepting validity concerns which have been addressed in section 3.2.4.6 (above), the demographic supplied by respondents indicates diversity in location, experience, technology, problem domain, and organisational role. The respondents appear to fall within the target population – however the diversity within the respondents means that the sub-population within the respondents is too small to carry out further analysis based on most of the demographic distinctions.

3.7 Conclusions

Based on quantitative and qualitative analysis, drawing together key findings, and trying to answer the original research questions the following conclusions emerge:

1. Design quality is important to industrial practitioners, but not as important as functional correctness.
2. Commercial and other time pressures limit the time spent on design quality and associated activities such as refactoring.
3. Practitioner experience is the key contributor to design quality, both working as an individual and as part of the peer review process.
4. Novel problems and changing or imprecise requirements make design more difficult.
5. In the absence of specific knowledge of a programming language or problem domain, practitioners *fall back* to depend on general guidance and discipline.
6. It is important to recognise essential complexity in a problem and be mindful of how it is distributed in design.
7. Many of the traditional design guidelines are viewed as important to design quality: coupling, cohesion, complexity, size, and programming to an interface. Inheritance and design patterns are also important but controversial and susceptible to misuse.
8. There are many lower level practices identified as contributing to good quality design e.g. SRP, LSP, OCP, small size, simplicity and good naming practices.
9. Tool support such as SonarQube is often used to support software design quality; there was also some evidence to support the use of metrics.
10. There is a close relationship between testing practices and design quality. Good testing methodology can help improve design, testable designs are thought to be better, and poor designs can be harder to test.
11. The findings did not appear to be influenced by experience, programming language or development role, though there was a suggestion that those with most experience are more confident in their design decisions, place more value on reviews by team leads and are more likely to rate design quality as very important.

The contribution of this research is that the practitioners who were surveyed saw experience and peer review as the most important contribution to software quality. These practitioners do generally find design guidance (coupling, cohesion, complexity and size), metrics and tools useful but not as important as experience. The practitioners see design quality as important but not as important as functional correctness. More surprising, perhaps, was the importance of Clean Code / SOLID principles to practice - as important as traditional guidelines and metrics - and the potential relationship between testability (ease of testing) and design quality.

3.7.1 Impact

There are implications from these findings for practitioners, researchers and educators. For practitioners, the findings confirm the importance of experience and peer review. As has been long recognised in the literature, it is vital that organisations recognise, reward and retain their experienced developers. However, it is also important for industry to recognise the need to support less experienced staff to and to support their development. Practitioners may also find some of the specific findings reported here on practices and tools useful in their own working environment.

For researchers, more work could be done investigating the role of experience in the design process, to what extent the nebulous notion of experience can be translated into better guidelines, metrics and tools. The survey has only identified the high-level practices that those working in industry find to be useful indicators of quality. Much more work needs to be done with industry to discover exactly how these indicators are used in practice and the extent to which an explicit link can be demonstrated between them and design quality.

There are also implications for the educators of students. Given that experience is so important in design, are there mechanisms that can accelerate students' acquisition of appropriate experience? One possibility is through the study of high-quality examples of industrial-strength design and architecture case studies to do so, perhaps supported by experienced practitioners. Another is to ensure that students understand the fundamental principles underpinning design guidelines such as coupling, cohesion, complexity, size, and Clean Code practices. At the moment, many students are taught the basics of key techniques such as inheritance, programming to an interface and design patterns – these findings suggest that experienced practitioners know more than this – they appear to know when and how to best apply these techniques.

4 Profiles of Interface Use

The survey responses discussed in Chapter 3 highlighted that design quality is important to practitioners, and that guidance to ‘program to an interface’ is popular and considered useful. Additionally, interfaces may help to address other concerns identified by the survey such as coupling, modularity, and testability. These findings motivate further investigation of interface use in object-oriented systems, which is the focus of this chapter.

Interfaces allow the definition of types which are not constrained to a single implementation. ‘Program to an interface’ is a key guideline from the influential design manual, *Design Patterns* (Gamma et al., 1994).

4.1 Motivation

The goal of this research study was to investigate how large-scale, ‘real-world’ Java programs use design guidelines. Specifically, it was to contribute to the understanding of how developers use interfaces in object-oriented (OO) systems with respect to guidelines such as ‘program to an interface’ which “*greatly reduces implementation dependencies*” (Gamma et al., 1994). As Abdeen et al. note, “*In the literature, few recent works attempt to address the particularities of interfaces.*” (Abdeen and Shata, 2012).

Note that this is an observational study – no attempt is made to infer practitioner motivation or provide a high-level model.

4.2 Problem Statement

Interfaces are used in many stages of design, in architecture, specification, design patterns, and implementation details. Studies counting interfaces confirm that they are present in systems in many forms, some of which appear to defy guidance.

A previous attempt was made to classify interfaces by arguments from design which illustrated that definitive and robust classification rules cannot be formulated in a top-down matter (Steimann and Mayer, 2005). This study, by contrast, takes a bottom-up approach - looking for patterns in interfaces and their use in practice.

Insights into the use of interfaces would be informative to both education and practice. An understanding of how interfaces are used in industry would allow the creation of realistic examples and tasks to ground the more abstract guidelines. Additionally, a better understanding of use in different scenarios may simplify making, and later reviewing, design decisions.

While there are some unconventional uses of interfaces – marker or constant interfaces – these are easily detected due to their unusual structure and would be difficult to confuse with more conventional interface use. This does raise the question if there are other unconventional uses that *are* difficult to distinguish from the general interface population.

4.3 Research Questions

While interfaces feature in previous work, interfaces are often excluded as their lack of source code can confound metrics. This study focuses specifically on interfaces and their properties as the objects of study. The following research questions are explored:

RQ4.1: How are interfaces defined in the systems?
RQ4.2: What is the profile of interface size across the systems?
RQ4.3: To what extent are interfaces implemented in the systems?
RQ4.4: How are interfaces used in the systems?

4.4 Research Objectives

This study investigates how much information about interface use can be assembled from examining source code:

- Gain insight into how interfaces are used both within and between systems, and determine if there is a useful 'baseline' for interface properties.
- Determine if there are any *natural* categories of interfaces.
- Determine how observed use compares to what would be expected if guidance, such as *program to an interface*, were followed.

4.5 Context

This research contributes to the goal of understanding the practices used by professional developers. As Baker et al. say in their foreword to an IEEE Software special issue on Studying Professional Software Design - "*Although we have quite a few high-level design methodologies we don't have a sufficient understanding of what effective professional developers do when they design … To produce insights that are relevant to practice, researchers need to relate to practice and be informed by it.*" (Baker et al., 2012)

4.6 Methodologies

Examining source code corpora is a common strategy in empirical software engineering, to the extent that there are large repositories of curated code available for *data mining* experiments. Several other large-scale studies have been conducted on similar corpora, to answer the question of 'what software looks like'.

This study is more detail oriented, with some aspects requiring manual inspection, so fewer systems were selected. In addition, because byte-code analysis may lose some of the details of how source code is written, analysis was based on source-code, which presents different challenges.

There are various frameworks for the analysis of source- and byte-code, such as MOOSE[9] and Soot[10]. These are adapted for experiments as required, and it was not clear that the available tools could support the requirements for this study. Consequently, this project uses a code analysis tool based on the Java source code parser from a popular Integrated Development Environment (IDE) called Eclipse.

The Eclipse JDT Core is a fully featured Java source code parser which is compatible with both the latest, and previous versions of Java. Parsed Java code is converted to an Abstract Syntax Tree (AST) which is accessed by defining appropriate visitor classes.

4.7 Experimental Design

This section explains the methodological and design choices for this empirical study.

4.7.1 Subjects

This study was based on eleven open source systems. The main criterion in choosing systems was that they be representative of 'real world' Java applications. The Qualitas Corpus[11] (QC) provides a curated collection of Java applications intended for the purposes of carrying out "*reproducible studies of software*". To ensure that the examined systems were mature, six systems were chosen from the evolution distribution of the QC (release 20130901) – where included systems have at least ten versions.

[9] moosetechnology.org
[10] sable.github.io/soot/
[11] qualitascorpus.com

Since this is not a replication study, and to avoid overfitting to a very popular research corpus, a wider search for candidate systems was performed beyond the popular research corpus, looking for Java applications with at least ten release versions and some indication that the system was in active use. From this search, the following systems were also selected: btrace, easysim, jedit, lobo. Finally, jhotdraw (JHD) was added to the collection as it has been regularly studied as an exemplar of good design practices (Ferreira et al., 2012).

For this study, the latest versions at the time were used – as shown in Table 18. For each system in the study, the application name and version number are listed. Additionally, column 4 indicates how many types (class, abstract class, interface) definitions are in the source code for that application. Note that some systems use inner classes and/or interfaces, and enumerated types to varying degrees, these are included in the counts shown to provide an unfiltered notion of system size. This count does not include references in the source code to third party types which are defined outside the analysed source code.

4.7.2 Objects

The objects of the study were interfaces in open source systems written in the Java language. While the interfaces were segregated based on origin-system for some analysis, the interfaces are generally treated as a single population.

Table 18 – Interface Study Corpus Overview

Application	**Version**	**Domain**	**Types Defined in Local Source Code**	**Source**
argouml	0.34	UML modelling tool	2297	Qualitas Corpus
azureus	4.8.1.2	Bit torrent client	3971	Qualitas Corpus
btrace	1.2.5.1	Dynamic tracing for Java	261	SourceForge
easysim	4.3.6	Interactive simulation platform	1000	University of Murcia
freecol	0.10.7	Turn based strategy game	978	Qualitas Corpus
freemind	0.9.0	Mind mapping tool	763	Qualitas Corpus
jedit	5.1.0	Programmers text editor	1033	SourceForge
JHotDraw (JHD)	7.6	GUI framework	748	SourceForge
jmeter	2.9	Load-testing application	1123	Qualitas Corpus
lobo	0.98.4	Web browser	903	Sourceforge
weka	3.7.9	Machine learning	1647	Qualitas Corpus

4.7.3 Instrumentation

In initial versions of this study, which only focused on *direct* implementation of interfaces i.e. a class had an implements statement in its header, the javaparser[12] library was used as the basis of the analysis tool. It was then realised that it was also important to take *indirect* implementation into account i.e. many classes were implementing interfaces indirectly via inheritance, and so a more sophisticated tool was developed using the Eclipse JDT Core[13] library. In switching analysis tools confidence was gained from the fact that both tools provided similar results. Another issue addressed by the revised tool was that some systems had a regular pattern of interface implementation via anonymous inner classes – omission of these from the initial analysis tool suggested that many interfaces had wrongly appeared 'unimplemented'. Although, as noted in other studies, the effect of inner classes is very small (Systa and Muller, 2000).

4.7.4 Data Collection Procedures

Source code from the Qualitas Corpus is provided in a standard archive format, including multiple versions of each application's source code where available. The first step of data collection was to unpack the Qualitas corpus using the supplied scripts, and collect the non-Qualitas systems into the same format for ease of comparison.

Analysis of source code occurred in two stages – initial extraction of data from source code is a relatively expensive operation as type-references must be resolved across the entire code-base for each application. This step was carried out using a tool based on the Eclipse JDT Core, which can parse and analyse source code, this took longer for the larger systems, as resolving types can become more difficult as a system increases in size and complexity.

The initial extraction was performed on laboratory PC with the larger systems requiring a reasonable amount of memory (4x 3.4Ghz, 16GB RAM), analysis of all systems was completed in a few hours. The next stage of analysis used the extracted data to perform further analysis, which was quicker, with the raw data acting as a cache. Using data extracted from the source code in this way traded memory for hard-drive space, which matched the available computing resources. The second stage of analysis took much less

[12] javaparser.github.io/javaparser
[13] org.eclipse.jdt.core

time than the extraction of the initial raw data and could be performed on a normal desktop machine (2x 3.16Ghz, 6GB RAM).

4.7.5 Analysis Procedure

Data were collected over several passes of the resolved AST for each program for different elements of the program structures. Sample information collected is shown in Table 19.

Table 19 : Definition for collected data

Direct	System Profile	Detail	Example
D1	System name	Name	argouml
D2	Types defined in source code	Count	2297
D3	Types that are interfaces	Count	192
-	Interface uses as:	-	-
8,13,14	Class attribute	Count	233
8,13,14	Method parameter	Count	368
8,13,14	Constructor parameter	Count	75
8,13,14	Local variable	Count	488
8,13,14	Return type	Count	336
	Entity Profile	**Detail**	**Example**
D4	Name	Full name and path	`org.argouml.uml.ui.TabTaggedValuesModel`
D5	Entity type	Class, Interface, etc.	`CLASS`
6	Implements count	Count of interfaces directly implemented	`3`
7	Extends count	Count of types directly extended	`1`
D6	Implemented names	Names of interfaces directly implemented	`(java.beans.VetoableChangeListener, org.argouml.kernel.DelayedVChangeListener, java.beans.PropertyChangeListener)`
D7	Extended names	Names of types directly extended	`(javax.swing.table.AbstractTableModel)`
-	**Dependency Relationship: B depends on A**	**Detail**	`Example`
D8	Type A nature	Class, Interface, etc.	`Interface`
D9	Type A origin	User Defined, Java, Third Party	`User Defined`
D10	Type A name & path	Location and Type name	`org.argouml.cognitive.Poster`
D11	Type B nature	Class, Interface, etc.	`Class`
D12	Type B name & path	Location and Type name	`org.argouml.cognitive.ToDoItem`
D13	Scope in Type B where dependency occurs	Scope: Class, Method, Constructor	`CLASS`
D14	Type A used as	Field, Parameter, Return	`FIELD`
-	**Interface Use Profile**	**Detail**	`Example`
D15	Origin system	Name	`argouml`
D6	Name and path	Full name and path	`org.argouml.kernel.Project`
D17	Method count	Count	`70`
D18	Method argument count	Count	`41`
17,18	Average method arguments	Method arguments / count	`0.44`
-	Dependency on:	-	
D19	Primitives	Count	`17`
D20	Self	Count	`0`
D21	Java types	Count	`57`
D22	Third-party types	Count	`0`

-	Dependency By:	-	-
10,11	Concrete class	Count	232
10,11	Abstract class	Count	36
10,11	Interface	Count	7
6	Direct implementer count	Count of types directly implementing this interface	1
5, 6	Direct implementers	Names of types directly implementing this interface	`(org.argouml.kernel.ProjectImpl)`
5,6,7	Indirect implementer count	Count of types indirectly implementing this interface	`0`
5,6,7	Indirect implementers	Names of types indirectly implementing this interface	`()`
5,6,7	Total Implementer count	Count	1
-	Uses as:	-	-
D23	Concrete class constructor parameter	Count	17
D24	Abstract class constructor parameter	Count	1
D25	Concrete class local variable	Count	15
D26	Abstract class local variable	Count	3
D27	Concrete class method parameter	Count	71
D28	Abstract class method parameter	Count	14
D29	Interface method parameter	Count	5
D30	Concrete class method return	Count	20
D31	Abstract class method return	Count	6
D32	Interface method return	Count	2
D33	Class method variable	Count	120
D34	Abstract class method variable	Count	0
23-34	Total use as variable	Count	274
-	**Inheritance Hierarchy Properties**	**Detail**	`Example`
D35	Root name	Full name and path of the root type	`org.argouml.cognitive.CompositeCM`
D36	Root implements interface	Root type implements at least one interface	`true`
D37	Root origin	Local, java, third party	`Local`
4,7	Type count	Types in hierarchy (including root)	`4`
D38	Abstract type count	Abstract classes in hierarchy (including root)	`1`
D39	Direct implementations	Use of implements at each level	`[1 0 0]`
7, 39	Indirect implementations	Indirect implementations at each level	`[0 2 1]`

Basic information about all types defined in the source code (class, abstract class, interfaces, including 'inner' structures) was recorded, with more detailed information being collected on interfaces and their use. In the *Direct* column, data derived directly from the source code is indicated with a ‘D’ prefix. Information inferred from extracted data has no D prefix and instead lists the D-value(s) used to infer the information.

For example, D2: *Types defined in source code* is a direct count, while *Interface Used As : Class Attribute* (row 5) is a derived value, using D8, D11, and D14.The *Interface Used As* indicate counts of how often interface types were used in each kind of variable.

The initial stage of data collection collected ‘raw’ data such as registering the existence of all the types in the corpus, creating detailed records of interfaces present including such data as method counts, types and number of method parameters, each appearance of interface types in the source code in type definitions and signatures, including the scope of the use such as local method variables (as noted above, this is labelled with a ‘D’). This stage also gathered information about implementation of interfaces. The second stage of data processing used the source code and the initial information to infer more specific data such as overall implementation counts and the distribution of interfaces in inheritance hierarchies. Additional information about specific entries is included in Table 19 in the ‘Detail’ column.The *dependency relationship* block shows a single recorded dependency. This data indicates the nature (class, abstract class, interface, enumerated type) of the types in the dependency relationship. In addition, the scope (class, constructor, method) and use as (field, parameter, return) can be combined to identify most uses of types. Some examples are constructor parameter, method return, class field (an attribute) vs method field (a local variable).

Details shown in the *Interface Use Profile* section show the details available for each interface in the study. Most items are simple, but revealing, counting metrics. Beginning with *internal* properties - the number of method signatures defined, the number of method parameters across all method signatures defined in the interface. Some of these details were copied over from the more general *entity profiles.* Most of the counts capture how the interface interacts with the rest of the system. The dependencies of each interface are broken down by nature – this is to determine how much interfaces depend on other types (especially less abstract types). The *Dependency by* items along with the implementation counts give some notion of the popularity of an interface. Finally, each use of an interface (as recorded in the dependency relationships) are collated to show where that interface is used in code in terms of the kind of interface-typed references.

The final section of Table 19 shows information collected about interfaces in inheritance hierarchies. The last two rows show an array for each hierarchy with direct and indirect interfaces implementations at each depth. For example [1 0 0] indicate one interface implemented at depth zero, and no other interface implementations in the hierarchy which has a maximum depth of 2. From the last row [0 2 1] indicates that an inheritance hierarchy has three indirect interface implementations in total, none at the root, two at depth one, and one at depth two. This is like the BIT vector proposed by Harrison et al. for presenting breadth of inheritance hierarchies (Harrison et al., 2000), which is discussed in more detail in section 5.5.3.2.

4.7.6 Evaluation of Validity

This section addresses validity concerns for this study. Overall, this study faces similar challenges to previous corpus studies.

4.7.6.1 Construct Validity

There are many dimensions of software which could be considered when searching for patterns of use. In this case, the most obvious and manually recognisable features were chosen as these are the features that practitioners have immediate access to. As such, the outliers and modes of use identified will be recognisable in practice. The outliers and *normal* use identified in this study have been discussed in terms of recognised modes of interface use from the literature and design guidance where clear mappings exist. However, there is no attempt to infer motivation or provide a higher level, complete model.

A further source of validity concern in software analysis studies is the tools used to extract the data. In this case, the tools used were based on a core component of a very popular IDE – ensuring that the core functions are well tested. The novel parts of the tool were tested during development and validated against real data. Additionally, the project has involved extensive manual inspection of source code, which has further confirmed that the tools used have correctly identified structures and counts within the code corpus.

4.7.6.2 Internal Validity

Internal validity is concerned with alternative causes. Each design choice and modification is the result of deliberate human intervention. So, it could be argued that software designs are the way they are due to an accumulation of undocumented practitioner decisions – which are inaccessible to the researcher. Though it is clear from the literature and guidance that these reasons may also be obscure to practitioners who rely on intuition, habit, and incomplete models of the systems they work in.

Looking at design *in the small* or at specific categories of outliers, the primary motivation for the kind of outlier observed may be identified. For example, the *listener* archetype is a design need in event driven systems, which is observed to manifest as a common structure (small highly implemented interfaces, with 'listener' in the name). However, this mapping between design need and structure is not always reliable.

This is only a threat to internal validity if there is an insistence that the mapping between design need and structure *are* always reliable. However, this research looks at identifying common uses – which does not require them to be complete. Practitioners are accustomed to using design heuristics – and would likely be suspicious of 'hard and fast' rules if they were offered.

Alternate causes for *design in the large* or system level results appear to be difficult to separate from other factors such as practitioner style, architecture, problems domain, and the constraints placed on a design by the reuse of external libraries and frameworks. Consequently, no attempt has been made here to make useful predictions about system level characteristics.

4.7.6.3 External Validity

A general concern when experimenting with software is that it is not representative of the wider software *ecosystem*. There is a threat to validity arising from the small number of systems analysed and their properties. There was no consideration of how the developers, their individual practices, and their organisation contributed to the systems. The variety of findings in this and other studies regarding the differences in problem domain, programming style, programming languages, architecture – indicates that it is unlikely that *software system* is the correct unit of comparison for this type of research.

The external population of *interfaces in Java systems* is a reasonable population to consider comparing the findings from this work to. It is reasonable to expect that any such external population must be at least as diverse as the systems examined here. Accepting, as much empirical software research does, that *normal use* is defined by what is observed – differences in opinion must be accounted for. For example, marker or constant interfaces may be avoided by some practitioners, while others may welcome guidance on the use of these structures.

On the other hand, while small, the corpus used in this study represents a varied selection of mature 'real world' applications that have serious application. Additionally, the initial high-level findings indicate that the corpus used in this study resemble larger corpora in terms of cross-system averages. It is therefore possible that these initial findings are representative of what may be found in wider practice, and in indeed some of the key findings are consistent with previous research.

4.8 Results

This section presents the main high-level observations across all eleven analysed systems and then discusses each system in turn.

4.8.1 Descriptive Statistics

Interface use in eleven open source Java systems was examined. Systems ranged in size from 261 to 3971 types defined in local source, and contained 2048 interfaces overall, with 15 to 1108 interfaces per system (4%-28% of defined types). Interface sizes range from 0 to 393 methods, and are used as variables and parameters 14999 times. Interfaces defined in local source code were implemented 13342 times across all systems with individual interfaces implemented 0 to 605 times[14].

4.8.2 Data Set Reduction

Despite the existence of conventions, there is no fixed way to lay out a Java program. As such the organisation of source code, test code, and imported libraries varies between systems. When resolving types in source code, third party libraries may not be available with the code in the Qualitas Corpus (QC) archive format. In these cases, the type of external imports can often be inferred from import statements.

In a small number of cases (< 0.5%), imports are made using Java's asterisk notation which can make it impossible to recover the original path of an imported type without access to the external library. In these cases, a best effort was made to handle the unresolved types

[14] weka.core.RevisionHandler – a one-method interface to access a 'version identifier', has 605 implementers.

consistently – matching on type name and the source library. Note that this has little impact on findings as the relationships with unresolved types are still recorded.

4.8.3 Results

The following definitions and counts were used in this study:

uT = The size of the set of user-defined types in source code. All types defined by the user in the source code, even if they are not referred to by other types or used in program execution. This also includes inner classes, annotations, and enumerated types. Also referred to as 'local' types.

uI = The size of the set of interfaces defined in the source code.

Interface use = When an interface is used to specify a type e.g. as a class attribute, method field (local variable), parameter or return type.

Direct interface implementation: Type A directly implements interface B when type A has a statement in its definition *A implements B.*

Indirect interface implementation: Type A indirectly implements interface B where A does not implement B in its definition, but some super-type of A directly implements interface B.

Local interface: An interface defined in the local source code of an application i.e. not from an application library or imported from a third-party component or framework.

4.8.3.1 Presence of Interfaces

Table 20 (column 5) shows the number of implementations of local interfaces in each system by non-interface types, this includes indirect implementation (inherited implementations), and shows that interface implementation is common across the examined systems. The table also shows the number of user-defined types (uT) in the systems (column 2) and the number of these that are interfaces (uI) (column 4). Column 6 shows the percentage of all types defined in local source code that are interfaces in each system, therefore 8% of the user-defined types in argouml are interfaces. It is unlikely that % interfaces would reach 100% as any system would have a mixture of concrete classes and interfaces, e.g. one interface defined for each non-interface type would result in a % Interfaces of 50%. Column 6 illustrates that most of the analysed systems have a relatively low proportion of interface definitions amongst user-defined types, in the range 4-28% (mean 10%, median 8%) with outliers being lobo (20%) and azureus (28%). Note that in column 5 azureus has many more implementations of local interfaces than shown in column 3 (total interface implementations by non-interface types), this is due to extensive use of anonymous inner classes which implement interfaces, but are not included in the count of user defined types.

On average, the presence of interfaces is in keeping with the previous findings of Tempero et al.: "*… the proportion of interfaces is rarely more than 20% of the total types, and usually around 10% [of user-defined types]*" in their large-scale study of Java programs (Tempero et al., 2008). This is reiterated with the finding that "*…classes and interfaces are used in stereotypically different ways, with approximately one interface being declared for every ten classes*" (Tempero et al., 2008). However, specific systems vary with no clear relationship between system size and amount or ratio of interface presence.

Table 20: High level system breakdown (direct +indirect implementations)

Application	User-defined Types (uT)	Implementations of any interface by non-interface types in uT	Local Interfaces (uI)	Implementations of local interfaces by non-interface types in uT	% of uT that are Interfaces	Class : Interface (ratio)
argouml	2297	6590	192	2201	8%	10:1
azureus	3971	3464	1108	4697	28%	2:1
btrace	261	92	15	28	6%	15:1
easysim	1000	1469	70	549	7%	13:1
frecol	978	1906	37	316	4%	22:1
freemind	763	1890	79	462	10%	8:1
jedit	1033	2137	65	265	6%	14:1
jhotdraw	748	2201	57	530	8%	11:1
jmeter	1123	3029	89	1249	8%	11:1
lobo	903	1492	179	691	20%	4:1
weka	1647	4720	157	2354	10%	9:1

The rightmost column of Table 20 shows the ratio of class : interface presence in each system (mean 11:1, median 11:1), this is calculated by dividing concrete classes by interfaces in each system. For example, argouml has 1949 concrete classes and 192 interfaces defined in the source code, so the ratio of classes is (rounded to) 10 classes to one interface. Steimann notes that as interface use in the JDK increases, the ratio of class to interface in the JDK remains nearly constant at 5.5:1 (class : interface) across versions (Steimann and Mayer, 2005). The JDK has fewer classes per interface than many of the systems examined - this difference may be due, in part, to the distinctive role of the JDK as a programming library. Only azureus and lobo meet this level of interface definition, with azureus having approximately one interface for every two classes.

Comparing the third and fifth columns of Table 20 shows that there are many more interface implementations in each system than can be accounted for by implementations of local interfaces (interfaces defined in the source code of the application). This indicates that interface implementation is a common means of interacting with external code such as code libraries.

4.8.3.2 Interface Size

Figure 23 shows a log-log plot of all the interfaces in the systems examined, plotting interface size (x-axis) against number of occurrences of that interface size (y-axis). Zero–sized interfaces are not shown in this plot. A break-down of interface size in the population:

- 50% of the interfaces across all analysed systems consisted of 0-2 methods
- 80% of interfaces have 8 or fewer methods
- The top 10% of interfaces have 17-393 methods
- 1% of interfaces have 97-393 methods

Marker interfaces (with zero methods) comprise 11% of the interfaces identified, indicating that they are not being used in the conventional sense to "*program to an interface*" (Gamma et al., 1994). To allow conventional interface-based programming, interfaces must provide behaviour (method signatures). These empty interfaces are present in all systems examined (containing between 2 and 75 such interfaces). While they may confound some forms of analysis, these interfaces are neither rare nor limited to specific systems.

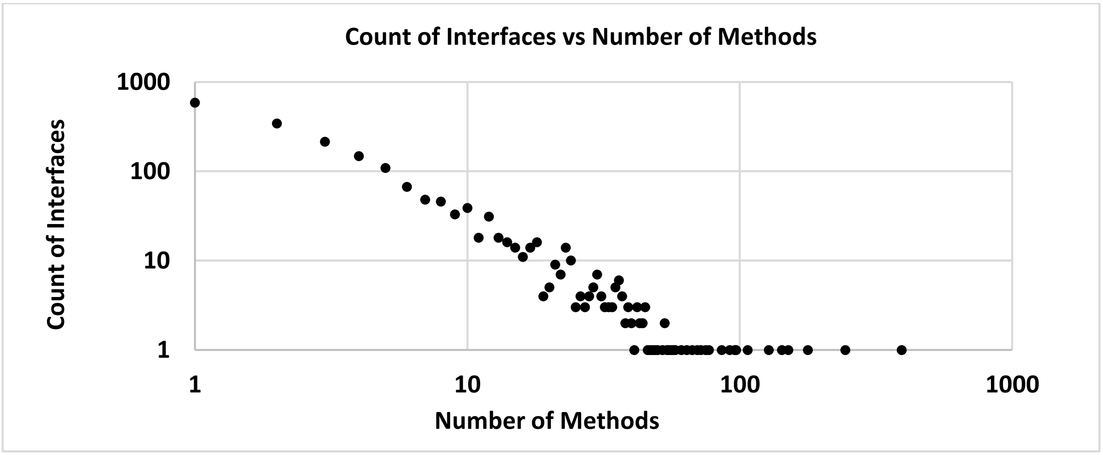


Figure 23: Number of methods in interface vs frequency of occurrence (log-log scale)

Note also the characteristic 'fat-tailed' distribution that has received much attention in the discussion of scale-free properties in object-oriented systems (Potanin et al., 2005). A fat tailed distribution indicates that there may be a 'natural' scale-free size variation in the population.

4.8.3.3 *Interface Implementation – System Level*

This section examines how much interface implementation there is in each system. For each application, Table 21 shows the number of user-defined interfaces (column 2). A number of these interfaces are inner interfaces, (included in, and shown next to the uI total). Interfaces marked 'abstract' *may* indicate the presence of generated or legacy code – these are included in the total population of interfaces in the remainder of this section.

The total number of *direct and indirect* (via inheritance) implementations of user-defined interfaces in that system (column 4), mean number of implementations per interface (column 5), and the mean number of implementations by user-defined, non-interface types (uT-uI) (column 6). For example, the first row shows that btrace has 15 (*including* 3 inner or abstract) interfaces which have 18 direct implementations, and 28 implementations in total (direct and indirect).

Table 21: Number of interface implementations per system (Total Implementations: 13342)

Application	**uI (inner)**	**Direct Implementations**	**Total Implementations (direct + indirect)**	**Total Implementations / uI**	**Total Implementations / (uT-(uI+enum))**
btrace	15 (3)	18	28	1.87	0.12
frecol	37 (6)	65	316	8.54	0.36
jhotdraw	57 (0)	110	530	9.3	0.79
jedit	65 (9)	106	265	4.08	0.28
easysim	70 (2)	143	549	7.84	0.59
freemind	79 (35)	127	462	5.85	0.68
jmeter	89 (2)	341	1249	14.03	1.21
weka	157 (9)	1110	2354	14.99	1.63
lobo	179 (1)	163	691	3.86	0.96
argouml	192 (22)	408	2201	11.46	1.05
azureus	1108 (132)	1419	4697	4.24	1.64

Some literature advocates that most references to concrete types should be via interfaces (Gamma et al., 1994; Holub, 2003; Martin, 2002). If this approach was pursued then each concrete type would be expected to implement at least one interface, this would be indicated by a value around 1 in column 6 (although the use of means hides the fact that some concrete classes are implementing many interfaces). The mean values range from the low (btrace, freecol, jedit) to perhaps higher than expected (weka, azureus), with over half the systems sitting below a value of one. This suggests that there are many types in these applications that *cannot* be accessed via an interface. In the following sections, it will be shown that different interface implementation patterns lie behind these summary figures.

Table 22 shows, for each system, how many user-defined classes or abstract classes implement interfaces, and how many interfaces are implemented. Taking the first row, argouml, as an example – 469 types (classes and abstract classes) in argouml implement zero interfaces, 588 implement one interface and so on up to 3 types implementing 20 interfaces. The total of each row is the number of non-interface types defined in each system. This total is the value uT Table 20, less the interfaces and any enumerated types defined in the system. Multiplying each cell entry by the column header and summing across the row gives the number of interface implementations for each system, as noted in the fifth column of Table 20.

The first column of Table 22 shows that most of the systems have some types that implement no interfaces (jhotdraw with a low 11% and btrace with a high 75%).

Table 22 : Breakdown of interface implementation (direct & indirect)

system	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	35
argouml	469	588	290	43	91	265	106	36	34	20	42	7	4	14	1		15	5	59	4	3		
	22%	28%	14%	2%	4%	13%	5%	2%	2%	1%	2%	<1%	<1%	<1%	<1%	0%	<1%	<1%	3%	<1%	<1%	0%	0%
azureus	1048	1197	260	78	50	158	31	18	8	2	5		2		1						1	1	
	37%	42%	9%	3%	2%	6%	1%	<1%	<1%	<1%	<1%	0%	<1%	0%	<1%	0%	0%	0%	0%	0%	<1%	<1%	0%
btrace	183	39	13	2	4	1																	
	75%	16%	5%	<1%	2%	<1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
easysim	323	387	50	20	39	28	20	27	15	13	8												
	35%	42%	5%	2%	4%	3%	2%	3%	2%	1%	<1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
freecol	405	155	54	10	10	18	102	91	21	6			1										
	46%	18%	6%	1%	1%	2%	12%	10%	2%	<1%	0%	0%	<1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
freemind	127	188	72	49	37	99	57	36	8	7	1		2	1									
	19%	28%	11%	7%	5%	15%	8%	5%	1%	1%	<1%	0%	<1%	<1%	0%	0%	0%	0%	0%	0%	0%	0%	0%
jedit	271	309	101	31	57	34	50	69	17	13	4		2										1
	28%	32%	11%	3%	6%	4%	5%	7%	2%	1%	<1%	0%	<1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	<1%
jhotdraw	76	141	79	51	85	89	95	46	9														
	11%	21%	12%	8%	13%	13%	14%	7%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
jmeter	287	285	58	22	47	57	57	132	41	14	6	8	7	5	3	2							
	28%	28%	6%	2%	5%	6%	6%	13%	4%	1%	<1%	<1%	<1%	<1%	<1%	<1%	0%	0%	0%	0%	0%	0%	0%
lobo	206	250	67	12	13	75	35	54	6	1													
	29%	35%	9%	2%	2%	10%	5%	8%	<1%	<1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
weka	285	319	170	89	100	161	90	76	59	45	20	7	4	3	5	6	4	2	1				
	20%	22%	12%	6%	7%	11%	6%	5%	4%	3%	1%	<1%	<1%	<1%	<1%	<1%	<1%	<1%	<1%	0%	0%	0%	0%

Thereafter most types implement at least one interface - in eight of the systems more than 50% of the types implement zero or one interface. The major exception to this is jhotdraw where 68% of its types implement two or more interfaces. Most of the systems have many types that implement up to 8-10 interfaces (btrace is the exception), all the way up to jedit with a type that implements 35 interfaces. Temporo et al. note that a type that implements

56 interfaces "*seems rather extreme*" (Tempero et al., 2008). Gößner reports that the number of interfaces implemented by a class is observed up to 20 in the JDK (J2SE 1.4.1-02) (Gößner et al., 2004).

The most prolific implementers from Table 22 were examined to determine their role in their containing systems:

- The implementer of 35 (33 direct, 2 indirect) interfaces is *org.gjt.sp.jedit.bsh.JThis* – which acts as a 'sink' for events – implementing many action listeners.
- *com.aelitis.azureus.ui.swt.views.skin.MyTorrentsView_Big* implements 21 interfaces (21 indirect), once again 14 of these are listeners.
- 20 *org.gudy.azureus2.ui.swt.views.MyTorrentsView (12 direct, 8 indirect)* – 12 of which are listeners.
- 20 *org.argouml.uml.diagram.static_structure.ui.FigModel*, *FigPackage*, *FigSubsystem* are part of an inheritance hierarchy where many listeners (9) are added which accumulate along with many small get/set access interfaces, and one substantial interface, (*org.argouml.uml.diagram.ui.ArgoFig*).

It appears that some of the systems examined have a substantial event-driven architecture which promotes the creation of many small *listener* interfaces.

Table 23 shows both the use of extends and implements in the systems studied. For example, row 1 shows 8% of types in argouml implement no interfaces and involve no concrete inheritance, 18% of types directly implement at least one interface and 51% implement an interface indirectly via inheritance.

Table 23: Use of implements and extends in non-interface types

System	No inheritance (super-type, direct or indirect interface implementation)	Directly implement an interface	Indirectly implement an interface	Extend	Implement an interface or extend a type
argouml	176 (8%)	369 (18%)	1063 (51%)	1468 (70%)	1920 (92%)
azureus	836 (29%)	1017 (36%)	469 (16%)	1053 (37%)	2024 (71%)
btrace	136 (56%)	24 (10%)	30 (12%)	84 (35%)	106 (44%)
easysim	178 (19%)	181 (19%)	360 (39%)	593 (64%)	752 (81%)
frecol	149 (17%)	84 (10%)	303 (35%)	667 (76%)	724 (83%)
freemind	83 (12%)	174 (25%)	286 (42%)	439 (64%)	601 (88%)
jedit	189 (20%)	286 (30%)	334 (35%)	487 (51%)	770 (80%)
jhotdraw	57 (8%)	122 (18%)	387 (58%)	492 (73%)	614 (92%)
jmeter	155 (15%)	184 (18%)	323 (31%)	702 (68%)	876 (85%)
lobo	120 (17%)	203 (28%)	237 (33%)	404 (56%)	599 (83%)
weka	56 (4%)	350 (24%)	437 (30%)	1054 (73%)	1390 (96%)

The final column shows how many types use inheritance at all. Note that the total of the second and final columns is the number of non-interface types for each system, which is uT-uI in Table 20. The same type may be counted in more than one column.

Bringing in inheritance (column 6) drastically changes the picture, compared to the variable use of interfaces indicated in columns 4 and 5, when inheritance is introduced a large majority of types in these systems either implement an interface or extend a type. Most of these systems are making extensive use of inheritance, it is also notable that many of the interface implementations are due to the presence of inheritance (indirect implementations).

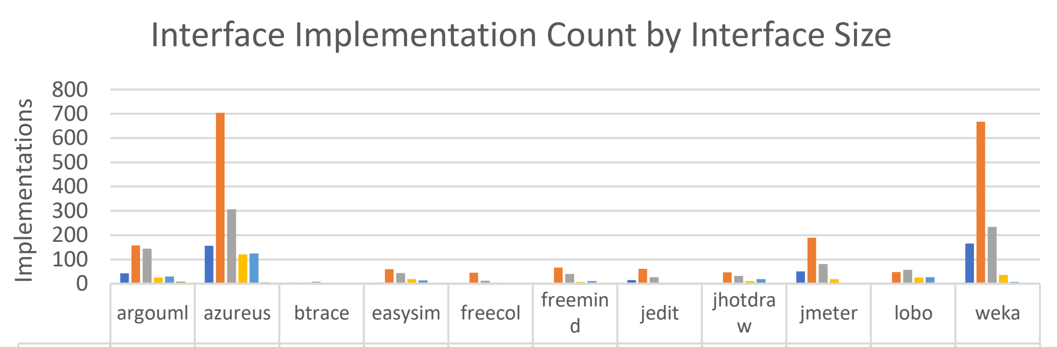
Abdeen and Shata previously found that 15-40% of classes directly implement (non-library and non-marker) interfaces. The findings here, with a direct implementation range of 10-36%, seem to be consistent with those results (Abdeen and Shata, 2012).

Tempero et al. found that most classes in Java programs are defined using inheritance from other user-defined types. Column 6 in Table 23 seems to support this with all except one of the systems having at least 71% of types defined using some form of inheritance (implements or extends). Tempero also reports that "*Applications generally have a lower proportion of classes implementing interfaces than classes extending classes, and interfaces extending interfaces is lower still*" (Tempero et al., 2008). Again, the findings in this study are consistent with that observation.

Interface Implementation Summary: [KF 4.1] Interfaces make up around 10% of total types, reaching over 20% in some systems. [KF 4.2] Many classes and abstract classes (~70%) cannot be accessed via an interface. [KF 4.3] The presence of interfaces is like findings in previous studies. [KF 4.4] 50% of interfaces overall have size 0-2. In eight of the systems more than 50% of the types implement zero or one interface. [KF 4.5] Event driven architecture promotes the creation of many small 'listener' interfaces. [KF 4.6] 23% of interfaces are implemented indirectly (at least once) via inheritance.

4.8.3.4 Interface Implementation – Interface Level

Figure 25 shows how interfaces of difference sizes (method counts) are implemented in each system. Note that inner and 'abstract' interfaces have been excluded here. Interfaces have been grouped by size based on the categories of method signature count identified by



	argouml	azureus	btrace	easysim	freecol	freemind	jedit	jhotdraw	jmeter	lobo	weka
🟦 0	42	156	5	5	3	2	15	2	50	3	166
🟧 1-2	158	704	5	60	45	67	61	46	189	48	667
⬜ 3-8	144	307	8	44	12	40	27	32	81	57	234
🟨 9-16	26	121		19	4	7	2	11	19	26	36
🟦 17-96	30	125		13	1	11	1	19	2	27	7
🟩 97-393	8	6		2						2	

Figure 24 : Number of implementations of interfaces distributed by size of interface

Figure 23 in the previous section: 0, 1-2, 3-8, 9-16, 17-96, 97-393. Taking argouml as an example, the leftmost (light blue) column indicates that there are 42 implementations of size zero interfaces in the application. Moving right, the columns indicate the number of implementations for size 1-2 interfaces (orange, 158), size 3-8 (grey, 144), and so on until the final category 97-393 (green, 8), as indicated by the data-table. Summing the figures in each column for each system matches the direct implementation count values shown in Table 21. Figure 25 shows that the most popular implementation range is interfaces of size 1-2, followed by 3-8. In these systems, if there is a 'usual' range, it is interfaces with a size in the range 1-8 methods (75% of all interfaces observed). This is close to Ferreira et al.'s finding that "*most classes have 0-10 public methods*" (Ferreira et al., 2012). A few of the systems frequently implement 0-method interfaces: argouml, azureus, jmeter and weka. For example:

org.argouml.model.DiDiagram - extended by seven other interfaces, shown in Figure 25. Which in turn have 10 concrete implementations between them. This interface is documented as giving "*better compile time safety*"[15] than passing object references

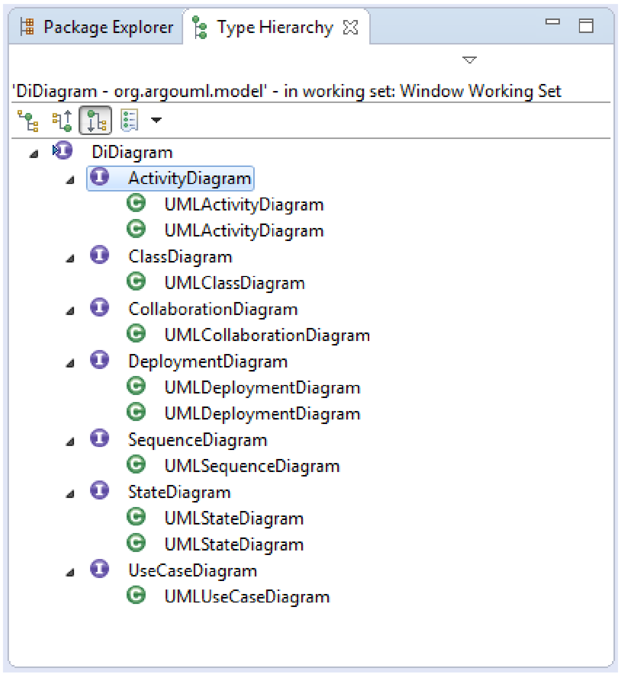
[15] org.argouml.model.DiDiagram [source code JavaDoc, v 0.34, http://argouml.tigris.org/]

around. This allows a related factory class to offer generic operations over an inheritance hierarchy.

org.gudy.azureus2.plugins.ddb – implemented by three inner classes, two of which are empty. In the case of the empty implementations, the name of the implementation carries information e.g. *RCMSearchXFer* indicates a 'Related-Content Manager' data transfer.

org.apache.jmeter.samplers.Remoteable – implemented by four concrete classes.

In all three cases, switching on subtype is used to disambiguate concrete subtypes in one place in the design – indicating that interfaces have been introduced with full knowledge that the 'switching on type' code smell would be required to make use of the concrete types. While switching on type is a possible indicator of missing inheritance, this solution is not directly applicable in these cases as the types being switched on do not share any behaviour because the shared polymorphic interface is empty. Consequently, a concrete super-type would not add any more information than the existing interface. Additionally, it is not clear that the subtypes in these classes share a common 'is-a' relationship. Based on the use shown here, a shared empty interface implementation might be regarded as a 'weak' form of polymorphism where there is substitutability with no provided or required common behaviour.



```
'DiDiagram - org.argouml.model' - in working set: Window Working Set
DiDiagram
    ActivityDiagram
        UMLActivityDiagram
        UMLActivityDiagram
    ClassDiagram
        UMLClassDiagram
    CollaborationDiagram
        UMLCollaborationDiagram
    DeploymentDiagram
        UMLDeploymentDiagram
        UMLDeploymentDiagram
    SequenceDiagram
        UMLSequenceDiagram
    StateDiagram
        UMLStateDiagram
        UMLStateDiagram
    UseCaseDiagram
        UMLUseCaseDiagram
```

Figure 25 : DiDiagram Hierarchy

Several of the systems examined have many implementations of large interfaces – most systems still have many implementations in the range 3-8 (azureus has 307), and then, as might be expected, much less of size above 8 – though azureus has a noticeable number of implementations, even in the 17+ method size range.

The high level of interface definition in azureus (one interface per two classes), indicates that there may be variations in style - in this case, the perceived cost/benefit of interfaces is favourable.

Table 24 shows the range of implementation for the interfaces in each size category. The most obvious trend is that maximum implementers decreases with interface size. Interestingly, the median does not change much with interface size, indicating that many interfaces in each category are implemented only a few times. That being the case, it is important to note that while many of the 'highly implemented' interfaces are small, not all small interfaces are frequently implemented.

Table 24 : Implementation rates for interface size bands

Methods	0	1-2	3-8	9-16	17-96	96-393
Min	0	0	0	0	0	0
Max	48	183	142	28	10	3
Mean	2.3	2.8	1.8	1.7	1.4	1.8
Median	0	1	1	1	1	2
St. Dev	5.8	10.6	6.5	2.7	1.2	0.7

Implementation Summary: [KF 4.7] There is wide variety in interfaces sizes, 0-393 methods. [KF 4.8] Three-quarters of interfaces have 1-8 methods. [KF 4.9] Many interfaces are implemented a few times; few interfaces are implemented many times. Most implementation is in the interface size range 1-8.

4.8.3.5 Use of Interfaces in Declarations

Figure 26 shows the percentage of each kind of variable and parameter that is defined using an interface. The left-most cluster of columns shows interface use in argouml – around the 5% level for all uses (class attribute, constructor parameter, constructor field, method parameter, method field and method return type). On the other hand, azureus has values ranging from 16% for class attribute up to 31% for method parameter.

Figure 26 shows clear differences amongst the amount of interface use across the analysed systems. azureus and jhotdraw are using interfaces more than systems such as btrace and jedit, which appear to make little use of interfaces. It is also interesting to note that weka, which had many implementations of interfaces in Table 22 has a low proportion of interface-typed variables. This indicates that the interfaces in weka are capturing relatively few of the internal dependencies.

There does not appear to be consistency of interface use across the systems, if interfaces are used more, they are not necessarily used more in all areas – with substantial variation in easysim, freemind and jhotdraw. In these systems, if there is a stand-out usage, then it as parameters and return types (orange and yellow bars in Figure 26 and Figure 27).

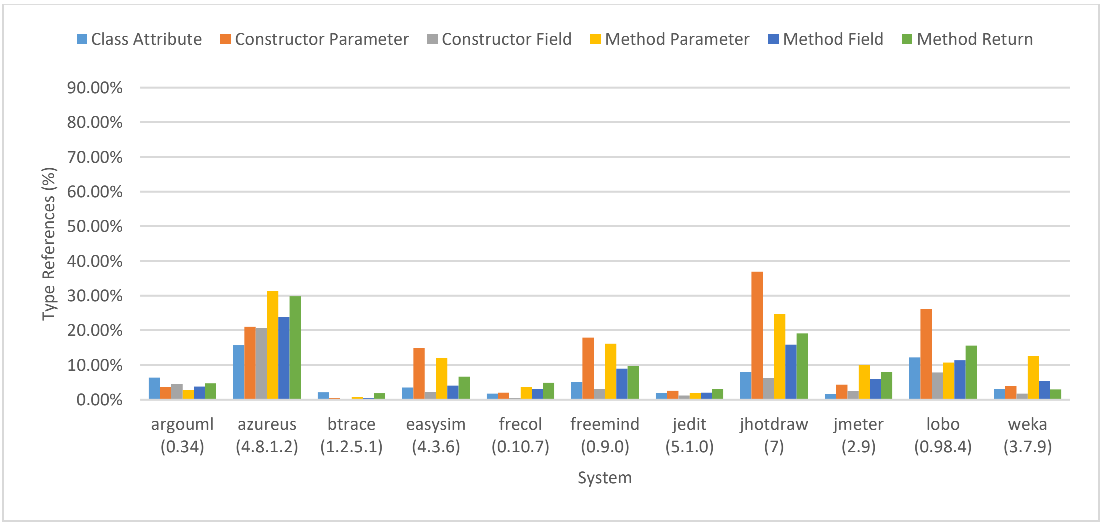


Figure 26: Percentage interface use in variables and parameters (primitives excluded)

JHotDraw has been extensively studied in the research literature as its design was influenced by the design patterns community, and is considered an example of 'good design' (Ferreira et al., 2012), including compliance with *program to an interface*. In Table 20 and Table 21 jhotdraw appears quite typical. However, in Figure 26, it does stand out with approximately double the use of interface-typed variables in all categories except as a class attribute. This might indicate that *program to an interface* is characterised by focus on abstractness in public protocols (constructor parameter, method parameter, method return).

It is not uncommon for studies to exclude size-zero interfaces and third-party or library types from code-analysis due to possible differences in structure. This is done to allow a study to focus on "*choice by the developer*"(Brekelmans, 2014) rather than programming convention or the constraints of library APIs. This has been done in Figure 27, which shows the percentage of interface references where the reference type is locally defined.

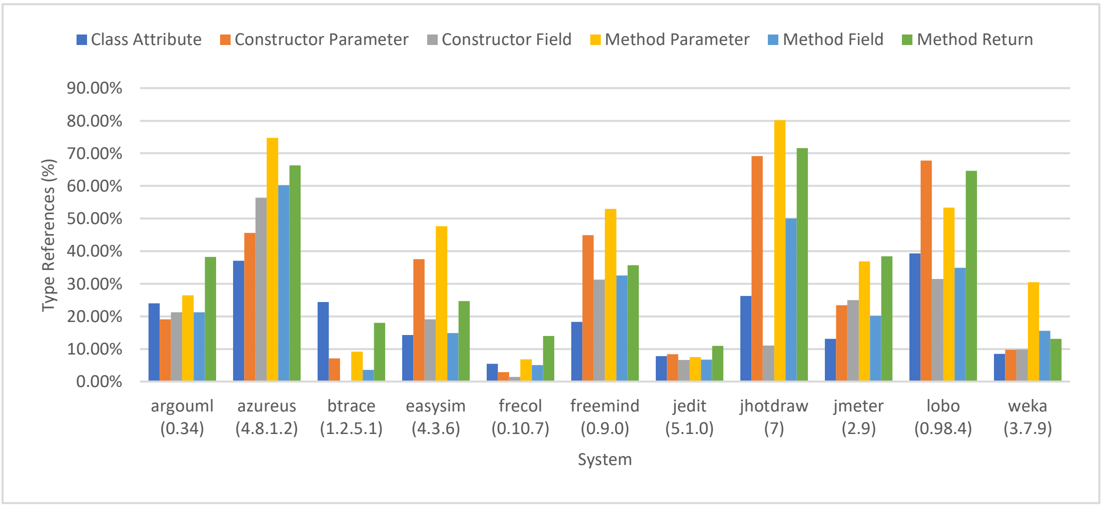


Figure 27: Percentage interface use in variables and parameters (local types only)

Figure 27 shows similar trends to Figure 26, but notably, use of interfaces is now up to 80% (e.g. method parameters in azureus and jhotdraw), with all systems showing a general increase. This difference indicates that practitioners are more inclined to use interfaces when referring to types that are locally defined i.e. their 'own code'.

Note also in Figure 27 that azureus has a more *uniform* use of interfaces across all categories, where the other high-interface-use systems have marked deficiencies in some areas, such as constructor field.

Figure 28 shows a plot of the number of interfaces defined in a system (x-axis) versus the percentage of references to user-defined types (as in Figure 27) that are interfaces (y-axis). This indicates how many static dependencies are captured by interfaces defined in each system. The most interesting feature of this diagram is the relative positioning of jhotdraw and azureus.

The diagram clearly shows the relatively high usage of interfaces that jhotdraw achieves with a relatively small proportion of interfaces (only 8% of user-defined types), whereas azureus has many more interfaces (they constitute 28% of user defined types). This shows that the number of interfaces defined in a system does not necessarily relate to the proportion of interface-mediated interactions in that system.

This may relate to the generality of the interfaces within a system. It is not clear if it is desirable to (re)design a system so that it has a lower number of highly used interfaces rather than many interfaces which each capture fewer interactions – in Figure 28 this would translate to moving azureus to the left. If the problem domain influences the number and reuse-level of abstractions that can be extracted, then this is a further indicator that a general notion of 'good' is highly variable.

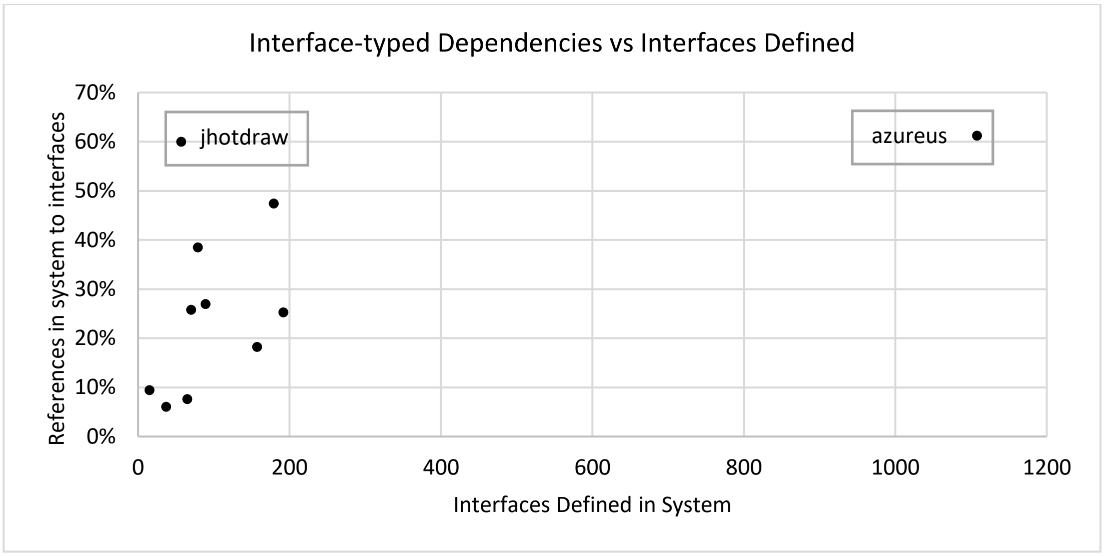


Figure 28: Percentage dependencies to interfaces vs no of interfaces defined in local source

Declarations Summary: [KF 4.10] There appear to be more interface-typed references in 'public' protocols such as constructor- and method-parameter, and return types. [KF 4.11] Practitioners are more inclined to use interfaces when they are referring to types that are locally defined. [KF 4.12] The number of interfaces defined in a system is not directly related to the amount of interface-mediated interactions in that system (highlighted by jhotdraw vs azureus differences).

4.8.3.6 Marker Interfaces

Interfaces with no methods are known as marker interfaces (Steimann and Mayer, 2005). These interfaces, can be used as containers for collections of constants, as a root super-type for hierarchies of interfaces, or as markers/flags for type checking. While these may be useful functions – this cannot be said to be "*programming to an interface*" in the conventional sense (Gamma et al., 1994), as empty interface do not guarantee the presence of any behaviour (methods) in subtypes. Marker interfaces appear in all examined systems, as shown in Table 25, but most notably in azureus (75 interfaces, 177 uses).

Row two of Table 25 indicates the number of marker interfaces in each system, followed in row three by the number of classes or abstract classes which implement marker interfaces. Row four indicates how many marker interfaces are not extended. Finally, row five indicates how often marker interfaces are used in variable declarations in each system.

Table 25 : Presence of marker interfaces across systems

System	argouml (0.34)	azureus (4.8.1.2)	btrace (1.2.5.1)	easysim (4.3.6)	frecol (0.10.7)	freemind (0.9.0)	jedit (5.1.0)	jhd (7)	jmeter (2.9)	lobo (0.98.4)	weka (3.7.9)
Interfaces	24	75	3	54	4	2	6	4	8	4	19
Implementers	42	156	5	14	3	2	18	2	50	3	166
Not Extended	5	19	1	50	1	1	1	2	1	1	8
Use as Variable	15	177	0	32	2	1	0	7	0	8	2

As markers are used to 'tag' types or group together the root of inheritance hierarchies, they can be implemented many times via inheritance. This means that inheriting types must also be compliant with the 'tagged' behaviour, with no mechanism (methods) to enforce this.

Use of markers as holders of constant values, while currently unfashionable, is less problematic. Constants are, by definition, less volatile so all implementers will have access to the same constant information while avoiding global variables. However, use of constant interfaces introduces structural coupling in that a subtype can depend on the fact that its super-type implements a constant interface.

Given that a constant interface is arguably more stable than a (non-interface) super-type, having each user of a constant refer to or implement that interface directly is more compliant with *"depend in the direction of stability"* (Martin, 2000). However, this may introduce the same interface to a hierarchy at multiple points – trading an increase in redundancy for less coupling (Abdeen and Shata, 2012). This illustrates why using a constant class rather than an interface is considered to be a better solution – references to a class do not carry the same design 'baggage' as implementing an interface.

4.8.3.7 Java Collection Interfaces

The Java language provides a variety of data structures in the form of a Collections library. These consist of abstract data types (ADTs) such as 'List' or 'Map' defined using interfaces, which are accompanied by various implementations. Figure 29 illustrates how types associated with the Java Collections framework are used across the systems.

The types are grouped by concrete class, abstract class, and interface for clarity. The most obvious point is the large preference for interfaces (76% of Collections references) over the respective implementations. For example, the Map interface is used in 4362 declarations, while the most common map implementation (Hashmap) reaches only 531 references.

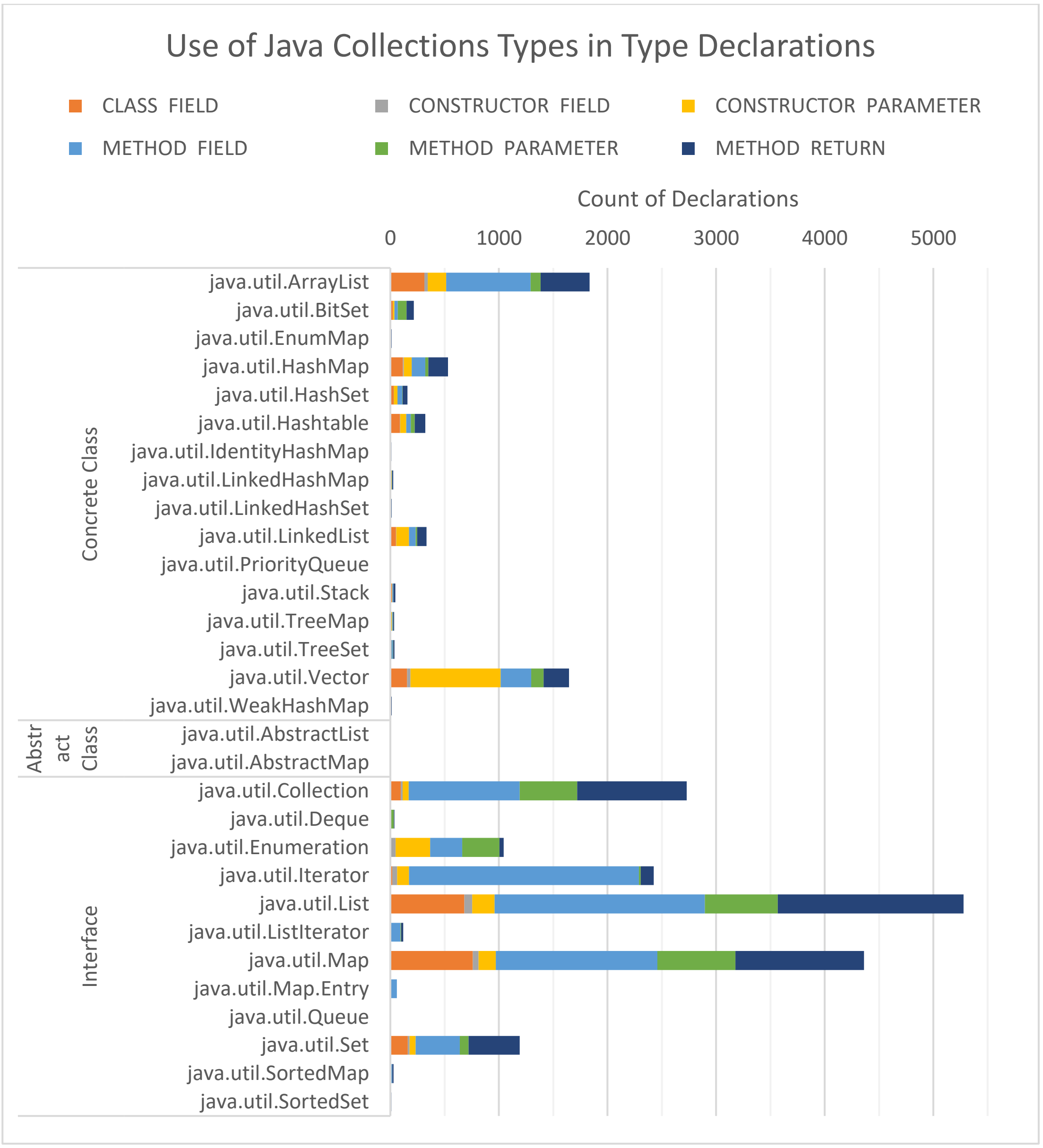


Figure 29 : Use of Java Collections Types in Type Declarations

Figure 29 also indicates that the Java collections types are mainly referenced via their respective interfaces. Though it is interesting to note a few exceptions. ArrayList is an outlier of sorts as there are both large numbers of concrete and interface (List) declarations. This may be due to the common use of lists as the ‘default data structure’ in place of tuples (which Java does not have), or primitive arrays. Collections interfaces are most present in method fields (local variables) and return types.

4.8.4 Interface Archetypes

While the sentiments from guidance are constant – decouple, abstract, modularise – the opportunity, time, and motivation to apply these principles appears to vary. As such the observed population of interfaces is where interface use meets the balance of not ‘too expensive’ and ‘useful enough’ – which may have to be the working definition of *good* for this section.

This suggests that the relevant population for future study should be ‘places where an interface could have been used’, whether an interface is present or not.

This section presents notable uses of interfaces which, it is proposed, represent acceptable compromises of cost and utility to appear in many, if not all, of the systems studied. By identifying common design trade-offs, it becomes possible to formalise the design forces involved and to try and estimate, if not identify the limits of applicability of that trade off – thus helping to reduce the number of unassessed interfaces. As pointed out in Al Dallal and Morasca’s work on reuse – not all elements of a design are constructed with every quality attribute in mind – so we should assess the salient qualities of each element (Al Dallal and Morasca, 2014).

4.8.4.1 Listener Interface

A key feature of OO systems is that objects communicate via message passing. This allows various architectures to emerge, one of which is ‘event driven’ architecture. In Java, this is often achieved via subscribing listener objects to other ‘observable’ objects. This common style of programming inflates the implementation count, given its popularity across systems – it may simplify analysis to consider the *listener structure* of a system as a distinct architectural feature.

While not enforced, it is conventional to append the term ‘Listener’ or ‘Observer’ to the end of the name of a listener definition. This is commonly used to indicate that an object is compatible with Java’s native *java.util.Observable* implementation. The corpus examined contains 393 (387 listener, 6 observer) such interfaces, a summary of which are shown in Table 26.

Table 26 illustrates that just under half of the listeners identified have the expected single method required by the Java event-handling model (`void actionPerformed (ActionEvent e)`*)*. The responsibility of a listener is well-defined and is indicative of a high-level decision to use an event driven approach. Variations in listener type names and the number and name of the listener methods indicate that there is still some choice available to the practitioner even when the overall structure is fixed by a higher level architectural decision.

Table 26 : Listener Interfaces

System	**Listener Interfaces**	**Implementers**		**Methods**											
		Range	**Total**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**12**	**13**
argouml	17	0-15	43	2	5	4	2	2	2						
azureus	287	0-183	543	8	133	69	29	20	15	7	2	1	1	1	1
btrace	1	1	1		1										
easysim	9	0-12	17	3	5		1								
freecol	3	0-1	1		1	1	1								
freemind	6	0-8	14		5	1									
jedit	11	0-2	11		3	2	2	2		1		1			
jmeter	13	0-25	70	1	5	4	1	1				1			
lobo	7	0-2	6		5	1						1			
weka	32	0-16	107	1	29	1			1						
jhotdraw	7	0-3	6		2	2	1	1			1				
			Total	15	194	85	37	26	18	8	3	4	1	1	1
			%	3.8	49.4	21.6	9.4	6.6	4.6	2.0	0.8	1.0	0.3	0.3	0.3

4.8.4.1.1 Anonymous Classes

In addition to 'full' classes, Java allows the use of *anonymous* classes which are unnamed and often used to describe objects which will only be used once or few times. Note that anonymous classes are not included in the type counts for each system. Table 27 shows the number of anonymous classes used in each system (column 2). Columns 3 – 7 show counts for implementation of interfaces by anonymous inner classes. It is notable that many inner classes are used to implement the *EventListener* interface – which is a common super-interface for 'wiring up' user interface elements to their handlers. The other popular category of interface is Runnable – which can be used to define some sub-process to be executed by a thread. Azureus, once again stands out. While it does not use the interfaces described above to a great extent, there are various specific '-Listener' and'-Performer' interfaces defined in the system which have the expected levels of use, indicating a preference for local interfaces over library types.

Overall, Listeners (local and library defined) account for around two thirds (62%) of anonymous classes. This indicates that an interface can be useful, even when not implemented by a 'real' class. By defining a simple interface such as an *EventListener*, a component is advertising what is required to interact with it. This is most like the Server/Client category of interfaces described by Meyer et.al "*...that enables its implementors to profit from some service offered by the caller.*", in this case, access to an event subscription mechanism (Steimann and Mayer, 2005).

Table 27: Anonymous classes in examined systems

System	Total	java.util. EventListener	java.lang. Runnable	Local '-Listener'	Local '-Performer'	Other
argouml	148	53 (36%)	35 (24%)			60 (40%)
azureus	1904	1 (<1%)	147 (8%)	1186 (62%)	143 (8%)	427 (22%)
btrace	41	1 (2%)	13 (32%)			27 (66%)
easysim	310	227 (73%)	83 (27%)			
freecol	272	120 (44%)	30 (11%)			122 (45%)
freemind	138	62 (45%)	28 (20%)			48 (35%)
jedit	132	45 (34%)	81 (61%)			6 (5%)
jhd	182	123 (68%)	53 (29%)			6 (3%)
jmeter	98	49 (50%)	3 (3%)			46 (47%)
lobo	130	33 (25%)	40 (31%)			57 (44%)
weka	597	567 (95%)	11 (2%)			19 (3%)

However, this highlights an issue with Meyer et al.'s proposal that there is a clear beneficiary in this relationship with one side of the server/client relationship gaining the most 'benefit'. Arguably an event-producer benefits from being able to gain subscribers. Indeed, an event producer that cannot be subscribed to is not very useful. So, it might be argued that the listener mechanism is an intrinsic part of the event producer. Consequently, it may be necessary to assess such interfaces in terms of the relationship they mediate e.g. publish-listen, observe-observable, in addition to their 'use' by the other parts of the system.

4.8.4.2 Constant Interface

Constant interfaces used to store constant values. These occasionally contain static 'utility methods' and, as such, are sometimes considered poor object-oriented style.

Across all systems examined, there are 203 interfaces with no methods, while some are suggestive of the heads of hierarchies or Marker interfaces (see 4.8.4.3). Constant declarations were not recorded in this study, however, like listeners, some practitioners follow a naming convention which identifies some instances of this kind of interface. Table 28 shows the constant interfaces which follow this naming convention.

Given the small number of constant interfaces, the purpose of each interface was determined by manual inspection. It is interesting to note that there are distinct categories of constant interface which are obvious from the definitions of the constants:

- Pattern matching – the interface records constants which represent flags or tags in some external format such as byte-codes or html tags.
- Enumerated types – the interface defines flags related to local state, often with smaller subsets of distinct flags. These often require no values other than to be

comparable and could be replaced with the newer enumerated types introduced in Java 1.5.

  - Note that some enumerated types use rotated bit patterns for uniqueness checks.

- Properties – stored values or strings for configuration/localisation.
- Other cases are where the constant interface behaves as an object factory.

Table 28 : Constant Interfaces, interfaces documented as 'generated' are marked with an asterisk (*).

System	Constant Interface	Implementers	Constants	Use
azureus	org.bouncycastle.math.ec.ECConstants	4	5	Factory
	org.gudy.azureus2.ui.swt.mainwindow.IMenuConstants	2	83	Enum (Bit)/Properties
	org.gudy.azureus2.ui.swt.progress.IProgressReportConstants	6	25	Enums (Bit)
	org.gudy.azureus2.ui.swt.twistie.ITwistieConstants	2	4	Enum (Bit)
freecol	net.sf.freecol.common.networking.NetworkConstants	2	5	Enums
	net.sf.freecol.server.ai.goal.GoalConstants	1	1	Property
jedit	de.masters_of_disaster.ant.tasks.ar.ArConstants	1	11	Properties
	installer.BZip2Constants	2	11	Abstract Class
	org.gjt.sp.jedit.bsh.org.objectweb.asm.Constants	1	177	Pattern Matching
	org.gjt.sp.jedit.bsh.ParserConstants *	13	140	Pattern Matching
	org.gjt.sp.jedit.bsh.ParserTreeConstants *	1	39	Pattern Matching
jmeter	org.apache.jmeter.protocol.http.util.HTTPConstantsInterface	3	39	Pattern Matching
lobo	com.steadystate.css.parser.SACParserConstants *	2	83	Pattern Matching
weka	weka.classifiers.lazy.kstar.KStarConstants	3	16	Enums/Properties
	weka.gui.graphvisualizer.GraphConstants	5	6	Enums

While the Pattern Matching constant interfaces tend to be larger, and enumerated types tend to be smaller - there is no clear distinction by the *size* (number of constants) of the constant interfaces.

Defining enumerated types avoids errors associated with arbitrary 'flags'. Properties can be handled in various ways in modern systems – Java has a native properties subsystem; dependency injection is also popular. Property entities represent an effort to document and isolate these kinds of constants in a single place, which satisfies modularity. Similarly, pattern matching interfaces *document external values* which the program must be 'fluent' in when dealing with external artefacts – opcodes, html tags, etc. Again, the constant interface acts as an insulating barrier between the internal representation (constant fields) and a suitable external representation.

In terms of guidance, however constant interfaces present some issues. The figures in Table 28 indicate that constant interfaces are usually implemented to access the constant values contained therein. This allows constants to be used as if they are local, which creates a static dependency on the structure (rather than the behaviour) of an interface at compile time. Furthermore, should the constant interface change, there is no other record of the constants used or required by the system. This is contrary to the usual design preference where protocol/policy (interface) and mechanism (implementation) are distinct.

This runs counter to most of the design guidance and intent surrounding interfaces – which are supposed to indicate flexibility. This is perhaps acceptable because of the very low chance of change in the purpose and function of the dependency – as it is a direct mapping to part of the problem domain.

4.8.4.3 Marker Interface

Marker interfaces are discussed in the literature (Abdeen and Shata, 2014; Fontana et al., 2016; Steimann and Mayer, 2005) – primarily when excluding them from studies – these sit in a niche in the practitioner's toolbox. In the first case, there are some places where marker interfaces effectively *must* be used due to design choices in the Java language – Serializable and Cloneable being the main examples in the design literature. It is unclear how these examples in the core Java language influence the perception of marker interfaces in practice.

More generally, Marker interfaces are characterised by their lack of behaviour. However, this is also a property of the constant interfaces identified above. Excluding the known constant interfaces identified above, leaves 188 zero-method interfaces which do not announce themselves as constant interfaces. Note that some of the listener interfaces described above in Table 25 may be more correctly categorised as markers, as they are used to group other more functional listeners into an abstract hierarchy.

An examination of the marker interfaces reveals no obvious categories or naming conventions (note that these are all interfaces from application source code and not the *usual suspects* from the java library – *Serializable* and *Cloneable*). It appears that this easily recognisable feature (no methods), is not sufficient to identify a specific motivation for use. A more detailed analysis of the use of these interfaces, and perhaps their role in the surrounding design and evolution is required to gain more insight into their purpose.

4.8.4.4 Token Decoupling

Much is made of decoupling in both the design guidance (comprising advocacy and design manuals) and empirical literature (studies on coupling metrics). However, there is evidence that the ‘extract interface’ refactor is sometimes performed mechanically, with little or no improvement on the situation at hand.

Here, token decoupling is defined as having occurred when an interface is present, but does little to improve the design. This is most notable when a large and non-cohesive class is the single implementer of a similarly diverse and unfocused interface. While this does technically eliminate a concrete dependency from the clients of the implementer, there is no distinct useful abstraction and the structure of the interface and its implementer are synonymous.

No formal definition of token decoupling (for the purposes of detection) is offered as the examples below have been noted during review of designs of the systems under study in this work. As the ‘token decoupling’ interfaces noted during inspection were large, the top 1% of interfaces (by method count) have been examined. Note that in most cases the larger interfaces are reasonable abstractions over the problem domain. So, size alone is not enough to determine the ‘goodness’ of an interface.

Table 29 shows the top 1% of interfaces (by method count), with the ‘token decoupling’ interfaces highlighted in red. Abdeen et al. noted that there is a large incidence of unused methods in interfaces – this might be expected if interfaces are being extracted ‘by rote’ (Abdeen et al., 2013b). The function of each interface was determined by manual inspection and is noted in the *Comment* column.

The token interfaces do not document an abstraction such as the need of the client (or group of clients), a service offered by the implementer, a transient role, or an ability. Consequently, they may harm navigation and comprehension of the design. Although, adding an interface, even to a badly designed class still provides runtime flexibility. Accordingly, the motivation to create a token interface *must* be considered an architecture-level design decision, creating a modular boundary that serves to insulate the design at large from a ‘toxic’ or ‘unwieldy’ component, regardless of the suitability of the abstraction (SRP) or the needs of the interface’s clients (ISP).

The existence of large entities in designs is not unprecedented – the recent work on the ‘scale free’ structure of software (Potanin et al., 2005; Taube-schock et al., 2011) indicates that entities with high fan in and high fan out may be inevitable. However – there are, as noted above, cases where single abstractions appear to be doing too much.

Table 29 : Sample 'token decoupling' interfaces

System	Interface	Methods	Comment
argouml	org.argouml.kernel.Project	70	Extracted mechanically with the intent to refactor, which has not been done.
	org.argouml.language.cpp.reveng.Modeler	109	Object builder
	org.argouml.model.CoreFactory	77	Factory
	org.argouml.model.CoreHelper	151	Stateless operations on UML elements
	org.argouml.model.Facade	393	Cohesive, but Object references force clients to do lots of casting.
	org.argouml.model.MetaTypes	143	
azureus	org.gudy.azureus2.core3.download. DownloadManager	161	Diverse functions, client groups, and levels of abstraction
	com.aelitis.azureus.ui.common.table.TableView	67	View element access
	org.gudy.azureus2.core3.download.DownloadManagerState	72	Properties getter/setter, uses primitive flags
	org.gudy.azureus2.core3.peer.PEPeer	86	Domain element
	org.gudy.azureus2.core3.peer.PEPeerManager	97	Tools for domain element
	org.gudy.azureus2.plugins.download.Download	96	Domain element
	org.gudy.azureus2.plugins.PluginConfig	107	Plug-in architecture hook
	org.gudy.azureus2.ui.swt.views.table.TableOrTreeSWT	178	Generic view element access
freemind	freemind.modes.MindMapNode	92	Diverse functions
	freemind.modes.mindmapmode.actions. MindMapActions	64	Tools for domain element
	freemind.modes.ModeController	75	Mostly a controller view of the model. Some unnecessary behaviour.
lobo	org.w3c.dom.css.CSS2Properties	244	Documents domain element

Finally it is interesting to note that most of the large interfaces examined in the top 1% are focused and have high conceptual cohesion. This raises the possibility that large interfaces, in contrast to large classes, may be better maintained as they grow. Possibly due to the low cost of change in interfaces compared to concrete classes.

4.8.4.5 Other Naming Conventions

In this section, a brief breakdown of the population is provided showing the need for individual assessment at the interface level.

Excluding the identified interfaces noted in the above sections (marker, listener, constant, token) from the population this leaves 1325 (69.6%) interfaces. Of these, there are other notable groups:

- 43 interfaces across 8 systems have names ending in '*-able*', 40 of which have only one or two methods. These correspond to Liskov's 'grouping' and Meyers 'enabling' archetypes.

- There is evidence of design pattern use with 33 interfaces with names ending in '-*Factory*', 24 ending in '-*Adapter*', 10 in '-*Component*', and 4 in '-*State*'.
- Finally, there is evidence of graph or tree structures, with 13 interfaces ending in '-*Node*'

While not completely reliable, naming conventions are clearly important and may give some indication of the purpose an interface and thus which criteria might be used for its assessment.

Marker and constant interfaces may not be considered to 'program to an interface' in the conventional sense as the clients of the interfaces are not able to 'ignore the implementation details' (Gamma et al., 1994). In the case of the markers - these require that the client pay attention to the implementing type(s) as no information is available programmatically via the marker. While clients of constants depend on non-abstract properties of an interface. In both cases – no implementation details are hidden from the clients.

Listener interfaces are the low-level manifestation of a high level (architectural) decision. Decoupling between event source and event sink must occur, so interface use is in a sense 'mandatory'. With this in mind, listener interfaces may be better assessed with an appreciation of the constraints which cannot be changed. This still leaves some choice in implementation details in terms of which listener methods are grouped together in interfaces and naming choices.

Interface Archetype Summary: [KF 4.13] Marker, constant, and listener interfaces are not programming to an interface in the conventional sense, so must be assessed by other criteria. [KF 4.14] Naming conventions appear to be an important way of recording design intent in interface use.

4.8.4.6 Presence of Previous Models

As discussed above (2.4.1), there have been previous efforts in the direction of identifying different use categories of interfaces. The most notable is the set of uses defined by Steimann and Mayer derived from "*the semantics of relationships between types*" (Steimann and Mayer, 2005):

> Idiosyncratic – an interface has a single implementation and the public interface of the implementing class matches the interface. No specific client.

Family – heads multiple implementations of a family of classes, often with alternative implementations or differing run-time behaviour, such as data structures. No specific client.

Client/Server – the implementer is the server, which offers the services needed by different clients. Different interfaces may be offered to different clients.

Server/Client – implementing the interface allows access to services offered by some server(s), generally for the benefit of the client.

Server/Item – enables processing of item interface objects for the benefit of some third party e.g. Comparable, Printable. These are typically the '-able' or '-ible' classes in Java.

It is interesting to note that most of these categories do not constrain the form of the interface, this may go some way to explain why the proposers were unable to derive a robust means of classifying interfaces into these categories. In order:

Idiosyncratic interfaces can be detected to some extent (733, 40% of interfaces have a single implementations), however, congruence with the public methods of the implementer was not recorded in this study. Furthermore – the 'specificity' of a client of the interface is a subjective as all abstractions have inherent assumptions.

- Proposal to improve definition: Idiosyncratic interfaces contain all of the public methods of their implementer, and all interactions with the implementer are via the interface.

Family – Some interfaces *are* implemented indirectly via inheritance (KF 2.6 23% of interfaces are implemented indirectly at least once via inheritance), however, the Family category requires that implementations are 'variations' without providing some notion of how far is too far. Implementation similarity has not been assessed in this study.

- Proposal to improve definition: In addition to having variations in implementation, the implementers of the family interface must be used interchangeably by some clients(s) of the system containing them.

Client/Server – The definition is clear, but rather general. It is possible to detect types which implement multiple interfaces – it is not clear if this is sufficient to qualify as a 'server' given how specific some interfaces can be. Some additional criteria may be required, such as the existence and attributes of clients.

- Proposal to improve definition: The server class should be able to perform its functions without having any clients via these (external) client-service interfaces. This is in contrast to dependency inversion via (internal) interfaces to sub-components of the server.

Server/Client – The notion of 'benefit' is not always clear at a modelling level, as is the idea that a type may 'publish' an interface which allows others to interact with it. However, these are semantic distinctions. In practice – it can be difficult to determine the 'direction of benefit' in the relationship between types. For example - some types of abstraction create dependencies in unexpected directions – such as double dispatch, or dependency inversion.

Server/Item – This category specifically references certain classes in Java. There is evidence that naming conventions are useful enough to be used in practice for some kinds of interfaces (KF 4.14). However, it not clear if all such interfaces can be included in this category as the name of a type places on constrain on its definition, or the details of its implementation(s).

It is interesting to note the amount of semantic information required to assess these interface categories. For example – the notion of an internal component/composition relationship is obvious to a designer, but not annotated on an interface. This may be mitigated to some degree by the addition of criteria related to interactions with these interfaces.

While some insight may be gained from these archetypes, the semantic details required to make these classifications is not recorded in the structure of source code. Unless this semantic information is preserved, these categories may be best considered at the level of other design tools such as CRC cards. As such, a more operational definition of 'direction of benefit' which can be inferred from available source code may allow progress in the absence of semantic information.

4.9 Answering the Research Questions

This section relates the material in the discussion back to the research questions.

4.9.1 RQ4.1: How is interface definition used in systems?

Interface definition in the systems examined matches previous work when compared at a high level *[KF 4.2]* – using average percentage of types defined as previous work reports.

However, averaging across systems hides wide variation among systems *[KF 4.1]*. In addition, care must be taken to separate the ideas of interface definition, interface implementation, and interface-typed dependencies which all vary in different ways within each system *[KF 4.12]*.

The proportion of local types that are interfaces appears to have no obvious relationship to other features such as system size *[KF 4.12]*. Low numbers of interfaces observed in some systems indicate that there is high tolerance for concrete dependencies - this may indicate variations in the local 'style' for each system *[KF 4.2, 4.3]*. However, there is also evidence that the amount of 'abstraction' (abstract dependencies) gained from interfaces varies *[KF 4.12]*, indicating that there cannot be an 'optimal' proportion of interfaces and that assessing the remaining concrete dependencies may be a better indication of the system-level effectiveness of abstractions *[KF 4.4]*.

Special cases of interface use which are commonly excluded from studies (marker, constant, listener) are common to all systems, or are common programming idioms. In this case naming conventions and similar structure, allow this subset of interfaces to be isolated from a more general assessment *[KF 4.13, 4.14]*.

For example, many of the systems appear to use event-driven architectures, indicated by the presence of many 'listener' derivatives which should perhaps be excluded or at least separately assessed *[KF 4.5]*. It would be reasonable to argue that once an event-driven architecture has been chosen, the choice to use individual listener interfaces is not 'optional'. At the very least, these specific interfaces can be subject to more specific assessment criteria, specifically, how well they preserve or promote the high-level goals of an event driven architecture.

4.9.2 RQ4.2: What is the profile of interface size across the systems?

The relevant design guidance advises which interactions should be interface mediated (as many as possible), or the subjective properties of interfaces such as being a 'complete' abstraction (Consistent, Essential, General, Minimal, Opaque (Hoffman, 1990)). This advice has no direct bearing on size, and illustrates that many design considerations operate at a higher level than implementation details such as how many methods are in an interface, offering little guidance on this matter.

The interface population is composed mostly of small interfaces - 50% having two or less methods, and 80% having 8 or fewer methods *[KF 4.4]*. Size allows identification of one kind of outlier for separate assessment *[KF 4.8]*. Marker and constant interfaces satisfy very specific design needs *[KF 4.13]*. Setting aside the general question of 'proper use', meeting a specific need maps to specific assessment – allowing segments of the population to be treated separately. This has the advantage of making the remaining population more uniform *[KF 4.8]*.

Similarly, looking at the larger outliers – the top 1% of interfaces are size 97-393 methods. On further examination, these very large interfaces meet very specific design needs:

> External documentation – some set of features or entity from the problem domain – this 'guarantees' conceptual cohesion, but may not otherwise be of high measurable quality.
>
> Token decoupling – decouples the rest of the system from a class, which usually has a large and diverse set of methods. While this does avoid a concrete dependency, the interface has the same low conceptual cohesion as the (usually) sole implementer.

These illustrate that in practice, unusual properties may be justified by some design needs, such as very large numbers of methods in interfaces *[KF 4.7]*. However, these justifications are not always obvious, so size alone is not sufficient to classify even the extreme outliers as *good* or *bad*. The examples discussed above are obvious due to their size, however there may be other motivations for interface use at more 'usual' interface sizes which are less visible.

4.9.3 RQ4.3: To what extent are interfaces implemented across the systems?

In non-interface types, direct implementation of interfaces is found in 12-48% (median 35%) cases, while 14-65% (median 43%) indirectly implement interfaces *[KF 4.6]*. These levels of use indicate that interface use for type definition is relevant to practitioners but is highly variable. The high variability between systems, again, indicates that there is a different amount of tolerance for concrete dependencies between systems *[KF 4.10, 4.12]*.

It appears that the implementation of interfaces is amplified by the presence of inheritance – casting some doubt on the 'either/or' relationship presented between composition and inheritance portrayed in the guidance (particularly the patterns literature) *[KF 4.6]*.

To completely comply with 'program to an interface', there would have to be a way to interact with all the concrete types in a design via an interface. This is not the case as 19-76% (median 28%) of the classes and abstract classes in the systems examined cannot be accessed via an interface *[KF 4.2, 4.3]*. This does agree, at a high level, with Tempero et al's finding that around 75% of non-interface types are defined using some form of inheritance *[KF 4.3]*. However, the variation indicates major differences between designs.

Across all systems examined, the trend is that smaller interfaces (size 1-8) are implemented more often *[KF 4.9]*– this becomes more evident in the larger systems which may relate to the claim by Tempero et al that "*…we can expect that as programs get larger, the numbers of implementations of popular interfaces and the number of descendants of popular classes will grow without limit.*" (Tempero et al., 2008). This does not address the interaction *between* inheritance and interfaces – where the presence of inheritance and interfaces together appears to give a 'boost' to interface use via indirect implementation *[KF 4.6]*. While this could be achieved via interface inheritance and defining more interfaces, this approach (as seen in *azureus*) requires much more interface implementation than is usual. This may not sit well with practitioners who express concern over the maintenance cost of what is viewed as unnecessary interface definition "*cluttering the codebase*" (See 3.4.5.1 Survey: Program to an Interface).

Ease of implementation may contribute to this difference as larger interfaces carry more behaviour and thus require more effort to implement. On the other hand, smaller interfaces may be used to represent interactions between types (roles), rather than types themselves. This would go some way to explain the larger number of implementers for some small interfaces *[KF 4.9]*.

Of the interfaces observed, few with over 50 methods have more than a couple of implementations, while smaller interfaces have tens (size 18 and below) or even hundreds (size 1) of implementations *[KF 4.4.9]*. On examining the implementers of interfaces, some types implement very high numbers of interfaces - these interfaces tend to be small indicating that the implementing classes play many roles *[KF 4.10, 4.12]*. This supports Potanin et al's finding that there are "*hub objects*" in OO systems which take part in many more interactions than other parts of the system (Potanin et al., 2005). However, Potanin et al. identified high fan-in hubs among *runtime objects* rather than the static dependencies identified here. It is not clear if these can be compared in a useful way.

Potanin et al. used memory models of Java programs which do not exactly reflect the usual programmer-facing static view of coupling (Potanin et al., 2005). Indeed, high fan in to abstract types (see 8.2.4.2) may be considered desirable. This illustrates that using static metrics to assess dynamic structures yields results which require careful interpretation to preserve the intent behind the metric or guideline.

4.9.4 RQ4.4: How are interfaces used in the systems?

'Use of an interface' in relation to guidance is where an interface-type reference is used by a client where a concrete type may be used. This study considered the use of interfaces in declarations of: class attributes, constructor parameters, method parameters, constructor fields, method fields, and method returns.

There was considerable variation in the systems examined – there were a few clear 'low use' systems where interface use is around 3-5% in all categories *[KF 4.10]*. In systems which reach up to 30% there is much variation, with constructor parameter, method parameter, and method return showing more interface use than the other categories.

Previous studies on interfaces have attempted to isolate instances where the practitioner has a 'choice' unburdened by library use or language-library conventions. When only locally (practitioner) defined types are considered, the use of interfaces across all systems increases, but not uniformly *[KF 4.11]*. Notably, the systems which previously showed low interface use overall still fail to make much use of interfaces (~10%) when there is 'choice' – indicating that even when there is an unhindered opportunity to use an interface, this opportunity is not always taken. However, this finding, does lend some weight to the aim of studying interfaces where there is practitioner choice, which has been used to frame other work in the area (Abdeen and Shata, 2012).

The systems which show very high interface type dependencies (~60%) in some categories still fail to meet Holub's goal of programming in terms of interfaces 80% of the time (Holub, 2003). It may be the case that this is an aspirational goal in the 80/20 sense, rather than the basis for a metric.

The high use of interfaces again appears primarily in method parameter, constructor parameter, and method return – what may be considered the more public parts of a class *[KF 4.10]*. However, this is not consistent between systems – even in the high interface-use systems, there is variation in the relative ranking of these three uses.

While other factors may be in play, it appears that practitioner ‘style’ and approach to design can account for an order of magnitude difference in interface use between systems. This raises questions about what feedback different practitioners would want to receive about interface use in their designs. For example – how distinct client subsets must be before application of interface segregation principle, or what criteria are used to justify the choice between interface and abstract class. These are perhaps best characterised as ‘consistency’ – where the practitioner should be notified if they are adhering to their own standards.

Further evidence of the influence of ‘style’ or domain variation can be seen when comparing azureus and jhotdraw. These systems achieve the highest levels of use of interfaces (~60%), with wildly different proportions of interfaces. This illustrates that the same level of ‘interface use’ may require different numbers of interfaces. There is no indication that these dimensions vary due to the same factors – the desired degree of abstraction/flexibility at the design level, and the number of interfaces at the implementation level.

It is also clear that naming conventions allow some common uses of interfaces to be identified (and thus assessed) more accurately *[KF 4.14]*.

4.10 Discussion/Interpretation

This section discusses the results in the broader context of the research goal – closing the gap between guidance and practice.

4.10.1 Interface Definition

The presence of interfaces is like that reported in previous large studies. This is usually expressed as % of types defined in the local source as this allows comparison between systems. Previous work has indicated levels of around 10% interface definition on average across systems which broadly matches the findings here. There is, however, no good explanation why this should be the *typical* value, and there is obvious variation amongst the systems examined.

This cross-system comparison raises the issue of what should be considered a *proper* interface – while many studies exclude marker or constant-bearing interfaces – it is notable that these are present in most systems. Overall these do not have a great impact on system level metrics, so while it may be appropriate to exclude them from some analyses, they should not be considered uncommon.

In terms of guidance – based solely on proportion of interfaces, the key message of ‘program to an interface’ appears to be applied less than may be expected - there are simply insufficient interfaces present to be *able* to refer to all non-interface types via an interface. This may indicate that compliance with ‘program to an interface’ is subtler than adding interfaces at every dependency and, in practice, does not simply mean that all non-interface types should be referenced via an interface.

With this in mind, Gamma et al. does acknowledge that the ‘native interface’ of a type is still an interface – so the issue may be how to distinguish when a dependency on a concrete is acceptable, and when the dependency must be replaced with an abstraction (Gamma et al., 1994).

With some interfaces being considered *improper* use, and some native interfaces being considered on par with abstract interfaces – the notion of what counts as a *real* interface for the purposes of applying guidance becomes more complex depending on which of these definitions the practitioner accepts. This may form the basis of local ‘style’, and practitioner tolerance for concrete dependencies.

There *is* some evidence of difference in *style* between systems – with some systems being very interface light, others where interfaces have been incorporated extensively. It appears that the developers of these systems use different criteria for determining how many interfaces should be in their respective systems and therefore different approaches to Program to an Interface.

4.10.2 Interface Size

There is little guidance relating to interface size – much of the design guidance in general is related to modelling considerations (cohesion) and relationships (dependencies), where size is the scope of one ‘reason to change’, ‘single responsibility’, or ‘abstraction’. In contrast, the primary means of measuring size in the empirical literature is by counting abstract method definitions.

This immediately raises some questions about common, but nonetheless *edge* cases. Interfaces with no method signature definitions are present in most systems examined, suggesting that these are more common than the literature expects. Most studies set these aside as they may not be considered *real* interfaces for the study e.g. confounding metrics. Without method signatures, these interfaces are found to serve two main purposes – marker interfaces and constant interfaces.

Marker interfaces are empty of everything but a name and are used to *tag* an implementer. They appear in all the systems examined, ranging from 2-75 per system, overall accounting for 11% of the interfaces studied – marker interfaces are not rare or unusual. While marker interfaces are used by the Java language (e.g. `Serializable`) the JVM has support for detecting these *flags*.

Marker interfaces used by practitioners lack this support and may require type-checking to detect the presence of a marker interface. Guidance has little direct impact on marker interfaces – there is no behaviour to *program to* and no conventional notion of *responsibility*. Marker interface may however impact coupling (low coupling) and bypassing abstractions (if cast).

Constant interfaces are used to store constant values in a system and may be implemented to share these constants via type inheritance. In modern practice Java, this may be considered an anti-pattern as interfaces are an abstraction mechanism while the representation of constants is, by definition, an implementation detail. Assessing the quality of these interfaces is limited to assessing cohesion i.e. how many constants each implementer makes use of.

Moving to from the small to the very large, the most notable categories of very large, the top 1% of interfaces range from 57 to 393 methods. Anything near 100 methods seems to be excessive and unlikely to meet design guidelines such as Martin's Single Responsibility Principle (Martin, 2005b). The notable uses for large interfaces are *external documentation* and *token decoupling* (as noted in 4.8.4.4).

External documentation interfaces reflect some external standard such as a document format – creation of these interfaces is *mechanical* in a sense, as the details of the concept being represented are defined in the problem domain. Indeed, some of these interfaces have been generated automatically. In these cases, the conceptual cohesion of the interface is in some sense guaranteed by the tie to an external standard or specification. However, this is no guarantee that the interface will appear consistent to metrics or code review, without understanding of this context.

Token decoupling occurs when an interface is used to decouple the rest of the systems from a class with many responsibilities – where the interface is as cluttered and un-cohesive as its source class. There is commonly only one implementation of these large interfaces, which are characterised by their diverse sets of responsibilities. In a few cases developer documentation notes that the interface has been extracted from a concrete class, with the intent to refactor the interface at some point. While there may be other smaller instances of these uses for interfaces, they are conspicuous within the largest interfaces overall.

A further notable category of use in Java applications are small enabling or grouping interfaces – these are implemented to indicate suitability for a specific 'narrow' interaction – common examples in Java applications are ActionListener and Observer which are library interfaces.

Examined systems show that 192 of 459 size one interfaces have 'listener' in their name – which might indicate that these interfaces can in some way get a *free pass* as they exist to support a specific (event driven) architectural style. However, there are 180 other interfaces with 'listener' in the name containing 2-13 methods. It is not clear how much complexity should be accommodated by such high-level design decisions before those decisions must be re-examined.

This highlights that while there may be good justification for interfaces being the way they are - such as some reason external to the design, being an intermediate refactoring step, or a common programming idiom. There is no easy way to segregate these during analysis without some extra knowledge of the system, or perhaps its evolution.

Interface size varies widely, with a 'fat-tailed' distribution. This indicates that there may be a scale-free aspect to the distribution of interface size (Potanin et al., 2005). If this is the case, then a small number of very large interfaces may be a natural *part of the landscape*. So there is general guidance to 'keep things small' (SRP, (Martin, 2005b)) and essential (Hoffman, 1990) on one hand – but evidence that large interfaces are in some sense inevitable. This raises the question of how to distinguish between interfaces where the size is *suitable for purpose*, and one where the interface is too large and should be reduced or separated. This highlights the need to tie implementation decisions to domain knowledge.

Existing metrics for interfaces factor in tangible code properties such as dependencies by subsets of clients, and the diversity of types in method signatures (Boxall and Araban, 2004). Metrics cannot account for instances where there is an external design *force* in action, which may supersede (make irrelevant) a given property of an interface. This should be considered when extracting *baseline* metrics from large corpora, and interpreting the metrics at the systems or interface level.

4.10.3 Interface Implementation

Guidance advocates preferring references to abstract types, examination of source code indicates that most of the systems examined do not implement interfaces enough to enable access to all concrete types via an abstract option. So, there is a mismatch between the implementation rate of guidance and the reality of designs as they are encountered.

The level to which non-interface types implement interfaces varies in two main ways. Many types, as noted above, do not implement interfaces. This is a distinct issue – which is set aside by 'program to an interface' studies (Abdeen et al., 2013a), which focus on the use of existing interfaces. Why are there so many concrete types that have no interfaces?

On the other hand **–** there are a few types in most systems which implement many interfaces. The highest value in the systems examined is 35 interfaces indicating not only that the implementer has many 'roles', but that it also likely has many collaborating objects which access the implementer via these interfaces. Tempero reports a "*rather extreme*" value of 56 interfaces (for *`gnu.trove.SerializarionProceedure`*) – this particular example stores methods for serialising many types – much like a visitor – with many implemented interfaces specifying the same overloaded method (Tempero et al., 2008). While outliers, these illustrate that an implementer may increase in size without necessarily becoming more complex, or changing its 'responsibility'.

Overall, small interfaces are implemented more – this may be due to the lower cost of implementing a smaller number of methods. In addition, small interfaces may capture a common interaction or cross-cutting concern at the system level – in which case more implementation would be of practical benefit. In systems with few interfaces, small interfaces persist even where no larger interfaces are defined. This may indicate that small 'non-architectural' interfaces can be useful, even in systems with little abstraction. This is another indication that there are sub-groups of interfaces which may require separate assessment.

The types which implement the most interfaces, implement many small interfaces that appear to be incidental to that implementer's main function. These smaller interfaces offer a restricted view of the implementer, while allowing the implementer to 'hook into' existing facets of a design. For example, in *argouml* there are small interfaces such as `HighLightable` and `Owned` which support implementation by offering a *vocabulary of established concepts within the system*. Use of interfaces in this way may indicate conceptual reuse where new types can be easily integrated into existing subsystems by *describing themselves* with existing interfaces. This may contribute to the observation by Tempero et al. that "*...programs become more inward looking as they age, depending on their own abstractions...*" (Tempero et al., 2008). These 'enabling' (Steimann and Mayer, 2005) interfaces are observed in many of the examined systems.

The interaction between inheritance and interfaces is complex, as it is not clear if the indirect implementation of interfaces via inheritance is deliberate or incidental for each case. However, the high amount of inherited behaviour indicates that polymorphism that might be achieved via interfaces is being accomplished via inheritance. In line with Gamma et al.'s recognition of the 'native interface' of concrete types (Gamma et al., 1994), it may be that case that super-classes (in particular abstract super-classes) provide enough abstractness without going all the way to implement an interface. If so – is there a recognisable design need that the abstract type is fulfilling, that would normally be handled by an interface?

If local (not library or third party) interfaces are considered in isolation – local interfaces appear to be implemented more often. This may indicate that there are other factors in play when non-local interfaces are considered or used, so guidelines for use of interfaces (or their absence) may also need to consider the origin of the interacting types before making a recommendation.

Considering the amount of type-type dependency captured by interfaces – the systems which appear to have been subject to intense design effort capture around 60% of non-primitive dependencies with interfaces. *JHotDraw* appears to capture a high degree of dependencies with relatively few interfaces, while *azureus* which has an unusual structure uses many more interfaces (possibly in part due to the low use of inheritance). This may indicate that there are diminishing returns on achieving higher proportions of interface use than this. In addition to considering the interfaces that are present – it may be informative to consider where interfaces are not used.

It is not clear to what extent the ability to capture many type-type interactions with a relatively low number of interfaces is a property of a design or the problem domain.

It is also worth considering the role of inheritance in enhancing apparent uptake of an interface – the pattern guidance presents the view that class inheritance and composition with interfaces are in competition. Rather than these being in opposition, the presence of inheritance enhances the number of interface implementations. Reducing concrete dependencies upon hierarchy members may mitigate some of the complications associated with introducing inheritance to a design.

In summary, interface implementation is closely related to dependency management, which is considered at (and thus has input from) all levels of design. There are indications that some motivations for interface use are 'non-local'. This may explain why abstractions are sometimes bypassed by those unfamiliar with such non-obvious or situational design motivations. As noted by a survey respondent "*I tear my hair out on a regular basis when I see components poisoned with dependencies between logical boundaries (this is something you always see with graduate programmers but never with older hands)*" (See 3.4.5.4 Survey: Coupling).

4.10.4 Interface Use in Declarations

There is uneven use of interfaces in place of other concrete types. Systems showing high use of interface-typed references favour interfaces in what may be considered more public, *protocol* areas – constructor/method parameters and method return types. This highlights the difference between simply adding more interfaces (azureus) and systems with high-impact interfaces (jhotdraw, freemind, lobo) which capture many interactions.

Use of interfaces in constructor parameters especially may indicate that interfaces are used as injected dependencies – where the constructed object does not have access to the precise implementation being supplied. Similarly, it is not enough that methods are called on abstract types – to have properly decoupled relationships, concrete dependencies should also be avoided among the parameters and return types. This appears to be the case and echoes Martin's stable dependencies principle which indicates that 'good' abstractions should depend on more stable abstractions or (preferably) have no dependencies at all (Martin, 2000).

Practitioners have the most freedom to use interfaces when they are using types that are defined in the local source code – and thus under their control. Practitioners appear to use interfaces more freely under these circumstances in most of the systems examined. This may indicate that that externally supplied types are perceived as already *abstract-enough* – or that adding an interface to third party or library types would not merit the effort involved. If this is that case, feedback from tools should be adapted to avoid prompting changes that need not or cannot be made due to higher level choices.

In summary – rather than assessing individual interfaces, it may be more useful to take a 'whole protocol' view between interacting objects – taking account of the entire interaction. This would include factors such as acquisition of references, and consider the 'least abstract' interaction between the types.

4.10.5 JHotDraw

This section discusses the notable properties of the 'design exemplar' JHotDraw[16] (JHD). This application was developed with high design quality in mind, including contributions from prominent members of the design pattern community. It has been used as an example of good design in several studies (Kegel and Steimann, 2008; Ó Cinnéide et al., 2016; Steimann, 2007), and was successfully used by Ferreira et al. to validate metric thresholds (Ferreira et al., 2012).

In this study, JHD is unremarkable in some areas, while it stands out in others. While not a large system, JHD (727 types) is comparable in size to other popular and well-used applications such as freemind (751 types). The system has an average rate of interface definition (8.8:1 class : interface ratio), which is almost exactly the median rate (8.7:1) found by Tempero et al. in their survey of a large corpus (Tempero et al., 2008). Considering system-level interface implementation, JHD again, is unremarkable in implementations per interface, and implementations per non-interface type.

JHD begins to stand out when 'abstract dependencies' are considered - 68% of its types implement two or more interfaces, it has few classes which implement no interfaces (11%), compared to the other systems. This indicates that there is the *potential* to refer to most of

[16] http://www.randelshofer.ch/oop/jhotdraw/

the concrete types in JHD via an interface. Furthermore, there is a high use of interfaces in 'public' protocols such as (method and constructor) parameters and return types indicating that – in addition to presence and use, 'placement' of interfaces may also be important.

Other work has noted that JHD "*is programmed against interfaces*" (Steimann, 2007) and has a "*rich inheritance hierarchy*" (Ó Cinnéide et al., 2016). This draws attention to the fact that JHD also exhibits the highest level of indirect interface implementation (65% of types), which is a sizable contribution to the overall rate of interface implementation. If there is substantial use of inheritance in a design, and the inheritance is used to implement interfaces, then it is likely that those interfaces will have more impact on dependencies due to indirect implementation.

This may indicate that interfaces are useful for identifying abstractions, while inheritance is more suited to the variations in implementation of that abstraction. This raises a doubt about the 'prefer interfaces over inheritance' view of design presented in some of the design literature. While there are perhaps cases where either interfaces *or* inheritance are a better choice, it appears that some combination of the two form the basis of many solutions.

4.10.6 Comparison of Findings with Interface-Related Guidance

The analysis found that overall implementation of interfaces was also lower than might be expected, especially if *program to an interface* was being pursued zealously. While some systems did achieve a high level of interfaced typed references, these appear to be the exception rather than the rule, with one being a recognised example of good design (jhotdraw), and the other having a very high number of specific interfaces (azureus).

Modularity appeared in several ways – most noticeably in larger interfaces which were often boundaries with external libraries (constant interfaces) or interfaces extracted to contain known areas of concern in a design.

Interface Segregation Principle compliance, is more difficult to detect – it is not clear what constitutes a 'good' compromise between client subsets for a given type. However, there are concrete types which implement high numbers of interfaces in the corpus – indicating segregated client groups. Though it is notable that many of these interfaces are very small and range from listener interfaces to locally defined enabler interfaces which may represent a system's internal vocabulary.

This is a further area where more context would inform – or at least preclude some forms of assessment. If an interface is imposed or supplied for implementation by the rest of the design, it is not reasonable to consider segregating that interface. This maps to the idea of a *server/client* interface proposed by Steimann and Meyer where there is some notion of an "*external service*"(Steimann and Mayer, 2005). Because this 'service' is defined outside the implementer of the client interface, the implementer has no 'ownership' of the protocol and must comply with the interface as it is. As such, it is doubtful if the ISP should be considered at this 'end' of the protocol relationship, simplifying assessment of whether the interface should be implemented.

As discussed above (4.10.2), the software design guidance offers little indication of what the properties of a system in the large should be if the guidance is followed. This is because in many cases, the guidance creates a 'pull' or demand for interfaces at individual points of need (often at the statement level), rather than prescribing a plan for the whole system.

Program to an interface encourages use of interfaces at the statement level, indeed the original patterns manual proposed several ways of hiding even object creation to allow most of a design to purely 'speak in interfaces'. Similarly, *ISP* encourages interface design to be guided based on the demands of one or more clients. Finally, *modularity* emphasises the benefits of the placement of interfaces at system boundaries, regardless of the size or characteristics of that boundary.

With these motivations in play, much of the information about a potential interface is not contained in the implementer, but the surrounding design. In addition, these demands may change as a system develops – roles may expand, groups of clients may diverge. Taking all of this into account – the definition of interfaces should be considered in light of all of the interacting types, not just the client or a specific implementation.

4.10.7 Interface-related Guidance and Interface Use Categories

The ISP and SRP promote similar ends in terms of interfaces – they should serve a single purpose. While SRP mandates a 'single reason to change' for objects – this may be rephrased as 'a single abstraction' for interfaces. In line with ISP – this single abstraction may be the abstraction required by a group of clients.

In terms of single abstractions – many of the subgroups of interfaces observed appear to represent a single abstraction:

- Marker interfaces have purely semantic meaning, arguably this cannot become bloated as there is no specified behaviour, though it may be reinterpreted by each practitioner.
- Listener interfaces have a very specific intent – the low method count in most of these interfaces indicate that there is little function creep away from event handling.
- There is indication that there are many smaller sub-groups of interfaces which have been marked by naming conventions for enabling and pattern related functions such as state, adapter, or composite (see 4.8.4.5).

However, some groups do not map clearly to a single abstraction:

- Constant interfaces may not constitute an abstraction as such, and while they do serve a single purpose, many of those observed contain multiple subgroups of unrelated constants.
- Token decoupling interfaces are by definition, neither SRP nor ISP compliant.

This hints at two different motivations for interface use:

Variation or Flexibility decoupling – where there is an intent to exploit polymorphism by programming with abstractions. This is the kind of motivation for interface use promoted by the patterns literature which exploits polymorphism to create highly adaptable, flexible solutions.

Defensive decoupling – where the primary motivation is separating good from bad parts of a design, containing change, and avoiding a concrete dependency. This motivation is captured in the ISP and 'depend towards stability' promoted by Martin and Parnas.

Again, these approaches highlight that different assessment criteria may be required depending on the motivation for the interface. For example, the usefulness of a flexibility-inspired interface may be measured by how well it matches the needs of its clients, and if it is reusable. While it may be more relevant to determine if a defensive interface has been bypassed, or if it leaks information such as implementation details.

4.10.8 How can interface use be assessed?

Interfaces are used to meet many design needs – this means that there are likely to be general quality concerns for interfaces, and specific concerns for each different mode of use. If these categories can be identified, the various means of assessing interfaces can be ranked by relative importance.

Marker interfaces contain no method signatures or constants and are often used as 'flags' to indicate object capabilities or to act as the root of a hierarchy. While the use of these interfaces may be controversial from a design point of view, once they appear in a system, this gives some indication that they are considered useful by the developer. The lack of content means that the only assessment that can be meaningfully made of a marker interface is the location of the definition of the marker interface in the source code and whether the name of the marker interface is appropriate.

Marker interfaces are used to avoid references of Object type – notionally, this provides a guarantee that the implementing object is 'reasonable'. However, a marker provides little more information about the object which implements it. Further operations on the 'marked' object require type checking. The conventional methods to avoid this would be to promote the required behaviour (methods) to the marker interface, thus avoiding the need to switch on type, or move the varying behaviour to the implementations to avoid the *need* to check for the raw type. Another 'smelly' solution would be to provide some programmatic 'flag' or 'memo' to indicate the type to eliminate the cost of the *instanceof* check.

In summary – marker interfaces provide a small amount of safety, but do not reduce the main source of coupling which is the type-checking code which must disambiguate and manipulate the 'marked' objects.

Listener interfaces are very common in OO applications as many adopt an event driven paradigm. Listeners are implemented so that the implementer can *subscribe* to some source of events. In their usual form, these interfaces extend one of the Java library ActionListener interfaces or one of its derivatives, detailing only the inherited methods. In most cases, the implementer is required to implement this interface to access a service, so there may be little choice in implementation if the system is using an event-driven architecture. It may be more appropriate to consider the entire event-driven architecture to address quality concerns.

Constant interfaces are used to store various constant values. Much like marker interfaces, these may provoke discussion of whether these are a suitable OO construct. Constant interfaces do not address any of the concerns of conventional interfaces – there is no value in implementing them as they provide data statically, they contain no executable code to speak of, in fact if they are used to share constants by implementation, they may never be invoked at all. Assessment is limited to suitability of interface name and whether the individual static constant values belong in the same constant interface.

Token Decoupling interfaces are found where there has been some effort to decouple parts of a system – often a 'god class' with an interface which has the same low-cohesion interface as its sole implementer. This interface may have many disparate sets of functionalities as it was *born* from its implementer. The priority for this interface is decoupling – so bypassing should be detected to ensure that the 'walled off' part of the system is in fact contained behind the interface.

Architectural interfaces represent system boundaries and components and as such may exist *despite* local needs. Again, the primary focus here is decoupling, however an interface such as a module boundary may offer multiple related services, or aggregate module functionality (for example through a facade) – so decoupling may trump even cohesion.

Abstract Data Type interfaces from the Java collections framework are present in large numbers. These are used in a manner which indicate that interface-typed references are preferred for data-types especially in protocol areas such as method return types. However, concrete references remain common for some data type such as *ArrayList* or *Vector*, indicating different uses of data types (4.8.3.7).

4.10.9 Motivation for Interface Use

Interfaces provide abstraction or stability in a design based on the teaching material and guidance. The main motivations for interface use are observed to fall into high-level categories, which might be described as:

- Architecture – represents a design abstraction (e.g. Model, API)
- Domain element – represents some domain or model concept (e.g. Salaried)
- Module boundary – prevent direct access to a type or module (e.g. Database)
- Perspective – provides a restricted or partial view of implementer (e.g. ImmutableList)
- Enabling – represents that the implementer has some narrow ability (e.g. Sortable)
- Role:

- Client – optionally implement this interface to access a service (e.g. Visitor)
- Specified Sub-Component – required by another type to function (e.g. State)

Several of these may apply to a single interface – for example an interface in an API may be a module boundary, and represent an element from the problem domain. However, in satisfying these different uses, interfaces may be assessed by different criteria.

Architectural interfaces describe the 'overall shape' of a system. Mapping interfaces to domain elements may lend these interfaces high conceptual cohesion, but the specifics of the interface may change as more is learned about the domain. Module boundaries exist to 'keep things apart'. Perspective (or 'partial') interfaces are the result of application of the interface segregation principle (ISP), and are derived from the implementer to restrict access, for example, to read-only methods. Role has been divided by a subtle but important difference – whether the interface is a local requirement or a 'published' means of interaction. Specified clients are often 'enabling' interfaces where these are implemented (at the option of the client) to access a service or sub-system. Specified components are required (locally/internally) by a client to function such as a state or strategy pattern object – which is less likely to be useful to the rest of the system.

These appear to illustrate different kinds of stability - some interfaces offer a specific (static) set of behaviour and will not change under any circumstances, possibly leading to multiple versions of the same interface e.g. an API. While other interfaces exist to separate elements, so may have to change *in the small* to reflect changes in the elements being separated, while remaining 'immobile' between client and implementer e.g. internal module interface.

There are different sources of motivation for interface use, which emerge at different levels of a design, these formed the basis of the interface use model defined by Mayer et al. (Steimann and Mayer, 2005). This models the cost and benefit to each of the objects involved – however it is difficult to reconstruct solely from the source code.

4.11 Conclusion

The key contributions of this work include identification of high level trends of interface definition, implementation and usage: typically 4-10% of user-defined types are interfaces; many of which are only implemented once (39%), or not at all (9%), a small number are implemented many times; many user defined types directly implement no interfaces and the majority of the remainder one interface; the marked use of small (0-2 method) interfaces and the common occurrence of extremely large (100+ method) interfaces, some of which are used many times. Using these analyses to profile the individual systems shows a wide variety of interface usage and apparent motivation e.g. systems with high amounts of interface definition and usage but very limited implementation variation behind those interfaces.

This study has shown that useful insights can be gained from a high-level quantitative analysis of interface usage in large-scale software systems. As well as gaining common insights into the amount of interface definition, usage, implementation and size it also found that interfaces are being used in different ways, not all in keeping with the design motivation behind the guideline *program to an interface*. The findings of this study indicate that this guidance is applicable at the 'statement level' which can manifest in different ways at the system level.

Mechanically extracting interfaces does not appear to be sufficient to ensure good abstractions. The practitioner may have to consider current and possible clients of an existing or proposed interface, the burden on the implementer(s), interactions with existing patterns or abstractions, and architectural decisions that are in force. It appears that some domains or solutions may have more opportunities to define 'high value' interfaces which capture many interactions. Finally, there is the tolerance of the individual, team, and reviewer preference for 'how much' interface use is expected or acceptable. Creating interfaces is potentially very complex, requiring information from different levels of the design process, some of which may not be obvious or easily accessible.

This work has identified common and distinctive patterns in the way that individual systems use interfaces: some only making little use, some using interfaces in keeping with the expected practice of implementation variation hidden behind relatively small interfaces and others having limited implementation of much-used and very large interfaces.

Again, these approaches highlight that different assessment criteria may be required depending on the motivation for the interface. For example, the usefulness of a flexibility-inspired interface may be judged by how well it matches the needs of its clients, and if it is reusable. While it may be more relevant to determine if a defensive interface has been bypassed, or if it leaks information or implementation details.

To gain a full insight further, more detailed research is required. Inheritance should be considered in conjunction with interface use, partly because interface implementation is often achieved indirectly via inheritance, but also because it appears that there is a large amount of inheritance being used in these systems.

4.11.1 Lessons Learnt

Framing of experiments to remove non-local interfaces does seem justified in terms of difference in use levels. But it might be worth considering whether the same motivation for non-use of interfaces persists in some way into the 'choice' areas. Third party and library types are easily detectable, however excluding them without due consideration may ignore secondary effects on a design.

The recent increase in 'connection based' views of software designs (Baldwin et al., 2014; Potanin et al., 2005; Schocken, 2012; Taube-schock et al., 2011) where there is a focus on dependencies and interaction, rather than the properties of individual types may yield more insights into the questions raised by this research. While some metrics do consider the clients or implementations of interfaces, there is not yet a holistic way of assessing the cost/benefit of interface use at all places interfaces might be used. The volume of data generated by even the small corpus in this study highlighted the need for more automated analysis support.

4.11.2 Limitations

This study was conducted on a relatively small corpus of open source object-oriented system written in Java. As with any study of this nature, the work would benefit greatly from replication in another language or using larger industrial systems. However, given the popularity of OO languages, and the similarities in language constructs there may be some useful insights in the findings of this work which can inform the wider body of OO design guidance.

4.11.3 Impact

The guidance regarding OO design is the matter of much debate – strong opinions are expressed about methods which affect design – agile, test driven development (TDD), design patterns; metrics are proposed, and debunked; code 'advisor' tools come and go, while core concepts still lack concrete definitions.

One issue in these debates is that the examples are often very general, or lack sufficient constraints to allow a useful conclusion to be reached. This work looks at specific uses of specific design elements and how these relate to the high-level language-agnostic guidance.

The designs discussed here reflects real-world use of design elements which have not been constructed to demonstrate a specific point or flaw. It is hoped that this work highlights the complexities of interpreting guidance ‘downwards’ to specific problems, and the compromises that are made ‘in the wild’.

5 Properties of Inheritance Hierarchies

It was found in Chapter 3 that the popular object-oriented design guidance is perceived as useful and relevant by practitioners. Chapter 4 focused specifically on the guideline 'Program to an interface' and how interfaces are used in object-oriented systems. Examination of interface use highlighted a close relationship between interface use and inheritance. However, 'correct use' of inheritance remains contentious and guidance, while plentiful, is incomplete.

This chapter examines the use of inheritance in open source systems with two main aims. Firstly, to build up a more detailed picture of inheritance use than previous work. Second, a corpus of inheritance hierarchies 'in the wild' are examined to determine if it compliance with guidance is detectable. This also highlights hierarchy properties which do not fall under popular guidelines – indicating a need to expand existing guidance.

5.1 Introduction

Inheritance in Java allows the definition of one type in terms of another. In practice, using inheritance to define a type has two inseparable effects:

- Type inheritance - the new type is considered a subtype of the parent type. Simultaneously, the extended type is a super-type of the new type.
- Module accumulation - the subtype inherits some of the substance of the super-type such as attributes and method definitions. Some of the inherited material may already be second hand, inherited by the super-type from *its* ancestors.

A difficulty is caused when the intended meaning of inheritance (if planned at all) is not explicitly encoded in source code for a design element, any predicted use (and therefore any extension) may be inconsistent, so the eventual use may be in an entirely different context to the original intention or predicted use. Furthermore, due to the variety of motivations for use of inheritance, it may be impossible to infer the original intent.

Upon encountering an inheritance structure, a practitioner must deal with many possible complex interactions such as self-calls, dependency inversion, and overriding. Examination of many types may be required to understand potential side-effects of any maintenance or extension activities. This is to say nothing of assessing the potential usefulness of the inheritance hierarchy or any of its members.

Like any powerful technique – inheritance provokes discussion between what *can* be done and what *ought* to be done. Practitioners have both modelling and operational needs during the design of software systems and inheritance can be used to solve these problems in various ways. For example, Meyer identifies thirteen useful (and three discouraged) uses for inheritance describing varying levels of rarity and conceptual difficulty (Meyer, 1996).

Inheritance structures intersect with low level concerns such as object composition and indirection, to higher level concerns such as pattern implementation and cross package dependency management. This means that there is a lot of potentially applicable or conflicting guidance which may apply to various aspects of an inheritance use.

There has been a great deal of varied advice and debate on the proper use of inheritance **–** this research aims to contribute to the growing body of empirical work on how inheritance is used 'in the wild' (in open source systems) and reflects on how the findings may close the gap between guidance and practice.

In addition, the use of inheritance is examined in more detail than in previous studies and an effort is made to characterise and explain the observed inheritance, paying more attention to hierarchy structure and local context.

Inheritance is an important feature of many programming languages, allowing practitioners to define new program elements by building on what already exists and reducing duplication in source code.

Design guidelines are high-level and conflict in some cases – little empirical work aims to inform practitioner decision-making at the same level as design guidelines. While simple in concept, inheritance appears in a variety of forms between languages, often with differing mechanisms for multiple inheritance, and method overriding.

This study investigates the extent to which inheritance use be characterised - with a view to informing design choices and objectively improving design quality. Thus, closing the gap between practice and guidance.

Java is a popular programming language both in industry and teaching environments. It is the subject language in much of the empirical software engineering literature, and well-studied corpora of Java systems are available facilitating comparison with other work. In addition, the popularity of Java means that it is the subject of many design textbooks and online debate – which has resulted in an extensive body of guidance. Finally, similarities between Java and other industrial languages such as C# mean that this work will be amenable to replication in the wider software ecosystem.

5.2 Research Objectives

Analyse: Inheritance use in open source systems, specifically inheritance hierarchies, their members, and the interactions between these elements and the surrounding system.

Purpose: Explore and characterise the presence and use of inheritance use with respect to the intentions raised in the survey (Chapter 3) and compliance with design guidelines.

The *point of view* is primarily that of the empirical software researcher, although insights gained may be of interest to practitioners, and users and designers of software metrics.

The *context* is object-oriented (OO) design literature and OO design guidelines.

Research Goal: How is inheritance used in practice and to what extent can its usage be related to OO design guidance?

This goal has been further refined into Research Questions:

1. How is inheritance used?
2. How can inheritance hierarchies be characterised?
3. To what extent can inheritance usage patterns be related to design quality where quality is defined via Guidance, Metrics, Code Smells, Entity Population Models, and Modelling?

5.2.1 Context

This study uses a corpus of open source Java systems assembled from systems in the Qualitas Corpus (Tempero et al., 2010b) and additional systems of interest which are also available from various open source repositories (see Table 30). Analysis was carried out with a counting tool based on the Eclipse JDT Core[17]. While this tool is novel, the core components are very reliable as they are sourced from the Eclipse Project, a well-known and widely used integrated development environment (IDE).

The systems examined are of varying size and problem domain. Six were sourced from the evolution release of the Qualitas Corpus as these systems represent mature systems with a

[17] http://www.eclipse.org/jdt/core/

robust version history and user base. A further seven systems were selected from the non-evolution Qualitas Corpus to provide a variety of problem domains. A student-generated system (*gizmoball*) was also added to the corpus as an example of 'design as taught'. Some of the systems in the corpus have already been discussed in other work, namely *eclipse*, *jhotdraw,* and *jdk* (Cai et al., 2011; Dyer et al., 2014; Kegel and Steimann, 2008; Rocha and Valente, 2011; Tempero et al., 2010b).

While the corpus analysed here is smaller than in other recent studies – high-level similarities with other corpora are shown.

Furthermore, while this study attempts to characterise inheritance use, there is no intention to establish an *entity population model* (characteristic normal ranges) for all other software systems as was the goal in previous work. In this case, inheritance use is characterised with a view that the entire corpus of software systems will be at least as diverse as those studied here.

5.3 Analysis Technology

While the structure of Java code remains relatively stable over versions of the language, there are diverse ways to target a corpus of Java programs for empirical study.

Byte-code analysis is perhaps the most accessible as most software is deployed as compiled *jar* files. Analysis of byte code has been used in popular high level code surveys (Baxter et al., 2006; Brekelmans, 2014; Collberg et al., 2007; Tempero et al., 2013, 2010a, 2008). These studies use tools based on various open source analysis frameworks.

While analysis of byte code does not present any inherent validity threats, it must be suitable for the experiment - as some of this work acknowledges, "*some aspects of the original code are lost*" (Collberg et al., 2007), such as through optimisation or deliberate obfuscation (Brekelmans, 2014). In addition, it can be difficult to reconstruct which specific element of source code has been converted to a given byte code. This is important as it was anticipated that some manual inspection of source code would be required to investigate/validate some of the findings in this study (Stevenson and Wood, 2018).

Since this study is concerned with the structural elements of the design, and practitioner decisions, it was decided that source-code analysis was more suitable in this case. Studying original source code has the advantage of "*maintaining full integrity of semantics and intent*" (Brekelmans, 2014).

5.3.1 Instrumentation

Data was collected using a tool built based on the Java Development Tools (JDT) Core component of the Eclipse project. It was noted that other studies have used analysis frameworks (e.g. MOOSE), which are large and inevitably carry with them the assumptions and requirements of their primary users. In addition, initial attempts to use the more heavyweight modelling frameworks revealed the time and (computing) resources required may be prohibitive for the analysis of large systems.

Indeed, in more recent studies of large corpora, it is stated that smaller tools have been constructed or that resources limited the extent of the analysis in some cases e.g. Tempero et al.'s study on inheritance use in Java "*had memory limitations that restricted the size of the systems that we could analyse*" (Tempero et al., 2013).

The influential and rigorous studies by Tempero use bytecode analysis using a purpose-built analysis tool (Tempero et al., 2008). The tools Byte Code Engineering Library (BCEL) and *javap* (a java byte code disassembler) have been used to study bytecode (Baxter et al., 2006). A further study examines bytecode using a tool "*based on the Soot Framework*" (Tempero et al., 2013).

Kegel uses source code analysis and abstract syntax tree (AST) manipulation using a tool of their own devising (Kegel and Steimann, 2008). Other tools used in similar studies include Codecrawler (visualisation) (Lanza et al., 2006) and MOOSE (re-engineering) (Girba et al., 2005). These in turn are based on abstract meta-models such as FAMIX (Lanza, 2003).

Finally, Sabané estimates that the time and effort required to understand and adapt (possibly unsuitable and usually out of date) existing research tools is too high and developed a new tool based on an existing modelling framework to detect fragile base-class structures (Sabané et al., 2016).

These tools are well-regarded in the area and have been validated over many studies – however, it was not clear from the outset if the tools available would be sufficient for what was initially a very exploratory study. It was also apparent that the assumptions of the tool-makers may influence the direction of the study and that documentation is often an afterthought. Given the learning curve and resource requirements for the use of these tools, and the potential for mismatch with the experimental requirements, it was decided that one or more small tools specific to the study would be more suitable.

Add to these considerations the exploratory nature of this project – the decision was made to produce a minimal tool that would avoid model building where possible by making multiple passes over the sample source code. This produced several benefits:

- The initial models build by the JDT core component only need to be built to retrieve primary data, subsequent analysis components depend only upon the gathered data sets.
- Sub-sets of the target corpus systems can be expanded on demand, such as only building and resolving ASTs for inheritance hierarchies.
- The memory requirements for the tool are relatively modest as it accumulates a large *flat* (file based) data store, which can be used as a backing store for subsequent analysis modules.
- Each analysis is recorded in code or script, so experiments are explicitly documented to promote reuse, validation, and replication.

Figure 30 shows a high-level overview of the tool used for data gathering and analysis. In the interests of reproducibility and ease of tracing what processing was done, each sub-experiment has been archived in its own package.

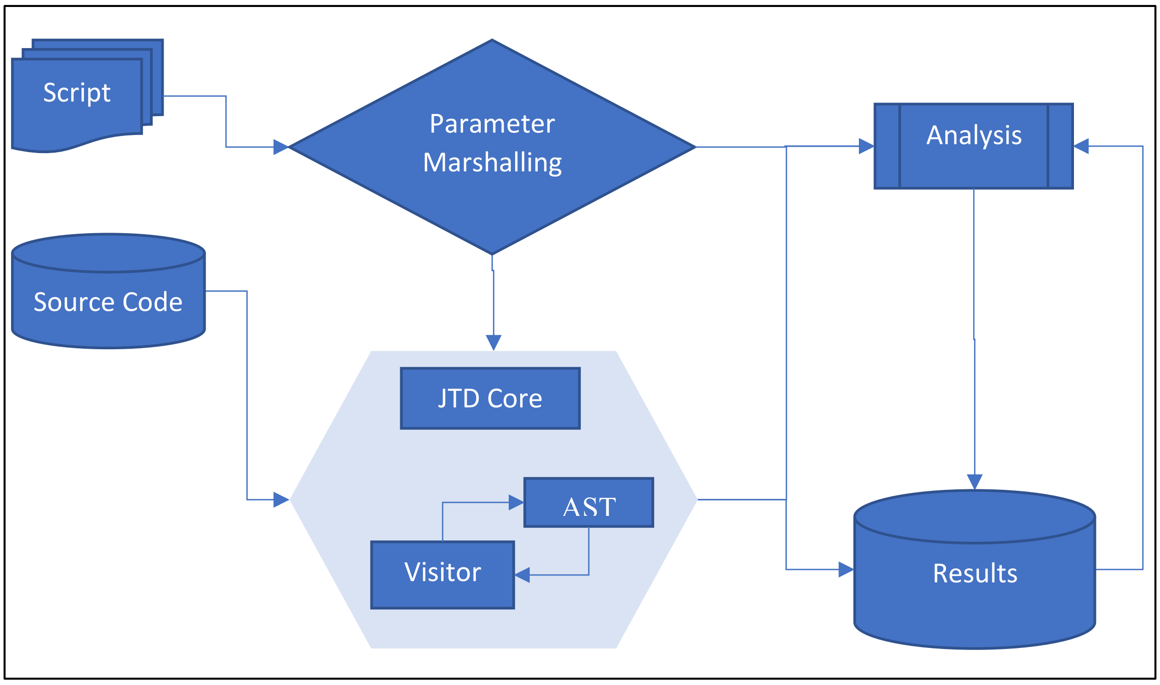


Figure 30 – Inheritance Analysis Tool

The *Script* is a python program which serves to record the exact parameters of the experiment including the details of the target source code, and location of output data.

Source Code is in Qualitas Corpus format. *Parameter Marshalling* extracts and distributes input parameters, and determines the type of experiment, triggering data extraction from source code if instructed. *Analysis* performs second stage data extraction which relies on direct or indirect data from the source code. *Results* is a cache for 'raw' data and calculated results, initialisation is 'lazy' which means that cached data is preferred, and new data is only calculated if it is required by a request from *Analysis*. *AST* is one or more Java abstract syntax trees which are generated by the *JDT Core*. *Visitor* represents one or more visitors uses to traverse *JDTCore* structures. Note that lazy initialisation can result in a call-back to *Analysis* from *Results*, however in these experiments the more primitive analysis would have already been calculated as part of an earlier query. In a more general setting, care would have to be taken to avoid circular dependencies between derived calculations which could lead to deadlock.

Once an experiment has been run, the package responsible is not modified thereafter, providing a permanent record of the experimental process and environmental variables. The main effects this had on the design of the tool were:

Redundancy/duplication – some experimental elements are very similar and might be refactored out in a conventional software design. However, since this duplication constitutes a permanent record, there is no need to keep all the duplicates in sync – which is the main issue with duplicate code.

This could have been achieved by looking back with version-control, however future accessibility is improved if all analyses are available in the same place. Similarly, the configuration parameters of each tool execution were captured as Python scripts, which can be re-run to exactly reproduce the experiments.

Intermediate results – the design makes heavy use of intermediate plain-text files. These are used as *pipes* between the various data gathering and analysis modules of the tool. These allow verification at each stage of the analysis, and allow more complex analysis to be done without re-constructing fresh models each time. This also allows (time) expensive initial data extractions to be run only once.

5.4 Experimental Design

This section describes all the information that is necessary to repeat the study.

5.4.1 Goals, Hypothesis, Variables

In this section, the research questions are refined down into a related set of sub-questions.

1. How is inheritance used?
   a) How much inheritance is used in software systems?
   b) How is inheritance used to define types?
2. How can inheritance hierarchies be characterised?
   a) How representative of hierarchy members is the root type?
   b) What are the overall shape characteristics of hierarchies?
   c) Can casting tell us anything about inheritance hierarchies?
   d) Can any insight be gained from method invocation patterns?
   e) How are abstract types used in inheritance hierarchies?
3. To what extent can inheritance usage patterns be related to Design Quality where quality is defined via Guidance, Metrics, Code Smells, Entity Population Models, and Modelling?
   a) Is it clear where inheritance or composition should be used?
   b) To what extent is compliance with guidelines evident in source code?
   c) Are any of the models of inheritance detectable?
   d) How well do the proposed models of inheritance reflect current practice?

5.4.2 Study Design

This is a case study of inheritance in open source java systems, with a view to gaining greater understanding of what kinds of inheritance structures are present in production code. With the exception of two special systems of interest (jhotdraw and gizmoball) detailed in the next section, the corpus was chosen to be of mature systems and/or from diverse domains to avoid bias from a particular domain.

All candidate systems in this study have been sourced from the Qualitas Corpus (QC) or other open source repositories and prepared into the same format as the unpacked QC systems. It was found that systems often contained compartmentalised source code or unusual package structures – even containing *unpacked* library code among the source code files in some cases. This presents a similar issue noted by Brekelmans, where C# binaries can be merged for distribution (Brekelmans, 2014).

Rather than attempt to separate code from disparate sources, it was decided that since all the code in a repository is notionally visible to a practitioner (usually via an IDE of some description), it does fall within the category of *practitioner facing* inheritance.

Similarly – while many studies exclude types from the standard Java libraries, the java libraries have been built into this study. This allows comparison of different hierarchies with the same library-root within and between systems.

In terms of the data required, the goal of this study was to gain insight into inheritance 'as it is'. Thus it was decided that pre-existing metrics (such as the popular C&K suite (Chidamber and Kemerer, 1994)) might obscure features of the population of inheritance hierarchies due to the function of metrics as 'opinionated' detectors for specific phenomena or features. This study attempts to focus on the *natural features* of inheritance structures that might be identified by inspection or practitioner familiarity.

5.4.3 Subjects

The subject systems used in this study are detailed in Table 30. It was anticipated that the analysis of each system may take some time, given the focus on context and structure over volume. Additionally, large scale surveys (of hundreds of systems) have already been replicated on the QC, using both source code and byte code analysis. The research goal of this project was more focused on the finer details of each system. Consequently, four additional systems were chosen to investigate if the projects had any design properties relating to this study:

- eclipse – this IDE has been the subject of evolution studies on its own and has undergone several major refactorings. The system is based on an actively developed framework – so it may be expected that this is an example of a *good working design*.
- gizmoball – this is a small game set as a design challenge by MIT (Yue et al., 2004) and used in many undergraduate CS programs. This version was chosen as it was submitted by a group of students displaying a good grasp of undergraduate design principles – and so is an example of *guidelines as taught.*
- JDK – the java development kit is the core library of the Java language, containing many day-to-day library functions. This is an example of Java *as it is spoken* by its designers.

- jhotdraw – was created as an *exemplar of good pattern usage* by an author of the original design patterns manual (Gamma et al., 1994), and is recognised as such in other empirical work (Ferreira et al., 2012).

JHotDraw (and to a lesser extent gizmoball) were included in this study to provide a comparison between the more general systems and education-facing systems used to illustrate *good design* to students. It is hoped that these would highlight any major differences in inheritance use between these approaches to system construction.

Radjenović et al. discovered size ranges for data-sets from a review of code-survey work ranging from small, medium , or large (<=200 |200-1000 |> 1000 classes respectively) (Radjenović et al., 2013). The systems selected here are representative of these ranges, and in aggregate place the study well into the large category.

The systems shown in Table 30 which also appear in the previous chapter have slight differences in the type counts compared to those shown for the corpus shown in Chapter 4 (Table 18).

Table 30 – Inheritance Study Corpus Details

Name	**Version**	**Types**	**Inheritance Hierarchy Members**	**Domain**	**Source**
ant	1.8.4	1203	605	parser, generator, make	Qualitas Corpus Version: 20130901e (Tempero et al., 2010b)
argouml	0.34	1972	1522	diagramming/ visualisation	
azureus	4.8.1.2	3319	1040	torrent client, database	
freecol	0.10.7	727	617	game	
freemind	0.9.0	445	343	diagramming/ visualisation	
eclipse SDK	4.3	22636	13053	IDE	
aoi	2.8.1	493	328	3D, graphics, media	Qualitas Corpus Version: 20130901
axion	1.0 M2	237	148	database	
columba	1.0	1181	883	email client	
freecs	1.3.2010 0406	139	80	chat server	
galleon	2.3.0	258	152	3D, graphics, media	
JDK	8u60	7659	3862	language library	
JHotDraw (JHD)	7.0.6[18]	310	237	graphics framework (design exemplar)	
gizmoball	-	84	43	game (student design project)	www.github.com/ tomcurran/gizmoball

[18] Note that the version 7.6 of *jhotdraw* is used in Chapter 4

This study is focused on the gross properties of 'conventional' inheritance hierarchies, so niche uses such as inner classes, inner interface, and enumerated types are excluded, thus the counts shown in Table 30 are lower than for the same systems in Chapter 4. The fourth column shows the total number of classes and abstract classes in the inheritance hierarchies contained in each system – note that this count includes third party and library types.

5.4.4 Objects

The objects of study in these experiments are **system**, **hierarchy**, and **hierarchy member**.

System – discreet applications sourced from the QC or another open source repository. Since source code is often organised in unusual ways, the primary targets of the tools used are source-code containing directories within the source files supplied. However, to properly resolve types, the entire source is used to locate types.

For example, if source folders contain an unpacked (and possibly modified) third party library, the third-party types will be resolved as super-types in the source code. Similarly, the resolver is aware of the types in the standard Java libraries and can resolve the types in a hierarchy discovered where a user-defined file extends a Java library type. Note that the origin of a type is tagged as Local, Java, or Third Party based on its source namespace.

Due to the nature of some systems, resolving Third Party types is done on a best effort basis. If source code contains a type that extends a missing third-party library type, the possibility that the absent super-type is a Java library type, or a local type can be eliminated. This leaves only the possibility that the unknown type is Third Party. While it is possible to retrieve the name of the missing third-party type that is extended from the type declaration of its subtype, it is impossible to know if there are any further members in that third-party hierarchy.

Hierarchy – a hierarchy is a directed acyclic graph of concrete or abstract classes which may be defined in the source code of an application, part of the Java standard libraries, or imported from a third-party library. The hierarchy members in a hierarchy are arranged such that there are no unconnected types. The member with no ancestors is known as the *root*. The relationships within a hierarchy are often described as *subtype : super-type* or *parent : child*. Noting that Java does not allow multiple inheritance – each subtype can only have one immediate (non-interface) super-type.

Hierarchy Member – an abstract class or concrete class which is a member of an inheritance hierarchy. To qualify as a member, the type must extend or be extended by another type. Note that this definition refers to the inheritance mechanisms in Java (extends, implements) and does not imply or require any semantic relationship between the types involved. Interface implementation by hierarchy members is recorded, but interfaces themselves are not considered to be hierarchy members in this study. Since all objects in Java inherit 'Object' as an implicit super-type, Object is ignored in this analysis.

5.4.5 Data Collection Procedures

Experiments were run as part of the tool development process, with each develop-validate-analysis cycle yielding more results.

Environment: Most of the analysis was run on an Intel Core 2 Duo (3.16 GHz) machine with 6GB of RAM, running a 64-bit version of Windows 7 Enterprise. Initial data gathering using the JDT Core required more memory for larger systems (*eclipse*, *JDK*) and was run on an Intel i5 (3.4 GHz) machine with 16GB of RAM, running a 64-bit version of Windows 7 Enterprise.

5.4.5.1 Data Set Reduction

While no outliers were removed, there are a few instances where a missing library, generated code, or unconventional syntax meant that the type of a variable or formal parameter was not resolved. In these cases, the number of unresolved types is negligible, but is listed in the figures where appropriate.

5.4.6 Analysis Procedure

This study follows in the footsteps of the more recent *census* style studies where reporting the corpus qualities is done without much post-processing. The intention is to describe what is observed, rather than validate a metric or find a correlation.

While an effort was made to eliminate testing code from the corpus, the development styles of different teams mean that some testing code may be integrated with the main code base, or that some public method functionality may be for the sole purpose of testing. Similarly, code written for use with injection frameworks may have a different structure from the same design that does not such a facility. However, these phenomena are a result of current practice, so must be considered in an analysis of programming practices.

During analysis, some corpus properties are segmented by *natural breaks* in the observed ranges, for example, the first 80% and last 1% of long-tailed distributions to properly represent outliers. In either case, these are explained near the relevant charts.

5.4.7 Evaluation of Validity

This section discusses the steps taken to increase the reliability of measurements – ensuring complete and correct collection of data.

Operational – As presented, the meaning of the various counts is taken directly from what is counted, these are highly representative. The second stage analysis involves comparing these more direct measures with each other. While no new measurements are named, it is reasonable to question if these comparisons are appropriate – this issue is addressed near the comparisons where appropriate.

Additionally, some of the scalar values e.g. depth of inheritance, are occasionally grouped into ranges for clarity of presentation. These ranges are based on the distribution of the observed population of inheritance hierarchies. In these cases, any noted trends or findings have been checked on the unabbreviated charts to ensure the ranges do not exaggerate or misrepresent the findings.

Internal – Creating a new tool carries with it inherent risks. At each stage in the building process (and analysis) 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 on collected data and have been used for validation.

External – An obvious issue with this study might be the size of the corpus. However, up until relatively recently, very large corpus studies were the exception rather than the rule, so corpus size on its own is not a major threat to validity depending on the claims being made.

Similarities to larger corpora in other studies are highlighted in the initial corpus summary. These similarities recur throughout the study, and appear to confirm that the smaller corpus selected here has similar general properties to larger corpora.

In addition, systems have been selected from a range of problem domains and application sizes to reduce issues with domain-specific programming styles. This study does not aim to build an *entity population model*; the generalisability of the hierarchy characteristics relies on innate characteristics of inheritance hierarchies and while it seems likely that other uses of inheritance may indeed exist, these would merely expand the discovered list of uses without necessarily invalidating the patterns of use that have already been identified.

Reliability – All the experimental methods relating to this chapter have, literally, been codified in the source code of the analysis tool and configuration scripts. While portability was not a primary goal for the tool it has been successfully used in undergraduate projects in different hardware and OS environments – the students could re-run and modify the experiments with minimal supervision. Finally – as noted in the findings, this analysis is re-applicable, but the numeric values ranges (e.g. max depth, max width) may be system specific.

While the software tool created here is specific to this study, it is based around a well-supported framework (eclipse JDT Core), and has been written in small, reusable modules which make the experimental apparatus easy to understand.

5.5 Results

5.5.1 Descriptive Statistics

The analysis tool developed for this study was applied to 14 systems, comprising of 40663 types defined in source code (range 84–22636 types). The systems studied contain 2440 inheritance hierarchies (range 6–1277 hierarchies), consisting of 22913 hierarchy members (range 2-1366 members). Hierarchy depth ranged from 2 to 11 (DIT_{MAX} 1 to 10), and hierarchy width ranged from 1 to 542.

5.5.2 Presence of Inheritance

The initial data collected were cross-system counts of how many concrete classes and abstract classes had been defined and how much inheritance has been used to define these types.

Table 31 shows the amounts and proportions of types defined by inheritance and/or interfaces in each system, which are broken down by type (concrete class, abstract class, or interface), and summarised for each system. Finally, the types are broken down by their exposure to inheritance (extension of a type, direct implementation, combinations thereof). Note that the percentage (%) figures shown are the percentages within that category for ease of comparison.

For example, in the first row, *Gizmoball* has 21 concrete classes that extend (are a sub-class of a concrete class or abstract class), which is 28% of all concrete classes in that system. Note that the first number column shows the number of types defined in the system for reference (local type definitions only, excluding library and Java types).

There is notable variation between systems, even in this small corpus there are exceptions to generalisations that could be made. For example, most interfaces are not defined using inheritance (except in freemind, JDK, jhotdraw), most concrete classes extend rather than implement interfaces (except axion, galleon, gizmoball), abstract classes are usually defined with some form of inheritance (except aoi, JDK).

The ranges of values are also notable – it is fair to say that most of the concrete classes examined are defined using some form of inheritance. There is some variation between systems in the *neither* category for concrete classes (10-29%, median 15%), but even the highest value (29%) indicates that concrete classes in all systems meet (and usually exceed) the finding by Tempero et al. that "*around three-quarters of user-defined classes use some form of inheritance*"(Tempero et al., 2008).

Abstract classes have much more variability in the *neither* category indicating various uses, for example abstract classes can be at the root of a hierarchy, or somewhere inside a hierarchy. The total of *neither + only implement* (0-100%, median 20%) indicates which abstract classes are at the top of a hierarchy (has no super-types that are not interfaces), while *only extend+both* (0-100%, median 23%) indicates which abstract classes are inside a hierarchy (the abstract class extends some other concrete class or abstract class). Across all systems abstract types are split approximately in half between being hierarchy roots (1760), and being inside a hierarchy (1648). However, there is noticeable variation between systems with extremes of both kinds – JDK notably has a high number of abstract class hierarchy roots, but many of these implement no interfaces.

Table 31 - Shows what proportion of each type defined using inheritance – Note that % shown are within each category e.g. red figures are % of all classes in that system

System	Total Types	Concrete Class				Abstract Class				Interface		Total			
		only extend	only implement	both	neither	only extend	only implement	both	neither	only extend	neither	only extend	only implement	both	neither
gizmoball	84	21	33	12	8	1	0	2	0	1	6	*22*	*34*	*14*	*14*
		28%	45%	16%	11%	33%	0%	67%	0%	14%	86%	26%	40%	17%	17%
JHotDraw (7.0.6)	310	158	35	26	28	12	8	4	2	17	20	*170*	*60*	*30*	*50*
		64%	14%	11%	11%	46%	31%	15%	8%	46%	54%	55%	19%	10%	16%
JDK (8u60)	7659	2408	1043	732	765	200	182	111	439	869	910	*2608*	*2094*	*843*	*2114*
		49%	21%	15%	15%	21%	20%	12%	47%	49%	51%	34%	27%	11%	28%
ant (1.8.4)	1203	659	110	130	153	38	10	16	8	14	65	*697*	*134*	*146*	*226*
		63%	10%	12%	15%	53%	14%	22%	11%	18%	82%	58%	11%	12%	19%
aoi (2.8.1)	493	232	62	34	104	14	2	2	14	2	27	*246*	*66*	*36*	*145*
		54%	14%	8%	24%	44%	6%	6%	44%	7%	93%	50%	13%	7%	29%
argouml (0.34)	1972	1073	235	196	163	65	27	27	16	48	121	*1138*	*310*	*223*	*301*
		64%	14%	12%	10%	48%	20%	20%	12%	28%	72%	58%	16%	11%	15%
axion (1.0 M2)	237	74	41	37	18	5	12	7	5	14	24	*79*	*67*	*44*	*47*
		44%	24%	22%	11%	17%	41%	24%	17%	37%	63%	33%	28%	19%	20%
azureus (4.8.1.2)	3319	516	728	337	640	25	56	11	30	127	849	*541*	*911*	*348*	*1519*
		23%	33%	15%	29%	20%	46%	9%	25%	13%	87%	16%	27%	10%	46%
columba (1.0)	1181	527	119	228	155	8	17	8	2	32	85	*535*	*168*	*236*	*242*
		51%	12%	22%	15%	23%	49%	23%	6%	27%	73%	45%	14%	20%	20%
eclipse_SDK (4.3)	22636	8700	2974	2034	3392	701	435	340	468	1245	2347	*9401*	*4654*	*2374*	*6207*
		51%	17%	12%	20%	36%	22%	17%	24%	35%	65%	42%	21%	10%	27%
freecol (0.10.7)	727	447	23	76	108	20	3	13	6	2	29	*467*	*28*	*89*	*143*
		68%	4%	12%	17%	48%	7%	31%	14%	6%	94%	64%	4%	12%	20%
freecs (1.3.20100406)	139	73	19	1	27	0	3	0	0	0	16	*73*	*22*	*1*	*43*
		61%	16%	1%	23%	0%	100%	0%	0%	0%	100%	53%	16%	1%	31%
freemind (0.9.0)	445	195	49	82	49	7	7	8	4	19	25	*202*	*75*	*90*	*78*
		52%	13%	22%	13%	27%	27%	31%	15%	43%	57%	45%	17%	20%	18%
galleon (2.3.0)	258	90	66	27	55	1	2	2	2	4	9	*91*	*72*	*29*	*66*
		38%	28%	11%	23%	14%	29%	29%	29%	31%	69%	35%	28%	11%	26%

Interfaces, as might be expected, are the least often defined by inheritance (0-49%, median 28%). Indicating that, while interfaces themselves are common, hierarchies of interfaces are relatively rare or small in most of the examined systems. There is least variation in the definition of classes in the *both* or *neither* categories (St Dev 0.06) – indicating that there is more variability between *only implement* and *only extend*. This may indicate different approaches regarding the use of interfaces vs inheritance.

All but one of the systems (*freecol*) define more interfaces than abstract classes. The JDK has a high percentage of abstract classes in the *neither* category, indicating that they are the root of a hierarchy with no interfaces. Systems with fewer abstract types appear to have more outliers – due to the low numbers in each category and the use of proportions to facilitate comparison. It is notable that in the medium and larger sized systems there is no consistent proportion of interfaces defined – but the smaller systems have distinctly (predictably) fewer interfaces. It is not clear if this is due to fewer opportunities to use interfaces in smaller systems, or a perceived lack of need of interfaces when a system is small.

Figure 31 shows the data from the last four columns of Table 31 in stacked proportional bars more clearly illustrate the breakdown of definition-by-inheritance use within each system.

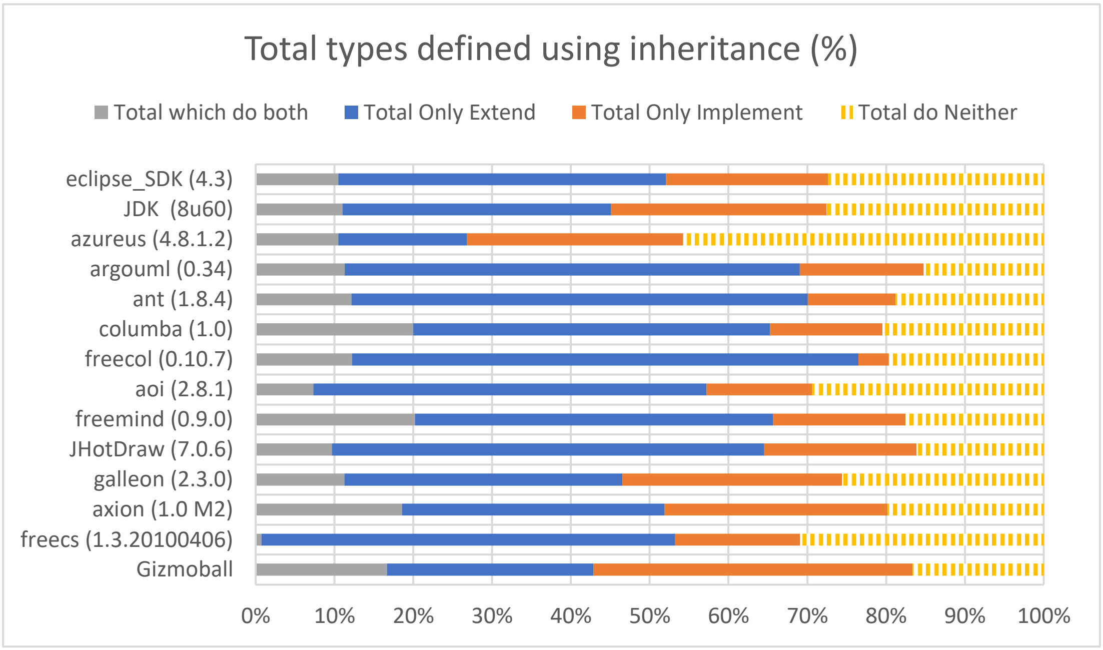


Figure 31 - Total types defined using inheritance, systems in order of size (number of types), largest system at the top.

The last category (Total do Neither) is striped to illustrate that the solid categories are all definition by inheritance in some way. Note that this chart shows the systems in size order with the larger systems represented by the topmost bars.

As noted above, Tempero et al. reported that as much as "*around three-quarters of user-defined classes use some form of inheritance*" (Tempero et al., 2008). Figure 31 shows that, unlike concrete classes, only 7 of the systems examined meet the *three quarters* figure across all their types, and only one system (azureus) falls noticeably short.

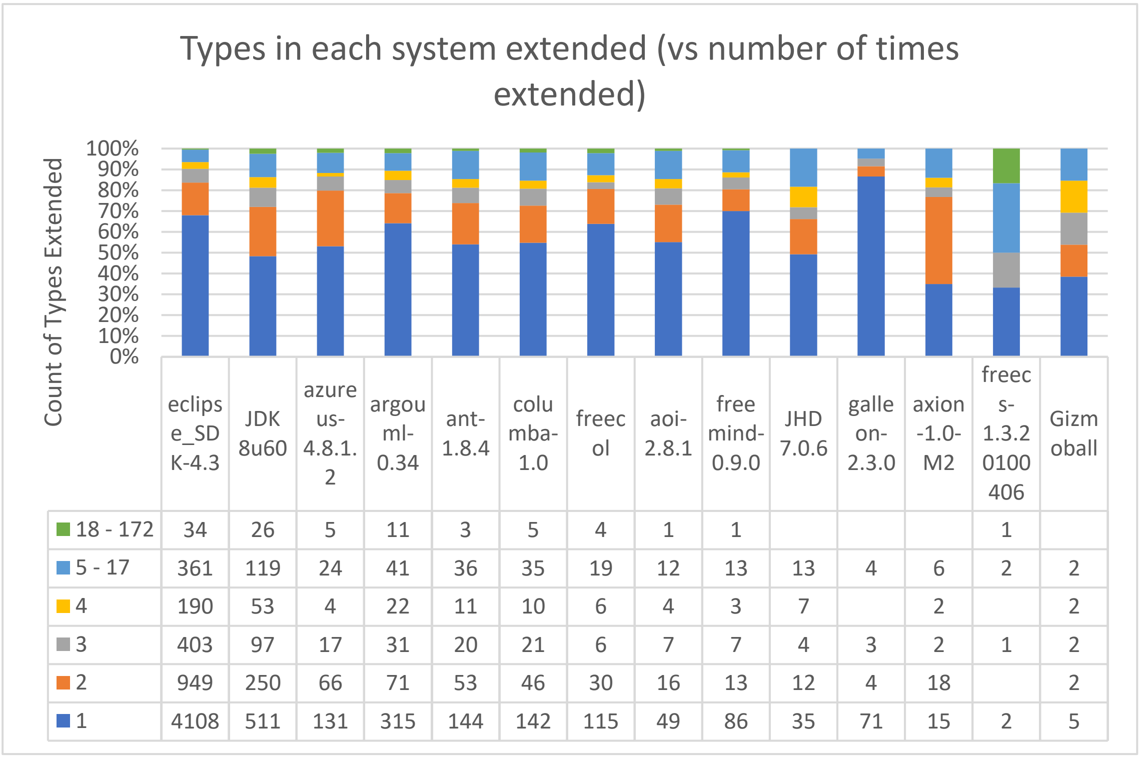
Figure 31 also shows that the medium sized systems are using (proportionally) more abstract definitions, namely more class inheritance. This may indicate a diversity of requirements in the larger systems, which permits less use of generalised abstractions. Azureus appears to have low use of inheritance, which has not 'been made up for' by the high definition of interfaces.

Figure 32 shows the extended types for each system, broken down by how many times each type has been extended. For example, the first bar shows that in *eclipse_SDK-4.3*, over 65% of the types that are extended are extended only once. The bars show the distribution and the actual counts are in the data table below the chart. The top 1% of types (18-172 extensions) and 10% of types (5-172 extensions) overall have been compressed into two ranges for legibility.

The first point of interest is the degree of uniformity among the large and medium systems (on the left of Figure 32, this contrasts with the smaller systems on the right. This indicates that some *norms* may only 'kick in' after system reach a certain size or complexity.

All observed systems have hierarchies in the top 10% range, indicating that popular abstractions are not necessarily restricted to larger systems – even with order of magnitude differences in system size. Considering the top 1%, there are several systems which do not have types that are this popular – this very high range may be empty in some systems as the number of hierarchy members would approach the system size of the smaller systems. This provides a natural limit and should perhaps be considered when assessing the significance of 'number of extensions'.

Note that, across all systems, over 80% of the extended types are extended only one (64%) or two (17%) times indicating that most individual uses of inheritance do not generally widen the containing hierarchy (i.e. NOC is low for most super-types), there are few types that see extensive direct reuse. It appears that even with one or few opportunities for reuse, extension is still used.



	eclipse_SDK-4.3	JDK 8u60	azureus-4.8.1.2	argouml-0.34	ant-1.8.4	columba-1.0	freecol	aoi-2.8.1	freemind-0.9.0	JHD 7.0.6	galleon-2.3.0	axion-1.0-M2	freecs-1.3.20100406	Gizmoball
🟩 18 - 172	34	26	5	11	3	5	4	1	1				1	
🟦 5 - 17	361	119	24	41	36	35	19	12	13	13	4	6	2	2
🟨 4	190	53	4	22	11	10	6	4	3	7		2		2
⬜ 3	403	97	17	31	20	21	6	7	7	4	3	2	1	2
🟧 2	949	250	66	71	53	46	30	16	13	12	4	18		2
🟦 1	4108	511	131	315	144	142	115	49	86	35	71	15	2	5

Figure 32 - Types extended in each system, broken down by number of times extended (order from largest system on left to smallest system on the right).

Some guidance indicates that super-types should be extracted after some number of examples with common code have appeared (commonly three similar types (Beck, 2014)), however this does not explain 81% of the extended types with one or two subtypes. This may indicate that much of the observed inheritance is a result of extension from an existing type, rather than extraction of a super-type. This would concur with the lack of available refactoring time noted by respondents in Chapter 3, however these different scenarios may present different concerns and trade-offs to the practitioner.

Summary of Presence of Inheritance: [KF 5.1] Inheritance is present in high amounts, meeting or exceeding levels found in related work, and is thus important in software design. [KF 5.2] If class and interface inheritance are included, levels are like those found in the literature. [KF 5.3] Types that are extended are only extended once or twice in most cases.

[KF 5.4] Some similarities may only appear as system grow, medium and large systems show similar distributions of type-extension.

5.5.3 Hierarchy Properties

There are a range of notions of magnitude or size relating to inheritance hierarchies. Figure 33 shows the number of hierarchy members across the hierarchies examined. These have been sub-divided based on their proportion of the total number of hierarchy members. For example, the bottom 80% of hierarchy members are in hierarchies with 2-7 members. Similarly, the top 1% of hierarchies range in membership size from 177-1366 hierarchy members.

Figure 33 displays a 'long tail' distribution (Tempero et al., 2010a). In a survey of software properties, Baxter et al. (Baxter et al., 2006) suggested that such distributions are evidence that practitioners are less aware of incoming dependencies.

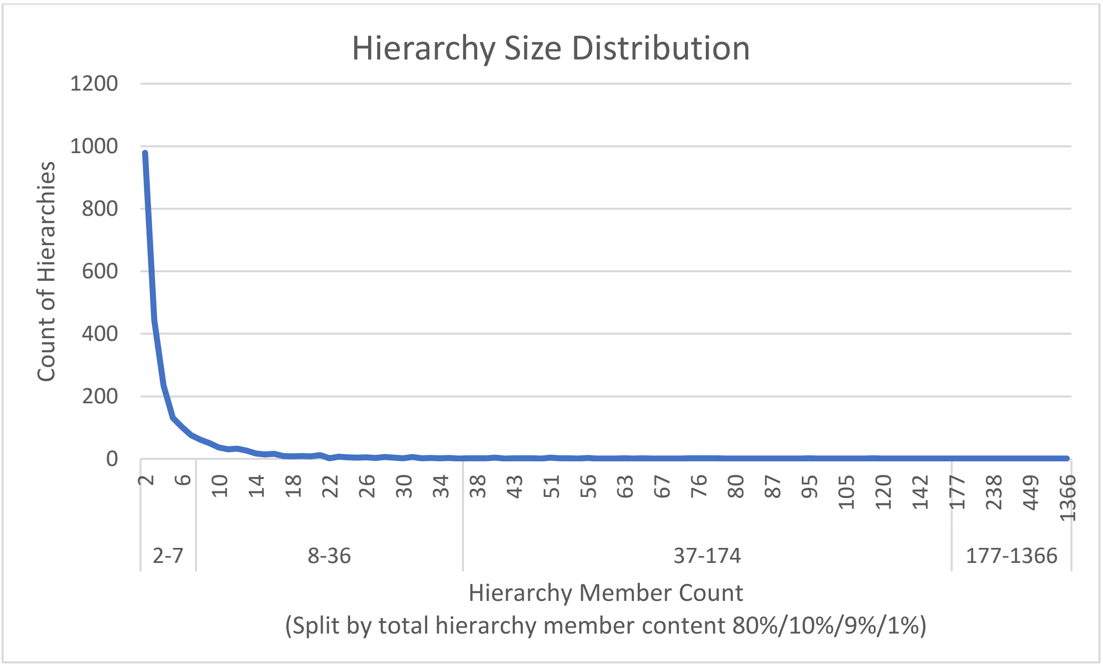


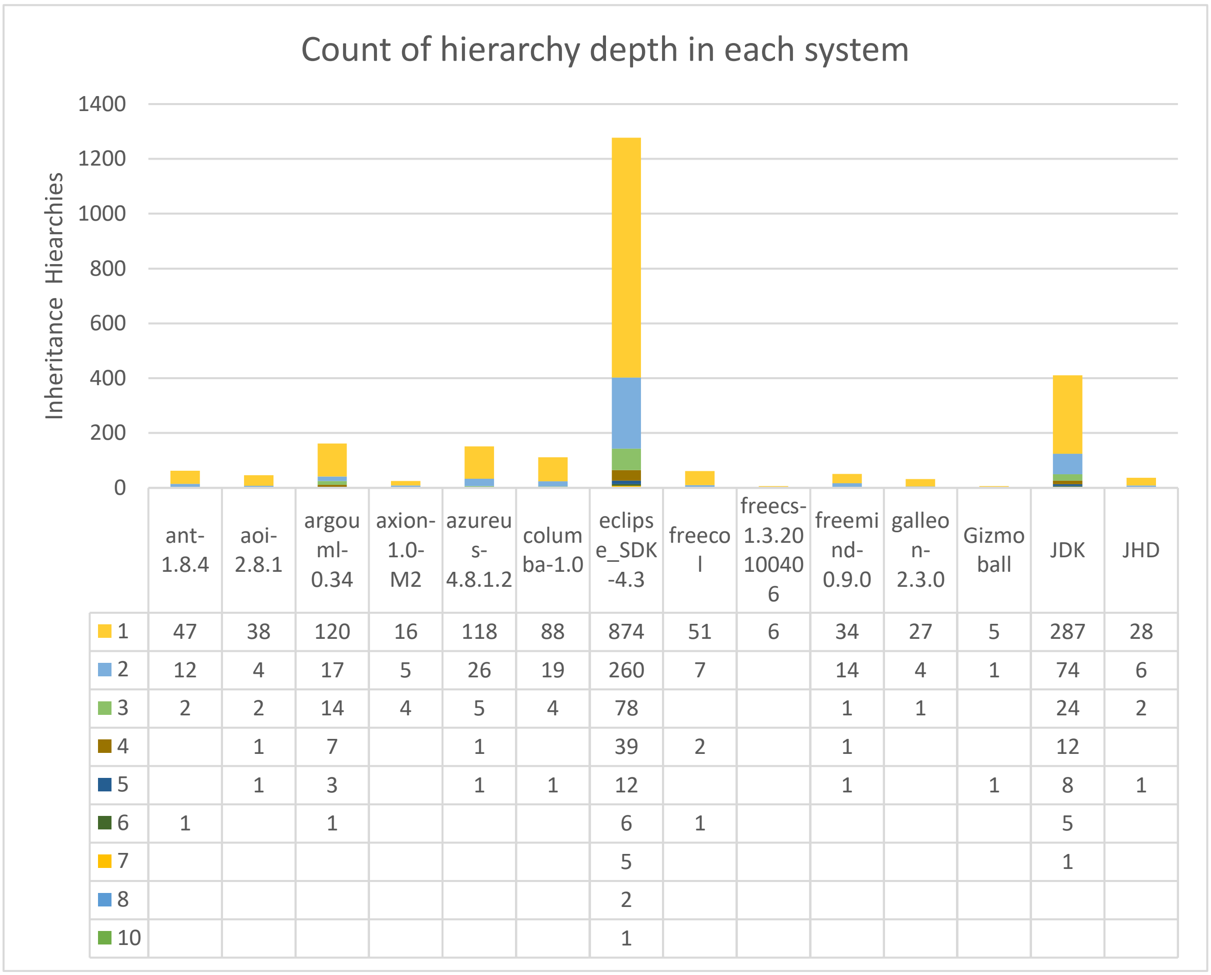
Figure 33 : Size in Inheritance Hierarchies

Since there is no indication within each hierarchy member of a hierarchy's size, using or extending a hierarchy member may ignore an arbitrary number of hierarchy members.

5.5.3.1 Depth of Inheritance

In this study, DIT_{MAX} has been measured by isolating all the inheritance hierarchies in each system. This included third party types when available, and also java library types (excluding the ubiquitous *java.lang.Object*). The depth of each hierarchy was then calculated by a simple recursive algorithm, starting from a count of zero at the root of the hierarchy. When calculating depth, interfaces are not counted.

Figure 35 shows the distribution of depths of inheritance hierarchies in the corpus. For example, the first column (ant-1.8.4) shows that the *ant* system contains 47 inheritance hierarchies with a DIT$_{MAX}$ of one, 12 hierarchies with a DIT$_{MAX}$ of two, two hierarchies with a DIT$_{MAX}$ of three, and finally one hierarchy with a DIT$_{MAX}$ of six. Note that the number of members in each hierarchy are not considered in this chart.



	ant-1.8.4	aoi-2.8.1	argouml-0.34	axion-1.0-M2	azureus-4.8.1.2	columba-1.0	eclipse_SDK-4.3	freecol	freecs-1.3.20100406	freemind-0.9.0	galleon-2.3.0	Gizmoball	JDK	JHD
🟨 1	47	38	120	16	118	88	874	51	6	34	27	5	287	28
🟦 2	12	4	17	5	26	19	260	7		14	4	1	74	6
🟩 3	2	2	14	4	5	4	78			1	1		24	2
🟫 4		1	7		1		39	2		1			12	
🟦 5		1	3		1	1	12			1		1	8	1
🟩 6	1		1				6	1					5	
🟨 7							5						1	
🟦 8							2							
🟩 10							1							

Figure 34 - This chart shows counts of inheritance hierarchies by maximum depth of any root to leaf

Most inheritance across the systems is trivial, with well over half (1739 of 2440, 71%) of hierarchies observed being of depth one. Additionally, all systems have >60% (avg. 76.66%) depth two hierarchies.

Tempero et al. notes that "*the depth of classes in the inheritance tree, does not increase with program size*" (Tempero et al., 2008). However, there is a clearly difference in use of (or attitude to) deeper inheritance hierarchies. While the deeper hierarchies are not restricted to larger systems, it is perhaps fair to say that there is more opportunity to accommodate more and deeper hierarchies in larger systems.

Recommended values for DIT range from a 'typical' value of 2 (Ferreira et al., 2012), in a survey of practitioners, Gorsheck obtained *preferred* thresholds of 3 or 5 for inheritance depth, 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*" (Gorschek et al., 2010).

If a depth of two, three, or five were assumed as a 'safe' maximum, this would exclude 10.33%, 4.71%, or 0.94% of hierarchies respectively. It is not clear what to do with this information. Ideally, guidance would have to cover every possible case – but a *rule of thumb* may only need to be applicable in *most* cases. However, if a hierarchy satisfies other design criteria (usefulness, domain modelling, low complexity), a depth cut-off seems arbitrary.

The deepest hierarchy (`org.eclipse.core.commands.common.EventManager`, depth 10) from the eclipse system is a performant replacement for a common Java library type (`java.util.Observable`). However, it appears that many of the sub-classes of `EventManager` are *implementation inheritance* (Liskov, 1988) seeking to re-use the `EventManager` public methods and functionality while ignoring, in many cases, the accumulated functions of the super-types. As such this may not be a 'proper use' of inheritance. Large hierarchies are discussed in more detail in section 5.5.7.2.

A further aspect to consider is reuse. The original DIT metric was inspired by the trade-off between beneficial reuse and the *cost* of keeping track of an increasing number of methods and classes (Chidamber and Kemerer, 1994). The relative lack of deep hierarchies indicates that the anecdotal 'unmanageable' deep hierarchies are relatively rare.

In addition to application size, the notable users of inheritance (eclipse, JDK) are large, mature frameworks, which have been the subject of intense design effort. As such, much of the common behaviour may have been consolidated – the presence of inheritance may indicate the degree of 'compactness' in the design.

It is possible that other factors are in play – some work on inheritance depth has focused on non-functional factors such as conceptual integrity in hierarchies (Dvorak, 1994), or effect of inheritance depth on maintenance (Daly et al., 1996; Harrison et al., 2000). However, there is no general accepted way to assess the conceptual integrity or maintainability of an inheritance hierarchy.

Summary of Depth of Inheritance: [KF 5.5] Most inheritance is shallow. [KF 5.6] Inheritance hierarchy depth thresholds proposed in other work are not representative of 'normal' use – with many falling well under the limits and well over the limits.

5.5.3.2 Hierarchy Width

This study has not focused on the conventional NOC metric (Chidamber and Kemerer, 1994), instead looking at overall hierarchy width. The NOC width measurement focuses on the number of descendants from an individual super-type. This is considered a measure of hierarchy complexity (more methods and classes), member complexity (behaviour may be difficult to predict), and potential reuse (Chidamber and Kemerer, 1994).

Harrison et al. discuss the issue of the representativeness of the traditional NOC measure and notes that their solution was to develop a similar metric, Breadth of Inheritance Tree (BIT) which summarises the width of a hierarchy into a vector of widths at each hierarchy depth (Harrison et al., 2000). In this study, the interest was primarily in capturing a single notion of width per hierarchy, without any notion of *growth rate* which the vectors generated by the BIT metric might permit.

Hierarchy width in this case is defined as BIT_{MAX}, the maximum count of classes at any depth throughout the entire depth of a hierarchy. This figure will be at least as large as the largest NOC measure in each hierarchy, and would be the maximum value in the BIT vector. So, previous work on NOC will provide a lower bound on what is observed when examining hierarchy width as it is defined here.

Figure 36 shows the structure of a hierarchy from the *ant* system – where yellow circles are local types that are abstract classes, and red circles indicate local concrete classes. This hierarchy may be considered 'well formed' in that it has an abstract body, with concrete class leaves. In this hierarchy, the NOC for any node will be no higher than two (for direct children), however the BIT (hierarchy width) of the third level is four (four classes wide). If this hierarchy were to continue to expand in the same way, the NOC would remain steady at a value of two for any given node, while the BIT would double at each level. The BIT_{MAX} of the hierarchy is the largest number in the BIT column.

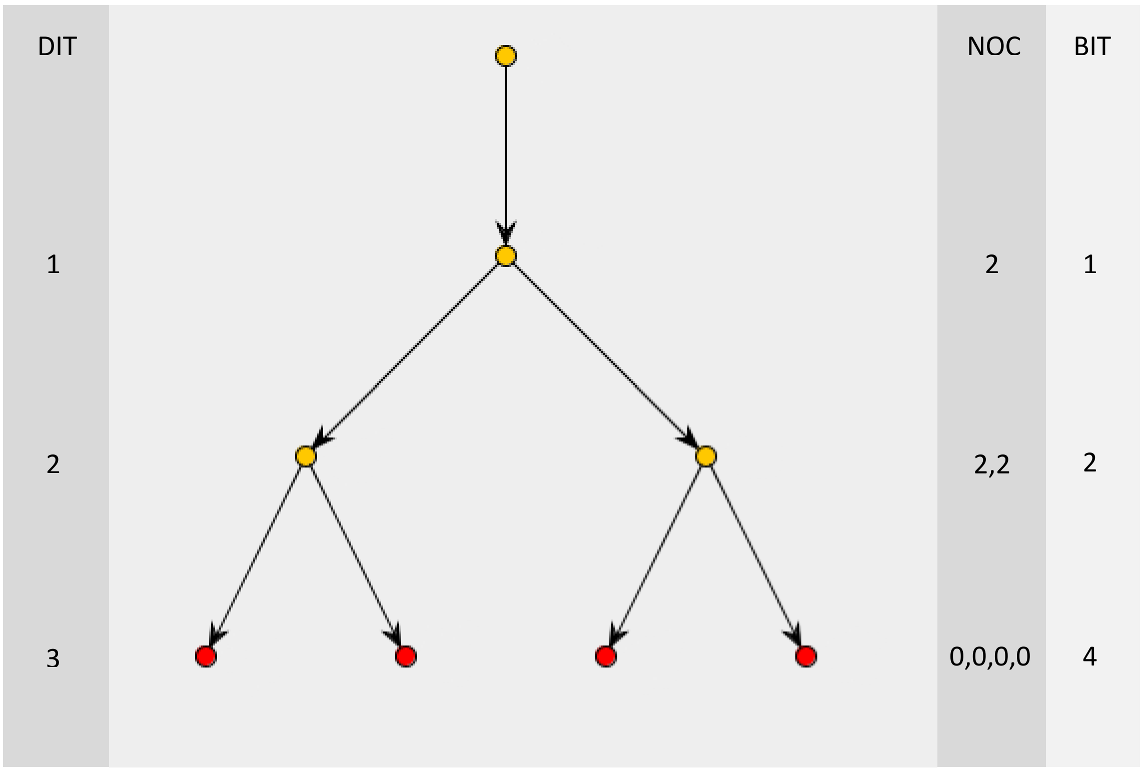


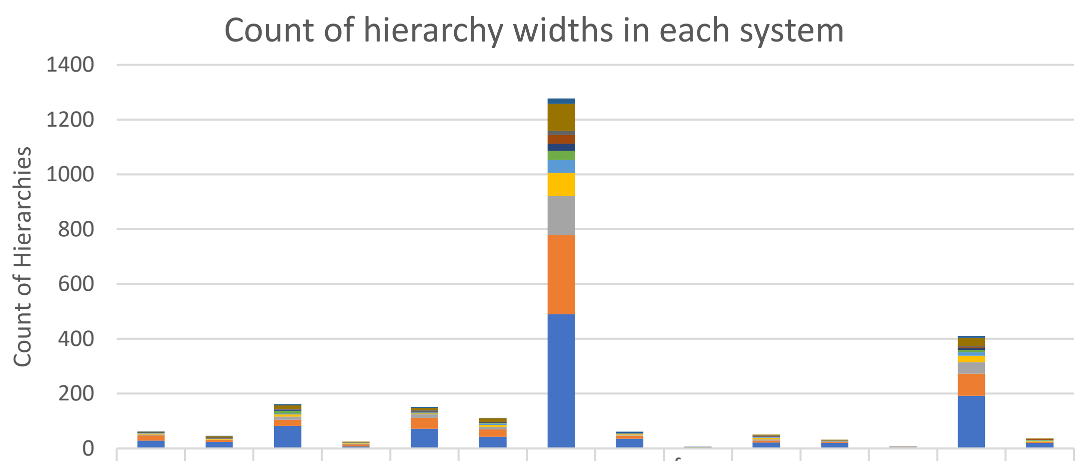
Figure 35 - Illustration of NOC_x: 2 vs BIT_{MAX}: 4 (org.axiondb.event.BaseTableModificationListener)

Figure 36 shows the BIT_{MAX} of the hierarchies in the examined corpus – for example the left most column indicates that ant-1.8.4 has 29 hierarchies of width one (light blue), up to one hierarchy in the top 1% range (44-542).

Like depth, small values dominate across the systems. Many inheritance hierarchies (1043, 43%) do not branch (are width one), and 21% (515) of hierarchies' branch only once. While the accumulation of ancestors is a concern for comprehension of leaf types, there is less discussion of hierarchy width in the literature. Wider hierarches may generate more paths from hierarchy root to leaf, but sibling classes (at the same depth) do not increase complexity as ancestors do. Consequently, the perceived 'cost' of widening a hierarchy may be less than that of deepening it.

In contrast to depth (Figure 35), wide hierarchies are more common in systems independent of size. Most systems contain a hierarchy in the top 10% of widths, and many having at least one in the top 1%. Another point to note is the very high numbers in the top 1% (44-542) range of hierarchy widths. Depth tails off at depth 10 in this study, though depths as high as 39 have been reported in other studies (Collberg et al., 2007). This is still an order of magnitude lower than the widest hierarchies (542) indicating that hierarchies grow more by widening, than by deepening – and that that widening is less limited.

This may contribute to high fan-in objects identified by Potanin et al., as a hierarchy widens the root type may attract more dependants to use the types in the hierarchy polymorphically, especially if each leaf or subtype represents a separate use case or service (Potanin et al., 2005). This is not necessarily an indicator of poor design, simply that useful generalisations may attract many dependents.



	ant-1.8.4	aoi-2.8.1	argouml-0.34	axion-1.0-M2	azureus-4.8.1.2	columba-1.0	eclipse_SDK-4.3	freecol	freecs-1.3.20100406	freemind-0.9.0	galleon-2.3.0	Gizmoball	JDK	JHD
🟦 44 - 542	1	1	5		3	2	20	4	1				7	
🟫 10 - 42	2	5	13	3	9	12	98	2	1	4	3		30	3
⬛ 9		2	1		6	1	15			1			4	
🟫 8		1	1		1	2	32			2		1	5	3
🟦 7	2	1	5			2	26	1		1			5	
🟩 6	1		9		1	3	33						9	3
🟦 5	2	1	4	3	3	4	47	2	1	4	2	1	13	1
🟨 4	3	3	8	2	3	7	85	4		6			23	2
⬜ 3	4	1	11	3	14	10	142	2	1	4	3	1	42	1
🟧 2	18	7	23	7	39	26	289	10		7	4		81	4
🟦 1	29	24	82	7	72	43	490	36	2	22	20	4	192	20

Figure 36 - Hierarchy widths in the examined corpus. Note that the top 1% and top 10% have been compressed for scale

Guidelines recommend that hierarchies should be refactored to extract intermediates with common behaviour when the number of immediate subtypes from one parent is larger than some value such as 27 (Johnson and Foote, 1988) or 9 (Suryanarayana et al., 2015). It is not clear if the wide hierarchies identified here have no common behaviour, or if this has simply not been consolidated.

Summary of Hierarchy Properties: *[KF 5.7] Most hierarchies are simple and narrow (width 1 or 2), with depth rarely reaching 10, while width often reaches 10 and frequently goes well beyond this value. [KF 5.8] Population is not uniform and this imbalance calls into question the representativeness (and thus usefulness) of system and corpus level averages. [KF 5.9] Depth may be limited somewhat by system size, width less so. [KF 5.10] Contrary to guidance, very wide hierarches are common.*

5.5.4 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 super-classes and interfaces. Figure 37 shows the distribution of size in native interfaces among hierarchy members – this yields a long-tailed distribution with 80% of hierarchy members having 10 or fewer public methods.

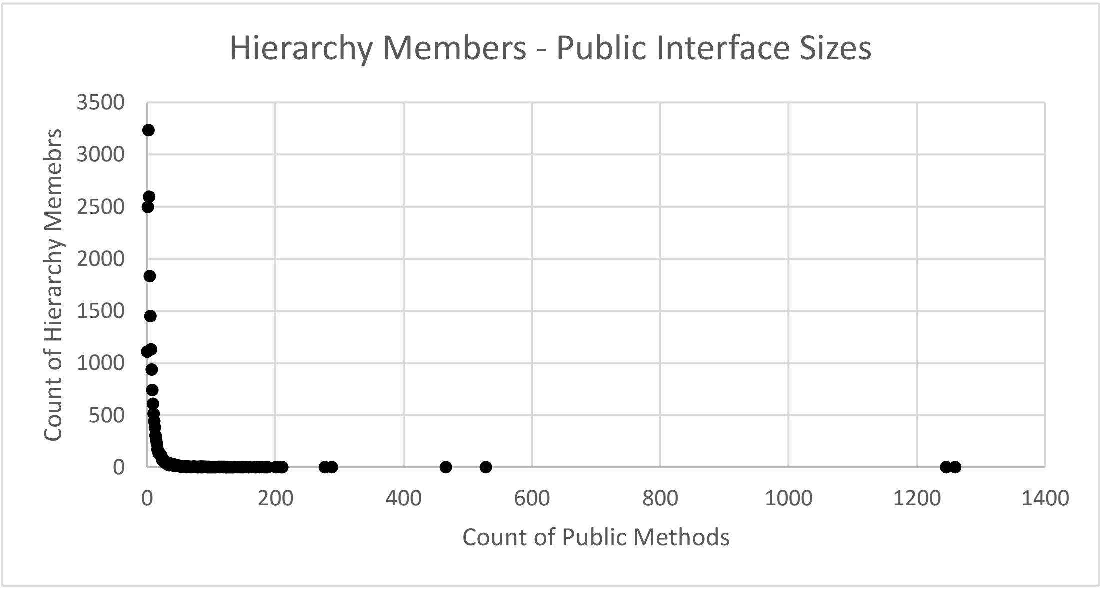


Figure 37 - Hierarchy Members - public interface size (public method count)

Most inheritance members (and thus hierarchies) have relatively small (narrow) public interfaces. Based on public method count, the top 10%, 1%, and 0.1% of hierarchy members have methods in the range 17-1260, 56-1260, and 150-1260 respectively. The inheritance hierarchy members with very high numbers of methods are shown in Table 32. Note that the very largest hierarchy members (in terms of public methods) are all constrained, to a greater or letter extent, by some external standard. Three of the largest hierarchy members are 'generated classes', while another is constrained by operating system API calls.

Table 32 - Top 0.1% of hierarchy members by size (number of public methods)

Inheritance Member	Methods	Purpose
org.eclipse.jface.text.TextViewer	150	Text displaying ‘widget’
org.eclipse.ui.internal.WorkbenchPage	150	UI view manager
org.eclipse.jdt.internal.debug.eval.ast.engine.ASTInstructionCompiler	158	Compiles an AST to instructions
javax.swing.text.JTextComponent	159	Base class for text component
javax.swing.plaf.synth.ParsedSynthStyle	168	Look and feel format component
org.eclipse.jdt.internal.corext.dom.GenericVisitor	170	No-op visitor for AST
org.eclipse.jdt.core.dom.DefaultASTVisitor	170	No-op default visitor for AST
org.eclipse.swt.custom.StyledText	175	Styled text renderer
org.eclipse.swt.internal.ole.win32.COM	183	Native OS system calls
org.eclipse.swt.internal.gdip.Gdip	186	Low-level graphics operations
org.eclipse.jdt.internal.corext.dom.HierarchicalASTVisitor	188	ASTNode decorator
net.sf.freecol.common.model.Unit	201	Game concept, fat class
net.sf.freecol.common.model.Player	209	Game concept, fat class
javax.swing.JTree	211	Complex JComponent
javax.swing.JTable	277	Complex JComponent
com.sun.corba.se.impl.logging.POASystemException	288	Generated class, POA protocol
org.eclipse.jdt.internal.compiler.problem.ProblemReporter	466	Error reporting tools
com.sun.corba.se.impl.logging.OMGSystemException	528	Generated class, ORB protocol
org.eclipse.swt.internal.win32.OS	1246	OS system calls
com.sun.corba.se.impl.logging.ORBUtilSystemException	1260	Generated code, ORB protocol

In addition, there are design constraints on other large hierarchy members that indicate that their size is not avoidable. For example, the abstract-syntax tree, and Java byte-code facing classes must provide, respectively, many similar methods, overloaded methods, and methods for handling each byte-code instruction. It is unlikely that the classes could be smaller while preserving cohesion.

Consequently, when assessing size, it may be appropriate to consider the size of the set of possible messages. For example, a class may have a getter-setter pair for each attribute or handling method for each event type over some set of related properties. It may be difficult to argue that that class is too large simply because the set of properties in the problem (or indeed solution) domain is very large. Under these circumstances, a dimension of the design (class size) is directly driven by a dimension in the problem domain (size of a set of related properties).

A *novel method* is defined here as a public method appearing in an inheritance hierarchy member that is not defined in the root type of the enclosing hierarchy. While a *root method* is a method appearing in a hierarchy member which is defined in a root type – redefinition likely means overriding.

Figure 38 shows the presence of novel and root methods at each depth of inheritance hierarchy. For example, the first column shows that at inheritance depth one (where the hierarchy root is depth zero), among all hierarchy members at that depth, 89590 inherited public root methods are redefined, and 77903 public non-root methods are defined. Hierarchy members at depth one and two have a wide spread of novel method proportions. However, with increasing depth in a hierarchy, there are more hierarchy members with proportionately more novel methods, until depth ten, where most (86-94%) of the public methods in each hierarchy member are novel. Note also that the amount of overriding falls away with hierarchy depth indicating that very deep hierarchy members may be adding variation by some other means, such as re-defining non-root methods.

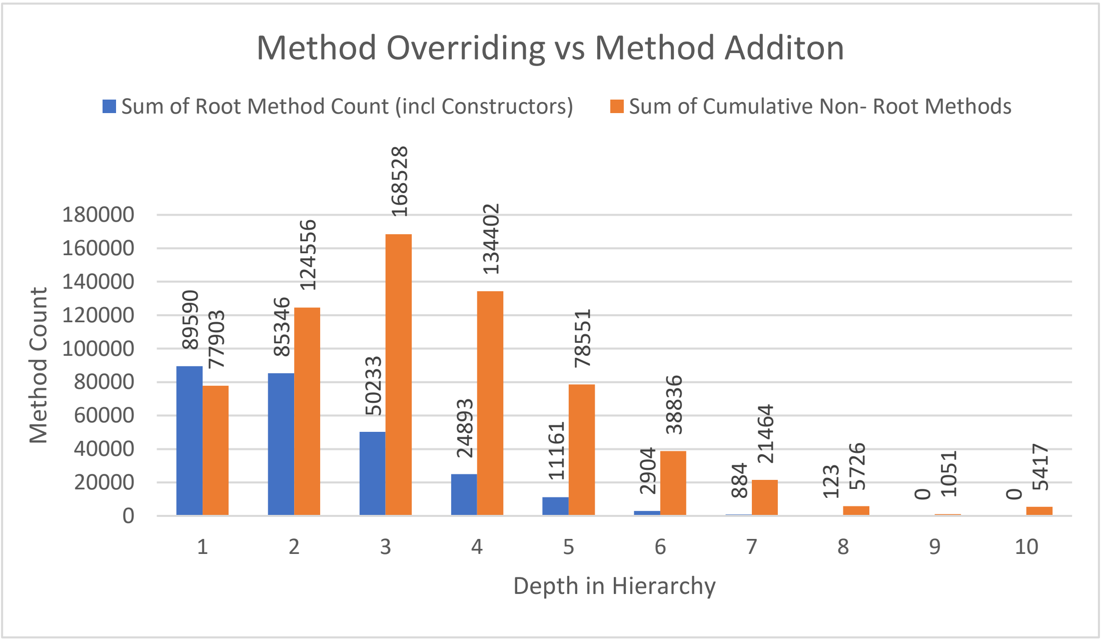


Figure 38 – Overriding vs novel methods

In shallower areas of the hierarchies there are members with very high novelty scores, indicating that novelty does not need to gradually build up. Conversely deep hierarchy members show less of a continuum – there is a split around depth 4-5 where hierarchy members either become more novel or stay at novelty zero.

These zero-novelty hierarchy members may only override or use constructor configuration to introduce variation, rather than novel methods. Around a twentieth (9542, 6.51%) of the hierarchy members analysed are in this 'zero' category, which do not add any novel public methods to their interface. These hierarchy members may be mostly or completely motivated by polymorphism – if so polymorphic criteria may be applied more stringently during assessment of these hierarchy members. 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. The largest no-novelty hierarchy (*javax.swing.plaf.*

nimbus.State) has 22 members, and the hierarchy member with the largest public interface (most methods) (*org.eclipse.e4.ui.css.core.impl.dom.CSS2PropertiesImpl*) in a no-novelty hierarchy is 245 methods, again this non-novel hierarchy is shallow (depth 1).

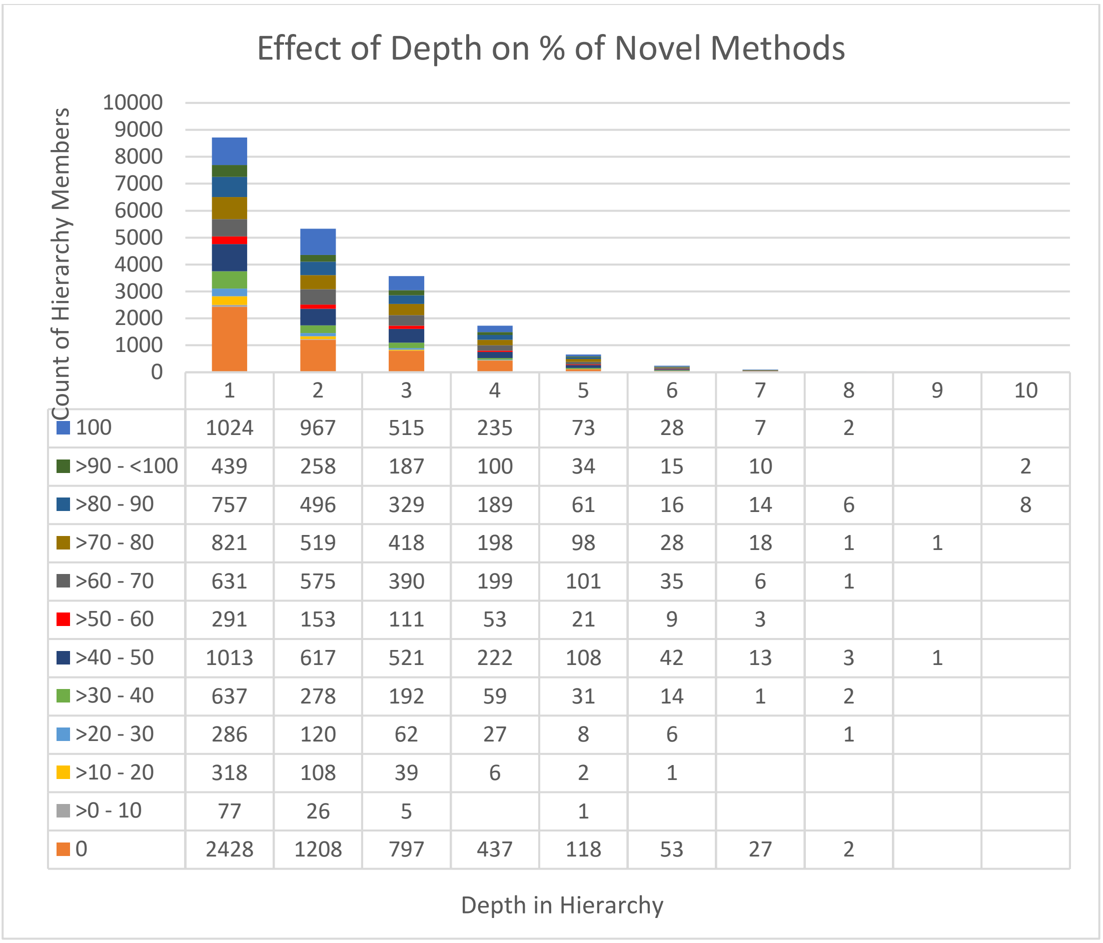
In terms of *normal use*, Figure 38 illustrates that a great many subtypes provide methods other than the public interfaces of the root type in their hierarchy. In addition, method overriding appears to tail off as the hierarchies deepen. While there is no direct relation to a specific guideline, the addition of novel methods indicates that access to hierarchy members is available other than via the native interface of the root-type, indicating that substitutability with super-type (polymorphism) is not the sole driver for inheritance use. This suggests that differentiation in shallow hierarchy members is via overriding *and* adding functionality, while deeper hierarchy members tend to be distinguished via adding behaviour.

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 might instead be 'composed' with the shallower parts of the hierarchy.

Figure 39 shows an effect of depth in hierarchy on the amount of novel (non-root) methods in each type – note that the >50-60% novelty category is marked in red, so anything above (and including) this category indicates that these children are more different from the root type than they are like the root type. For example, the first column indicates that there are just under 9000 hierarchy members at depth one – of these 2428 have no novel methods.

Figure 39, shows that a quarter of (non-root) inheritance members (5070, 25%) have zero novelty (introduce no new methods), and one-seventh are completely novel (2851, 15%). The effect of depth shows that there is a 'base' of zero-novelty in each depth category down to a depth of eight, while the remaining hierarchy members tend towards mid-high novelty as depth increases. Notably – there is also a bump in the figures in the 40-50% range indicating that slightly more hierarchy members than might be expected sit at around half inherited, half novel methods. This increase is noticeable, even as hierarchies deepen.

Niculescu et al. proposed that inheritance hierarchy members should contain a balance of inherited and implemented behaviour, with 50/50 being a *good* range this balance, with a tolerance of around 25% (Niculescu et al., 2015). Using this 25-75% novelty range covers around two-fifths (7985, 39%) of hierarchy members, indicating that an assumption of balance may be useful in some cases, but is by no means the general case.



	1	2	3	4	5	6	7	8	9	10
🟦 100	1024	967	515	235	73	28	7	2		
🟩 >90 - <100	439	258	187	100	34	15	10			2
🟦 >80 - 90	757	496	329	189	61	16	14	6		8
🟫 >70 - 80	821	519	418	198	98	28	18	1	1	
⬛ >60 - 70	631	575	390	199	101	35	6	1		
🟥 >50 - 60	291	153	111	53	21	9	3			
🟦 >40 - 50	1013	617	521	222	108	42	13	3	1	
🟩 >30 - 40	637	278	192	59	31	14	1	2		
🟦 >20 - 30	286	120	62	27	8	6		1		
🟨 >10 - 20	318	108	39	6	2	1				
⬜ >0 - 10	77	26	5		1					
🟧 0	2428	1208	797	437	118	53	27	2		

Figure 39 - Effect of depth on % novel methods

Summary of Public Interfaces: [KF 5.11] Large public interfaces are relatively rare in hierarchy members; most hierarchy members have narrow public interface (<10 methods). [KF 5.12] Some large hierarchy members are generated code or are constrained to be large due to some external factor such as a domain entity – so the reason for their size is non-local. [KF 5.13] Novel methods are common - most inheritance hierarchy members are not consistent with the native interface of their root type. [KF 5.14] Zero-novelty hierarchy members are usually in shallow hierarchies – novelty (adding functionality) may be required for depth. [KF 5.15] The frequency of novel methods indicate that reuse is a more common motivator than polymorphism [KF 5.16] Overriding falls away faster with depth than adding more methods.

5.5.5 Hierarchy Structure

Inheritance use unavoidably creates structures as it is used. More extensive use of inheritance creates larger structures – a single *root* type may support tens or even hundreds of subtypes.

5.5.5.1 Shape

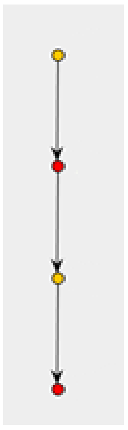
While some metrics capture the shape of a hierarchy indirectly, shape has not previously been studied in detail. The structures identified here are based on observed structural differences between inheritance hierarchies, and not on any specific design guidance:

- *Line* - hierarchy with no branching (includes simple two-member hierarchies)
- *Fan* - hierarchy consists solely of one root and its immediate sub-types, previous literature has noted very wide hierarchies as a *flying saucer* shape (Lanza, 2003)
- *Subtrees* - hierarchy consists of a branching root with at least one other branching node
- *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.
    - May be considered a derivative *Fan*.

These categories are based on observations via visualisation of hierarchies and provide an immediately accessible summary of an inheritance structure's shape. Furthermore, these categories can be applied to any hierarchy, and include all possible legal hierarchy structures.

Examples of each shape are shown in Figure 41, Figure 42, Figure 43, Figure 44, and Figure 45. These figures show schematic views of inheritance hierarchies from systems in the Qualitas Corpus.   Round nodes indicate that a type is defined in local code (not third party or language libraries), the colours are red for concrete classes and yellow for abstract classes.

Figure 44 – Line -
org.eclipse.debug.internal.ui.viewers.AsynchonousModel



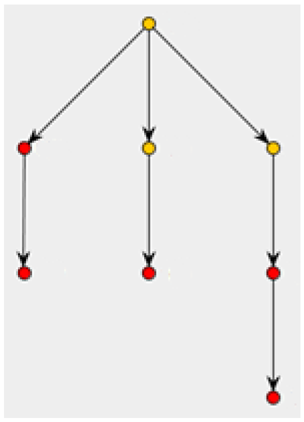


Figure 43 - Branch-Line -
org.eclipse.team.internal.core.subscribers.ChangeSetManager

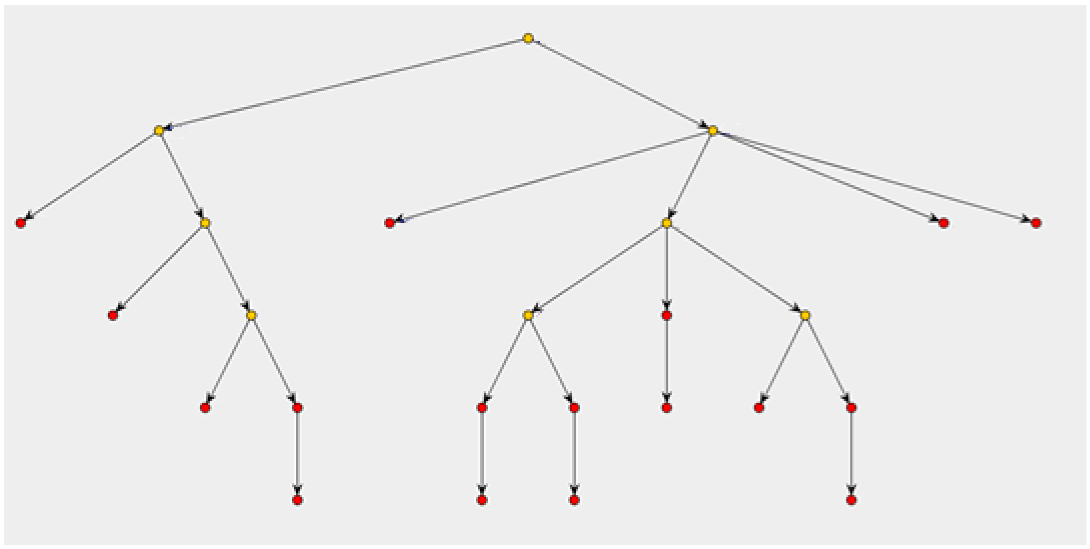


Figure 42 - Sub-Trees - org.eclipse.jdt.apt.core.internal.declaration.EclipseDeclarationImpl

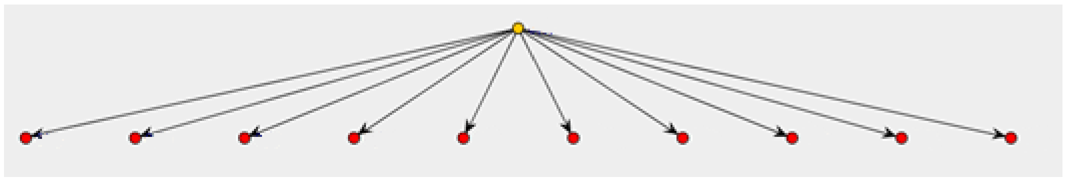


Figure 40 - Fan or 'Flying Saucer' - org.eclipse.swt.graphics.Resource

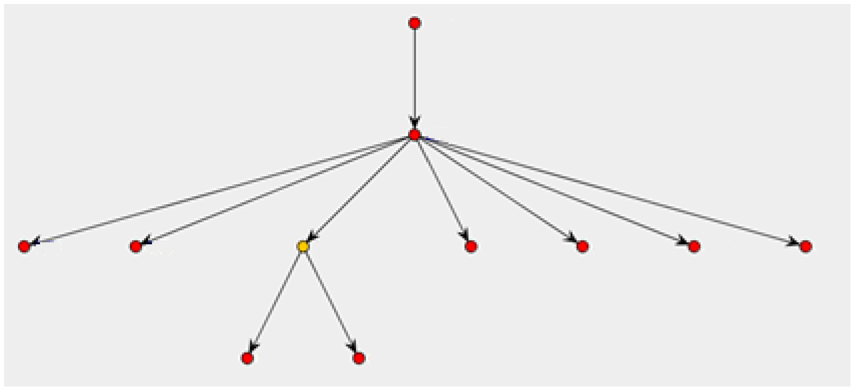
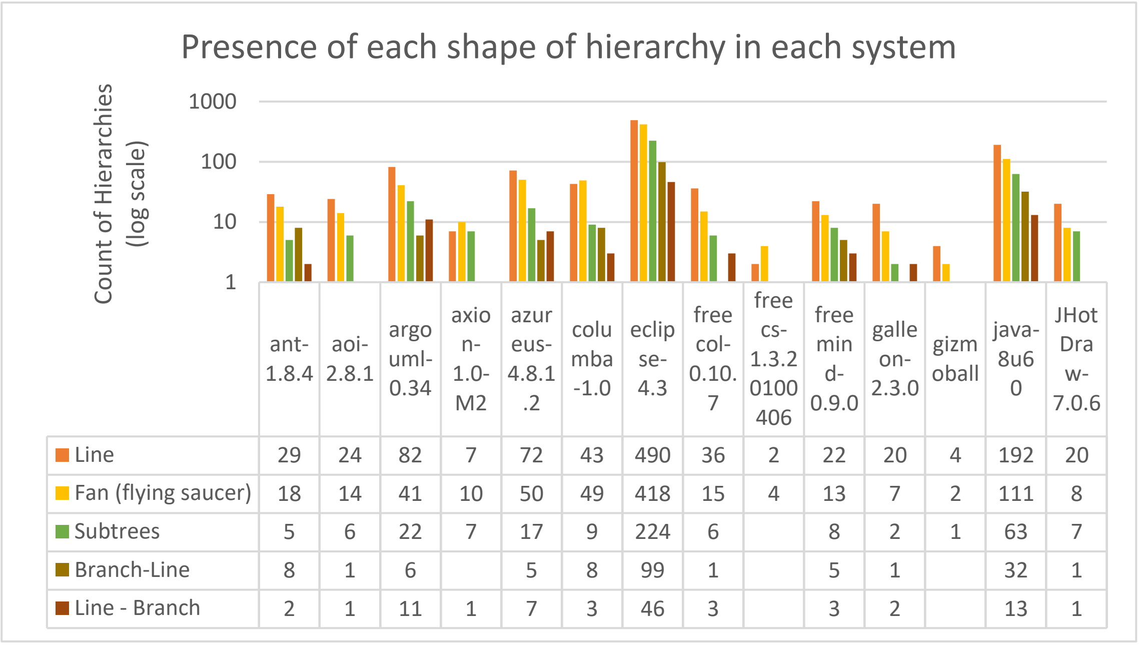


Figure 41 - Line-Branch - org.eclipse.jdt.internal.ui.viewsupport.JavaUILabelProvider

Figure 45 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).



	ant-1.8.4	aoi-2.8.1	argouml-0.34	axion-1.0-M2	azureus-4.8.1.2	columba-1.0	eclipse-4.3	freecol-0.10.7	freecs-1.3.20100406	freemind-0.9.0	galleon-2.3.0	gizmoball	java-8u60	JHotDraw-7.0.6
🟧 Line	29	24	82	7	72	43	490	36	2	22	20	4	192	20
🟨 Fan (flying saucer)	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 45 - Breakdown of shape in corpus systems. Note: Log scale on number of hierarchies

Line dominates – a further indication that much of the inheritance present in systems is un-complicated (in this case non-branching). The next most common shape in each system is *fan*, which represent many one-depth variations on a single super-type. The *line* and *fan* shapes account for 74% (1803) of all hierarchies examined – 94% (979) of line-shaped hierarchies are of depth one, and all but one line-hierarchy is *shallow* (DIT_{MAX} 1-3). Finally, *line* hierarchies appear with 979 at depth one, 58 at depth two, 5 at depth three, and 1 at four, indicating that depth 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* (DIT_{MAX} 5-6) and 96% of deep (DIT_{MAX} 7-10) hierarchies – indicating that branching is required for depth. If we consider hierarchy width (BIT_{MAX} 1-542), 80% of hierarchies are of width 4 or lower, so width 2-4 might be considered a *reasonable* size for most hierarchies. Of the 497 hierarchies with a width (BIT_{MAX}) 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 *sub-tree* 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 two thirds of hierarchy members being part of wider, deeper, overall larger hierarchies, this may be the default impression of inheritance hierarchy members, even though these members represent only 15% of hierarchies. This may skew the perception of inheritance use, since the larger (and potentially less comprehensible) hierarchies are more visible.

The largest hierarchy found has the root `org.eclipse.core.commands.common.EventManager` and consists of 1366 types in a hierarchy of depth 8, and root methods are never overridden. The root of the hierarchy defined a replacement for a common idiom in OO languages – an observable type to which other objects can subscribe. There are 138 nodes which branch, many with dozens of children. The hierarchy is too large to show here in any practical format.

Summary of Shape: [KF 5.17] The same profile of shapes appears at the system level in most of the corpus. [KF 5.18] Inheritance use is dominated by simple hierarchy structures – line and fan. [KF 5.19] The fan structures examined indicate that hierarchies can grow arbitrarily wide. [KF 3.20] Deeper hierarchies without branching are rare. [KF 5.21] Relatively few hierarchies contain over half of inheritance hierarchy members [KF 5.22] The imbalance in scale of the larger hierarchies skews the apparent complexity of hierarchies.

5.5.6 Abstract classes

In this section, the use of abstract classes in inheritance hierarchies is discussed. 54% (1325) of the inheritance hierarchies examined contain at least one abstract class – with 50% (1207) of the hierarchies containing 1-3 abstract classes. This is especially true in larger hierarchies where the abstract hierarchies form large *backbones* within the hierarchies. Hierarchies are observed to have either a *contiguous backbone* where all the abstract classes in the hierarchy form a contiguous structure, or a *fragmented backbone* has gaps in the structure of its abstract classes. Examples of intact (Figure 47) and fragmented (Figure 46) use of abstract classes are shown below. Note that in Figure 46, the large yellow arrows indicate the locations of abstract classes, all other types in the hierarchy shown are concrete classes.

Inheritance is used to satisfy two properties – polymorphism, and reuse – which require a common root type, and new classes with access to the re-used code respectively. These properties do not place any constraints on the intermediate structure connecting the root type to the eventual leaves.

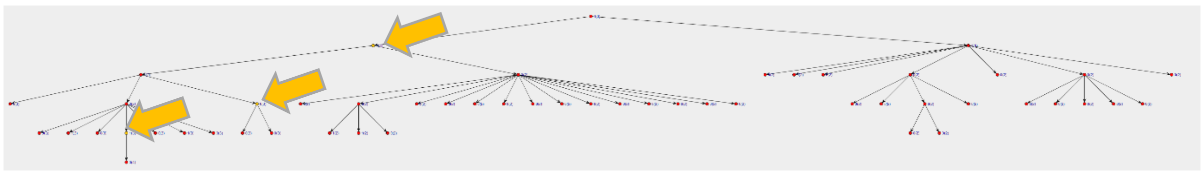


Figure 46 - Hierarchy with sub-trees shape and fragmented abstract class structure, abstract classes indicated with yellow arrows (freemind.extensions.HookAdapter)

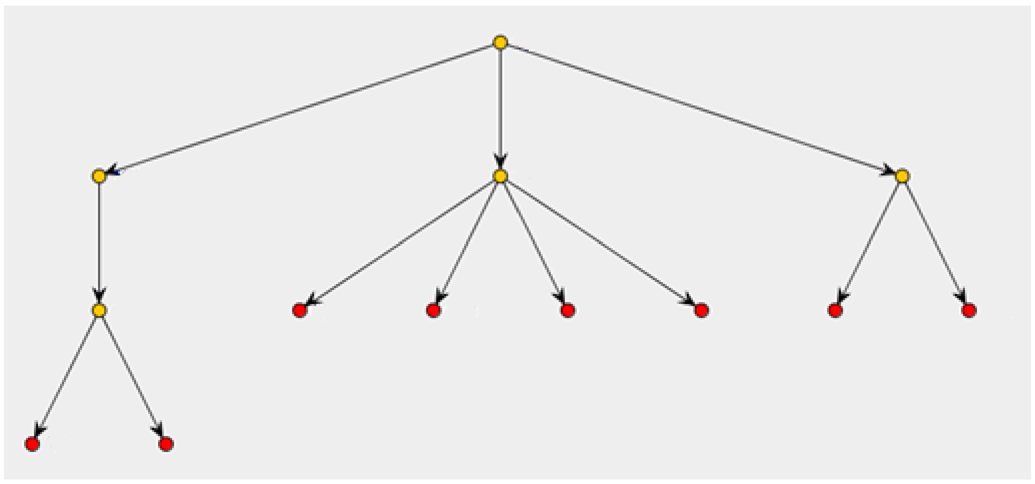


Figure 47 - Hierarchy with a complete abstract 'backbone' (freemind.modes.LineAdapter)

The relevant guidance encourages the use of abstract classes to fill in the intermediate structure between the root type and the leaves of a hierarchy. The most relevant guidance is the Dependency Inversion Principle which states that “*High level modules should not depend upon low level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions*” (R. C. Martin, 1996), and Johnson and Foote’s recommendation that “*The top of the class hierarchy should be abstract*” where their example indicates that one concrete class directly inheriting from another should be avoided by creating a common abstract super-type (Johnson and Foote, 1988).

In Figure 48 each grey circle represents the number of abstract classes in a hierarchy, while the red spots indicate the number of abstract classes in that hierarchy that are linked in a contiguous sub-tree (or *backbone*) within that hierarchy. Hence each grey circle containing a red dot represents a hierarchy with a *contiguous* backbone consisting of all abstract classes in the hierarchy.

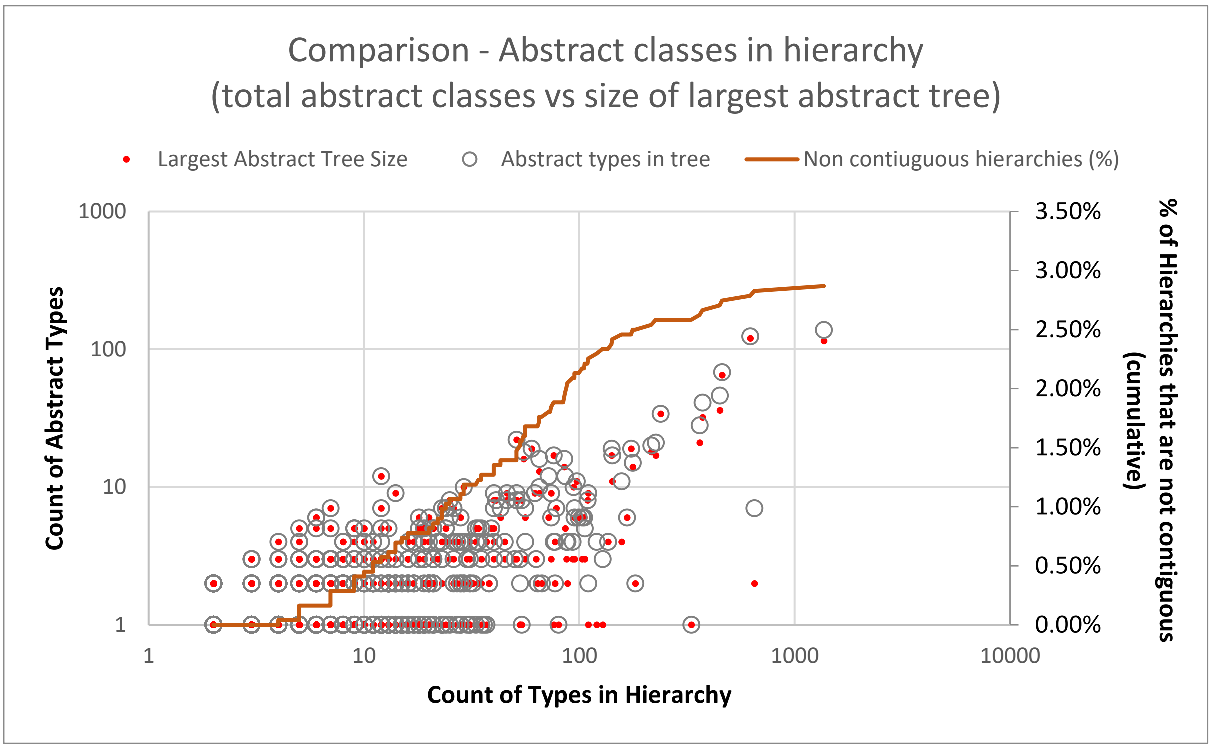


Figure 48 - Abstract classes in hierarchy vs largest contiguous abstract tree in that hierarchy

This shows that most of the hierarchies' largest abstract-class tree (the red dots) are the same sizes as the count of all abstract types on that hierarchy (the grey circle). 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. These *fragmented* hierarchies are indicated where the red spot is not in the centre of the grey circle. Note that while there are non-contiguous hierarchies as small as 7 members, most non-contiguity is found in the larger hierarchies, with over half of the largest 50 hierarchies being non-contiguous. The cumulative total percentage of non-contiguous hierarchies is shown as an orange line to illustrate that that the non-contiguous hierarchies have a higher count of types.

Hierarchies with fewer types (concrete classes and abstract classes) appear to have a more regular internal structure, as they lie on or near the diagonal indicating that all their abstract classes are on one contiguous structure. Among the very large hierarchies there is more of a mixture – there are at least two factors at play in these cases.

Firstly, as hierarchies become larger (usually via branching and depth as described above), a higher proportion of the hierarchy is composed of leaves, which are usually concrete classes – this may explain why the larger hierarchies do not approach the diagonal as much as the smaller hierarchies. Secondly, as hierarchies grow, abstract classes appear to be used more intermittently, introducing non-contiguous abstract classes, which means that (for each hierarchy) the largest contiguous tree (dot) will drift further from the count of abstract classes (grey circle).

With regards to the guidance, it may be better to assess – what proportion of non-leaf hierarchy members are abstract classes. It is also not clear if non-abstract non-leaf classes could or should be made abstract.

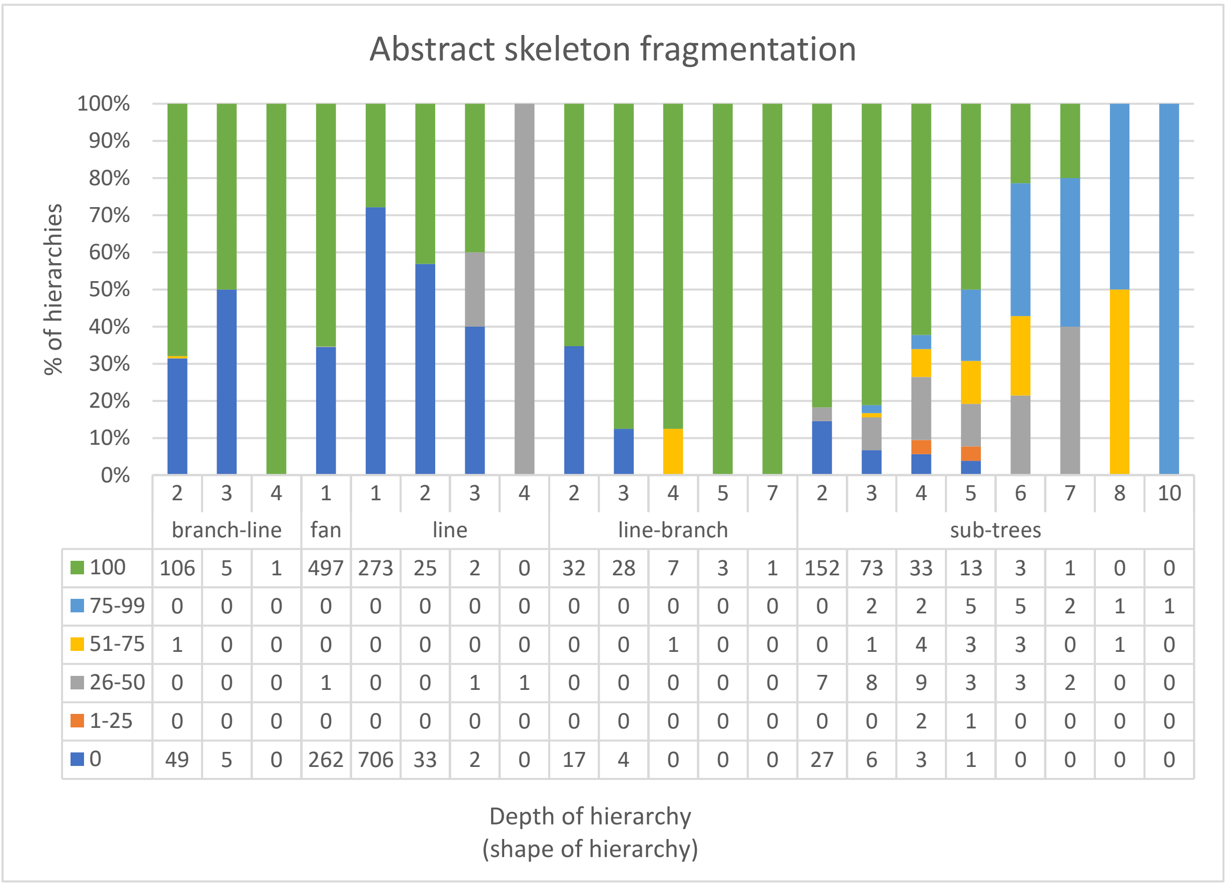
Summary of Abstract Classes: [KF 5.23] Abstract classes, are present in around half of the hierarchies and are, in most cases, arranged in a continuous 'backbone' within the hierarchy. [KF 5.24] Larger hierarchies make proportionately less use of abstract classes and are more likely to have non-contiguous abstract backbones.

5.5.6.1 Interaction of Shape and Abstract Classes

Only 3% (70 of 2440) of the hierarchies examined 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 *sub-trees* shape, with the other shape categories having only one or two out-of-order abstract classes (where an abstract class depends on a concrete class).

Figure 49 shows the hierarchies in each shape and depth division along the x-axis, note once again that depth one is root plus one level and so on. Each column is further divided by how cohesive the abstract backbone inside the hierarchies are in terms of what proportion of the abstract classes in the tree is contiguous (0%/1-25%/26-50%/51-75%/75-99%/100%). For example, the first column represents that there are 156 depth-2 branch-line hierarchies (root type plus two or more direct subtypes). The divisions in the bar indicate that most of these hierarchies (green segment) have an intact abstract skeleton, one has a somewhat fragmented inner structure (yellow segment), and the remainder have no abstract classes (blue segment).

Overall, a similar pattern to that shown in Figure 48 is shown – the large number of green segments indicate fully contiguous abstract-class structures across many of the shape categories. Notably, this contiguity falls off with depth in sub-tree hierarchies, and increases with depth in line-branch hierarchies. In addition, when contiguous (100% green) falls away in sub-trees hierarchies, this is replaced by *almost* contiguous ratings indicating that there are holes in the structure of the abstract class backbone, rather than a general absence of abstract classes. Contrast this with the replacement of full contiguous backbones (green) with no backbones (light blue) at all in branch-line, fan, and line – indicating an all or nothing approach to adding abstract classes in these hierarchies.



	branch-line			fan	line				line-branch					sub-trees							
	2	3	4	1	1	2	3	4	2	3	4	5	7	2	3	4	5	6	7	8	10
🟩 100	106	5	1	497	273	25	2	0	32	28	7	3	1	152	73	33	13	3	1	0	0
🟦 75-99	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	2	5	5	2	1	1
🟨 51-75	1	0	0	0	0	0	0	0	0	0	1	0	0	0	1	4	3	3	0	1	0
⬜ 26-50	0	0	0	1	0	0	1	1	0	0	0	0	0	7	8	9	3	3	2	0	0
🟧 1-25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	1	0	0	0	0
🟦 0	49	5	0	262	706	33	2	0	17	4	0	0	0	27	6	3	1	0	0	0	0

Figure 49 - Shape and Depth of hierarchy effect on abstract backbone integrity. Note root = depth zero.

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 breech of the DIP - commonly where an abstract class inherits from a concrete class. Figure 46 shows an example of a sub-tree's structure with out-of-order use of abstract classes. Overall, 46% of hierarchies contain no abstract classes, of the remaining 54%, only 6% (146) are out-of-order. *Sub-trees* account for 75% (109) of out-of-order hierarchies – so in larger hierarchies, where a continuous chain of abstractions may be of most use in managing a complex abstraction – there are more out-of-order hierarchies. This may indicate that hierarches that grow over time are not being consolidated to keep their structure uniform.

Summary of Shape and Abstract Classes: [KF 5.25] In many hierarchies, there is an all or-nothing approach, where abstract classes in a hierarchy form a continuous structure, or are not present at all. [KF 5.26] In deeper hierarchies, abstract classes may form an inner structure, but these are often also out-of-order. [KF 5.27] Hierarchies which contain abstract classes are generally in-order, (most super-types are abstract classes).

5.5.7 Method Invocation

In this section, method invocations on inheritance hierarchy members are used to gain insight into how hierarchies are used in systems. Guidelines and textbook examples of inheritance hierarchies emphasise that hierarchy members are accessed polymorphically via the native interface of the hierarchy root. Alternatively, an inheritance member may be used directly, relying on hierarchy membership primarily for code-reuse.

Overall, 45% (18141, median 51%) of hierarchy members have no direct method calls (excluding constructors), this may indicate that these hierarchy members are being accessed polymorphically. This can be confirmed further by separating out method invocations to root types. Figure 50 shows method invocations on hierarchy members divided by locally defined methods (grey), and inherited methods (orange).

To see past the size differences between systems, each pair of rows represents 100% of the method invocations to hierarchies in each system. For example, the first two rows show that *jhotdraw* root types receive 78% of method invocations to hierarchy members, while the next row shows that the non-root methods receive 9% of method calls as local (novel) methods, and the remaining 13% are inherited method calls to non-root types.

In all but one system (freecol), there is a much higher proportion of method invocations to root types than subtypes. Calculated as total invocations to root types minus total invocations to non-root types - this ranges from 25-86% (median 49%) in favour of root types. This indicates that the 'textbook' view of accessing hierarchy members via the root-type is not unfounded, but that this does not explain all method invocations on hierarchy members. For example – the low invocation-to-root systems appear to have a higher proportion of invoking inherited methods on subtypes. To investigate this further, the pattern of method invocations must be considered for each hierarchy.

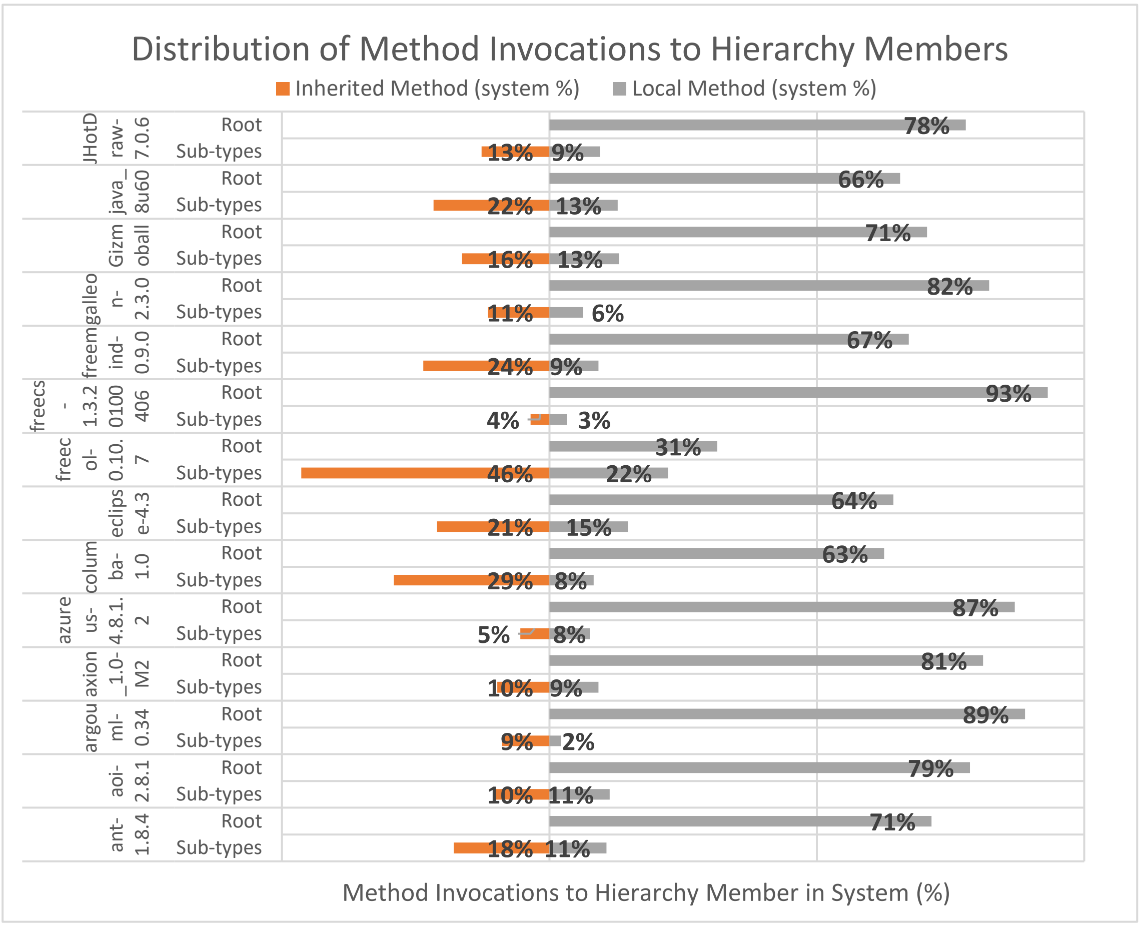


Figure 50 - Method Invocations - roots vs non-root types

Patterns of method invocation on hierarchies can be classified by the main motivators for the use of inheritance - polymorphism and reuse. Reuse is difficult to estimate as it may depend upon object composition, and how often methods are invoked at runtime (Niculescu et al., 2015). Polymorphic access, on the other hand may be estimated because it leaves a structural trace where dependent code refers to a super-type in place of a subtype. Each hierarchy member may be classified into one of four possible categories - *never directly invoked*, *invoked only for root-defined methods*, *invoked only for local (novel) methods*, *and invoked for both root-defined and local methods*. If, for example, a hierarchy is used polymorphically, methods calls would be directed to a reference of the root type, regardless of the actual subtype of the object handling the method call. From this, five distinct (automatically distinguishable) modes of use for hierarchies can be defined at the hierarchy level, shown in Table 33. The category names have been defined to reflect the use of the hierarchies.

Table 33 - Method invocation patterns

Invocation Pattern	**Description**	**Invocations on Root**	**Invocations on Non-Root**	
		Root Methods	***Root Methods***	***Novel Methods***
Strict Polymorphism	All invocations to hierarchy members are to root-type references (excepting constructors)	✓		
Common Interface Polymorphism	Methods are invoked on non-root type references, BUT only root methods are invoked.	✓	✓	
Balanced Reuse	Novel methods are invoked on hierarchy members, BUT if a member is directly invoked, at least one invocation is a root method.	✓	✓	
Aggressive Reuse	Members are only invoked for novel methods			✓
Fragmented	Invocations to hierarchy members are a mixture of the above.	Mixed		

Each category of use has been labelled to describe the property of the hierarchy that is captured. 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.

Categorisation was performed by applying the above criteria to each to method invocation on each hierarchy member, then to each hierarchy in turn and was applied strictly. For example, a single non-root invocation on one hierarchy member is sufficient to downgrade a hierarchy from strict- to common-interface polymorphism. As such *fragmented* is a catch all category for hierarchies where invocations pattern varies among members, and the entire hierarchy cannot be placed in one of the other categories.

Figure 51 shows the number of hierarchies which exhibit each pattern of method invocation. This is further broken down by the number of members in the hierarchies, and whether the root of the hierarchy implements an interface.

For example, the first column indicates that there are just under 1200 hierarchies with 2-7 members where the root implements no interfaces. Of these hierarchies, 487 show strict polymorphism (light blue) in which all method invocations are via the hierarchy root type.

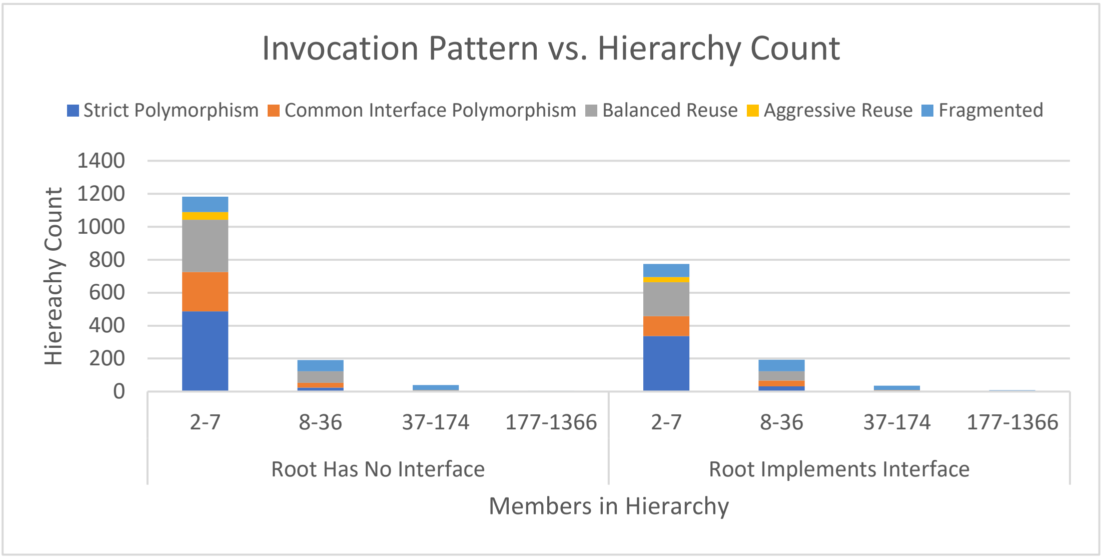


Figure 51 : Invocation Pattern vs Hierarchy Count

The invocations via interfaces are not counted here, so hierarchies where there the root implements an interface have been segregated for clarity. Figure 51 shows that this makes surprisingly little difference in the profiles – the reason for this is not clear.

54% 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, 47.1% are accounted for by fan and line hierarchies. This illustrates that larger hierarchies tend to be in the non-polymorphic categories indicating that these hierarchies are perhaps more driven by reuse.

It is important to stress, as above, the skewing effect of large hierarchies on what might be viewed as 'normal'. Figure 52 shows the same plot as Figure 51, except that the sizes (member count) of the hierarchies in each category are used. This illustrates that most inheritance members are in fragmented hierarchies which do not have uniform use of polymorphism use across hierarchy members.

Summary of Method Invocation: [KF 5.28] Method invocations on hierarchy roots indicate that just over half of hierarchies appear to be used exclusively for polymorphism. [KF 5.29] Method invocation patterns indicate that larger hierarchies have inconsistent 'use' among their members.

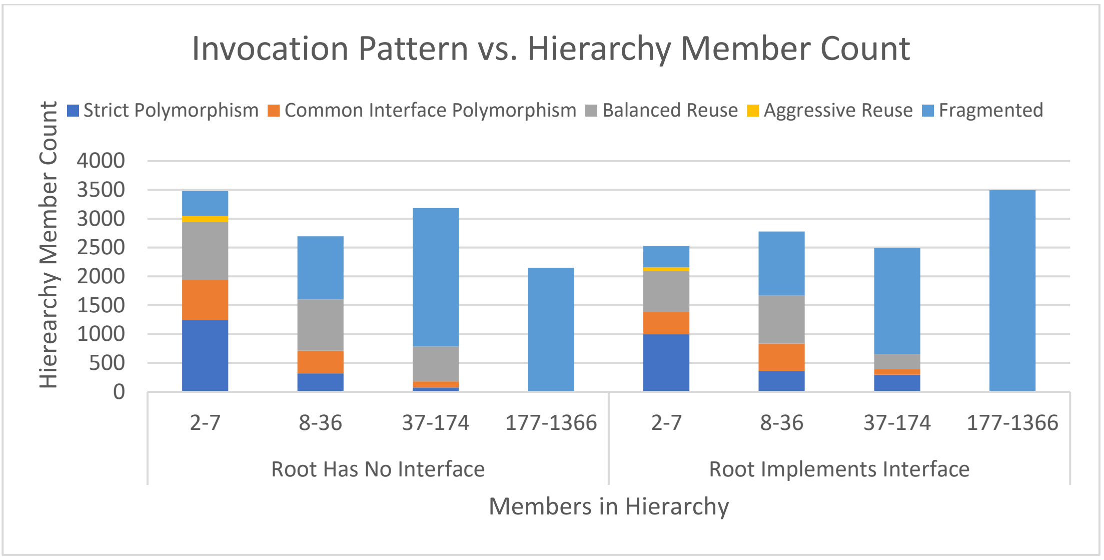


Figure 52 : Invocation Pattern vs Hierarchy Member Count

5.5.7.1 Casting

One of key properties of inheritance in Java is polymorphism – the ability to have multiple objects play the same 'role' without the need to know the raw type of the object. Casting might be viewed as the opposite operation – the practitioner asserts in code that an object of one type *can* be used as an object of another type. Casting can be used to gain access to object properties that are hidden by abstractions – such as bypassing an interface to retrieve a raw type.

Across the systems there are 127736 cast operations. Of these, 80565 (63%) involve an inheritance hierarchy member. Across the examined systems, 4-36% (median 20%) of hierarchy members are involved in at least one cast. This proportion of casting-participation is similar if other population features such as depth in hierarchy, % of novel methods are considered. This indicates that casting may simply be another 'fact of life' where inheritance is used.

Of these 80565 casts, some are from one hierarchy member to another in the same hierarchy (21%, 1659). Other notable groups are casts are *into* a hierarchy from outside (6%, 2769), out of a hierarchy (4%, 1846). The remaining casts are between hierarchy members that are not in the same hierarchy (37%, 29448), on examination, many of these involve interfaces implemented by all of part of the hierarchy. There appears to be a high level of substitution 'traffic' between library super-types, interfaces, and user defined hierarchy members that would require much more detailed study to characterise.

Figure 53 shows a breakdown of casting for inheritance hierarchy members at each depth. For example, the first column shows that there are just over 1000 cast-participant inheritance hierarchy members at depth zero (hierarchy roots), of these, most 51% (orange, 535) are *cast to* some other type, 17% (blue, 182) have been *cast from* another type, and finally 31% (grey, 329) have been involved in casting in both directions. The yellow line is scaled against the right-hand side axis and shows the average number of cast operations on cast participants at each depth. Note that there are no hierarchy members at depth nine which are party to any cast operations.

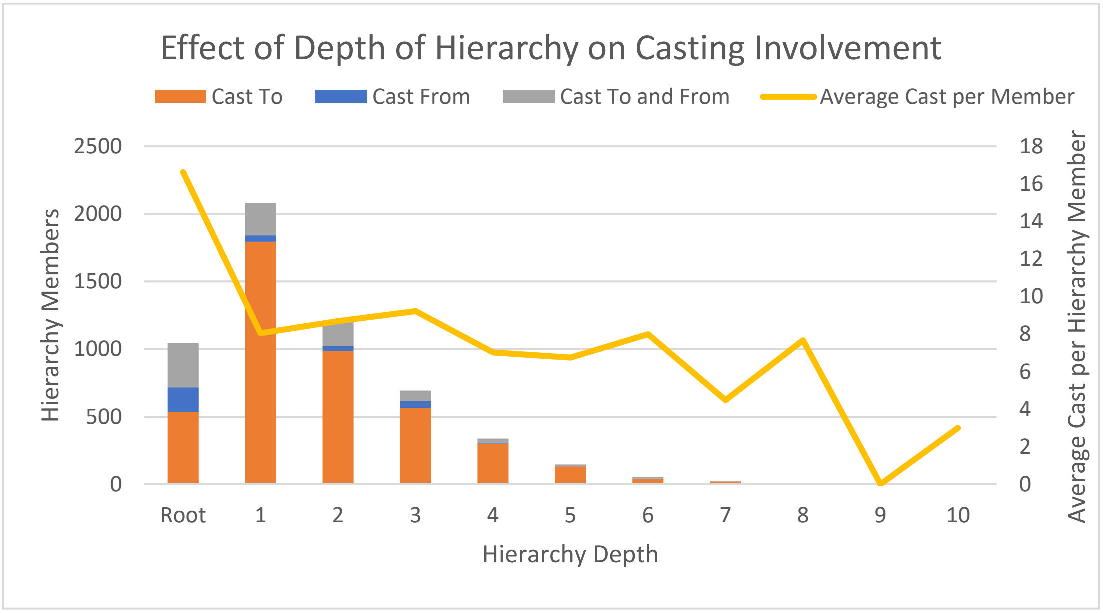


Figure 53 - Effect of Depth in Hierarchy on Casting Incidence

There is a marked decrease in proportion of types involved in casting among the root types (depth zero), which corresponds to fewer root types – this is reasonable given the root is the narrowest part of the hierarchy. However, there is a notably high amount of average cast operations on root types – both in terms of the type (to, from, both), and the number of casts. This indicates that abstractions are frequently bypassed, suggesting that they may be insufficient for meeting the needs of hierarchy clients in these systems.

Overall, most hierarchy members participate in 'cast-to' operations, in the case of the root it is not clear why this would be the case – the only conversion that would require a cast to a root type are interface or *java.lang.Object* types are being *downcast*. On the other hand, casting may be used for documentation purposes. For non-root types, the case is perhaps clearer although it is notable that there is no sign of large numbers of casting operations in deeper hierarchies.

Summary of Casting: [KF 5.30] There is a lot of casting in the systems examined, including a large number related to inheritance hierarchies. [KF 5.31] Root types are slightly less likely to be involved in a cast, but if they are, they are involved in about twice as much casting on average compared to non-root types.

5.5.7.2 Large Hierarchies

This section reports on the largest hierarchies and what observations can be made about them.

The largest hierarchy examined is `EventManager` (DIT 10) in eclipse, which is similar to the `java.util.Observable` class supplied in the standard Java libraries – though the documentation of `EventManager` notes - "*This handles the management of a list of listeners -- optimizing memory and performance. All the methods on this class are guaranteed to be thread-safe.*" – these additional constraints explain why the library class was not used. The developers of the eclipse system have detected an issue with such a large and sprawling hierarchy – as one recommends:

"*Deprecate org.eclipse.core.commands.common.EventManager. This class is unnecessary and promotes an "implementation inheritance" pattern that is wrong in so many ways.*"

(Keller, 2016)

They also note that "*EventManager is the superclass of many API classes*", so any future improvements are constrained by these dependencies. The structure, is depreciated and documented to discourage future reuse, but remains in the design.

The next deepest hierarchies (DIT 8) have the roots `org.eclipse.jface.viewers.Viewer` (15 root methods) and `org.eclipse.jface.window.Window` (18 root methods). While these hierarchies are not quite as large (85 and 361 types respectively), they do have similarities to `EventManager`.

- They are deep and wide compared to the rest of the corpus.
- They have a distinct, continuous, backbone of abstract classes comprised of most of the abstract classes in the hierarchy (83-88%), and their root type is an abstract class.
- Method invocation is fragmented (see Table 33) – with some subtype methods being invoked directly, and others only via root methods.

However, there are differences. The `Viewer` and `Window` hierarchies have fragmented method invocation and increasing method overriding as depth increases– indicating functional divergence from the root type. Contrast this with `EventManager`, which has no overriding of root-defined methods. These examples indicate that hierarchies can grow very large with or without substantial amounts of overriding. A factor to consider is the amount of behaviour that is available *to* override, `Viewer` and `Window` have 15 and 18 public methods respectively, while `EventManager` has 5 protected final methods – meaning that these cannot be overridden. This may reflect the approach proposed by Bloch where practitioners should "*design and document for inheritance or else prohibit it*" (Bloch, 2008).

There are 48 sub-tree hierarchies of depth 5 or greater (of 377 total) – these include the top half of hierarchy members in sub-trees structures (7244 of 14338 types). These most complex hierarchies have been manually inspected to identify any commonalities in the purpose or characteristics in these large structures. If there is no obvious mapping to the property under consideration, the hierarchy is left unassigned, in line with the manual classification process used by Steimann and Meyer when classifying interfaces against a theory-based taxonomy (Steimann and Mayer, 2005). A full listing is shown in Table 34, note that the large hierarchies are listed in order of the number of hierarchy members. Initially, the abstraction represented by the root type is considered in relation to guidelines:

- *Generalisation* (12) – The root type forms the top of a recognisable is-a relationship. Sometimes called a partial type (Halbert and O'Brien, 1987).
- *Grouping* (6) – The root abstraction is small and well-defined, but does not define the subtypes and there is lots of semantic variation in the hierarchy (Steimann and Mayer, 2005).
- *Cross Cutting Concern* (8) – an aspect or trait that is more associated with a concrete implementation than a Grouping root type – representing an implementation detail or architectural abstraction.

This spread of uses indicates that generalisation (the textbook polymorphic 'is a' relationship), is a common motivator for large hierarchies. However, many hierarchies also exist to propagate enabling capabilities or simply reuse implementation. Additionally, 12 hierarchies remain unclassified –indicating that these high-level distinctions are not always clear. Next, the overall structure of the hierarchy is examined to determine if there are any patterns or architectural aspects to the hierarchy:

- *Context* (DCI) (1) – hierarchy members resemble Contexts from DCI – enactment of a use case (Reenskaug and Coplien, 2009).
- *Event/Message* (2) – subtypes are messages, where type variation (names) carries semantic information, rather than variations in behaviour.
- *Mirror* (4) – the hierarchy mirrors another structure, defining members (e.g. decorators, visitors) based on the other structure which is usually another inheritance hierarchy.
- *Pattern* (18) – the hierarchy structure is defined by an obvious architectural or design pattern. (Adapter (1), Composite (11), Observable (3), Pipe and Filter (2), Strategy (1))

Over half of the large hierarchies (25/48) examined have a recognisable pattern – which may make understanding and maintaining the hierarchy simpler. However, these require reviewing the entire hierarchy in some cases, for example, mirror hierarchies are more comprehensible if the reader has a passing familiarity with an additional external model. Note that composite structures are the most common patterns in large hierarchies by some way – demonstrating that *whole-part* relationships are commonly modelled using inheritance. This does highlight that the structure, purpose, and trade-offs relating to many large hierarchies may not be recognised by a practitioner without training in design patterns.

The manual examination of the large hierarchies revealed two further high-level motivations for the creation of hierarchies – again these are shown in Table 34. Firstly, as a *Framework Hook* (12), demonstrating that framework extension points *are* being used, and can form the basis of substantial-sized hierarchies. The second conspicuous influence has been labelled *Specification/API Tied* (12) – where the structure of the hierarchy is bound to some external standard – indicating that the hierarchies represent a highly structured problem domain e.g. 5 of these are Composites representing structures such as document standards. These are of note as there may be an obvious mapping form the domain to particular implementations (such as inheritance or design patterns) which may simplify design-decision making.

Table 34 : Summary of inspection of large hierarchies (DIT >=5)

						Root			Structure									
Types in Tree	**Source System**	**Root Type**	**depth**	**width**	**Abstraction**	Cross Cutting Concern	Generalisation	Grouping	Adapter	Composite	Context (DCI)	Event/Message	Mirror	Observable	Pipe and Filter	Strategy	**Extends Framework Hook**	**Specification / API Tied**
20	argouml-0.34	org.tigris.gef.presentation.FigText	6	6	GUI elements												X	
21	Gizmoball	java.util.Observable	5	5	Enable observer pattern	X								X				
21	aoi-2.8.1	buoy.widget.CustomWidget	5	11	GUI elements												X	
24	columba-1.0	javax.swing.tree.DefaultMutableTreeNode	5	8	Mutable tree nodes					X							X	
25	eclipse_SDK-4.3	org.eclipse.jdt.apt.core.internal.declaration.EclipseDeclarationImpl	5	7	AST declaration nodes								X					Java AST
28	eclipse_SDK-4.3	org.eclipse.ui.internal.intro.impl.model.AbstractIntroElement	5	8	Start-up configuration for plugin													Plugin configuration
29	eclipse_SDK-4.3	org.eclipse.jdt.internal.core.JavaElementInfo	6	9	Java language elements			X										
31	JHotDraw_7.0.6	org.jhotdraw.draw.AbstractFigure	5	12	Figure in drawing canvas					X							X	
31	eclipse_SDK-4.3	org.eclipse.jdt.ui.text.java.correction.ChangeCorrectionProposal	5	17	Context sensitive change proposal		X									X		
34	java_8u60	javax.swing.text.View	6	10	View in MVC - lightweight proxy for model entity					X							X	
37	java_8u60	com.sun.org.apache.xerces.internal.dom.NodeImpl	5	12	Document object model					X								DOM
40	eclipse_SDK-4.3	org.eclipse.jdt.internal.compiler.lookup.Binding	5	15	Type bindings for compiler					X								Compiler notation
40	java_8u60	java.io.OutputStream	5	15	Writing bytes		X								X			
40	java_8u60	org.omg.CORBA.LocalObject	5	22	CORBA interface definition language object representation													CORBA IDL
43	java_8u60	java.io.InputStream	5	17	Reading bytes		X											
51	eclipse_SDK-4.3	org.eclipse.pde.internal.core.text.DocumentXMLNode	5	29	XML document nodes					X								
53	freemind-0.9.0	freemind.extensions.HookAdapter	5	26	Plugin 'hook' implementations	X												Plugin API
56	java_8u60	java.util.AbstractCollection	5	29	Skeleton for collections implementations		X											
56	eclipse_SDK-4.3	org.eclipse.jdt.core.dom.ASTVisitor	6	31	Visitor companion hierarchy			X					X					
60	eclipse_SDK-4.3	org.eclipse.core.runtime.jobs.JobChangeAdapter	7	19	Adapter to implement JobChangeListener			X										
62	argouml-0.34	org.tigris.gef.presentation.FigNode	5	23	Diagram elements		X										X	
65	eclipse_SDK-4.3	org.eclipse.core.databinding.observable.ChangeManager	5	35	Making things observable	X								X				

65	eclipse_SDK-4.3	org.eclipse.emf.ecore.impl.MinimalEObjectImpl.Container	6	18	Wrapper			X	X								X	
72	eclipse_SDK-4.3	org.eclipse.jetty.util.component.AbstractLifeCycle	7	20	Has lifecycle start, end, and stages	X		X									X	
74	eclipse_SDK-4.3	org.xml.sax.helpers.DefaultHandler	5	40	XML parser/generator												X	XML
76	eclipse_SDK-4.3	org.eclipse.ltk.core.refactoring.Change	5	25	Change to UI or data element						X							
76	java_8u60	java.util.EventObject	5	32	Message to be passed		X					X						
84	eclipse_SDK-4.3	org.eclipse.swt.internal.mozilla.nsISupports	5	64	Platform neutral commands												X	Mozilla NSL
85	eclipse_SDK-4.3	org.eclipse.jface.viewers.Viewer	8	16	Adapts model to viewable widget		X											
87	argouml-0.34	javax.swing.JPanel	5	37	GUI elements												X	
97	eclipse_SDK-4.3	org.eclipse.core.databinding.property.value.ValueProperty	5	48	Program administration			X										
99	eclipse_SDK-4.3	org.apache.lucene.util.AttributeSource	5	78	Language processing filters										X			
103	java_8u60	com.sun.org.apache.xpath.internal.Expression	6	34	XPath expressions					X								Xpath standard
106	java_8u60	java.awt.Component	6	42	UI non-menu components					X								
109	azureus-4.8.1.2	org.bouncycastle.asn1.ASN1Encodable	5	56	Encryption library component												X	
110	java_8u60	com.sun.org.apache.xalan.internal.xsltc.compiler.SyntaxTreeNode	6	57	XSLT expressions					X								XSLT
141	freecol-0.10.7	net.sf.freecol.common.model.FreeColObject	6	57	Save/loadable entity	X												
142	eclipse_SDK-4.3	org.eclipse.ui.forms.AbstractFormPart	6	62	Part of a form, mainly concerned with saving, validating and updating data		X											
157	eclipse_SDK-4.3	org.eclipse.swt.widgets.Widget	7	58	UI element		X											
177	eclipse_SDK-4.3	org.eclipse.jdt.internal.compiler.ast.ASTNode	7	44	Java abstract syntax tree					X								Java AST
238	java_8u60	javax.swing.plaf.ComponentUI	6	93	Pluggable look and feel								X					
361	eclipse_SDK-4.3	org.eclipse.jface.window.Window	8	135	Window inside main UI		X											
373	ant-1.8.4	org.apache.tools.ant.ProjectComponent	6	155	Component in a project	X												
449	eclipse_SDK-4.3	org.apache.tools.ant.ProjectComponent	6	200	Grouping			X		X								
460	eclipse_SDK-4.3	org.eclipse.jface.dialogs.DialogPage	6	183	Dialogue page		X						X					
620	eclipse_SDK-4.3	org.eclipse.core.runtime.PlatformObject	7	186	Adaptable object	X												
650	java_8u60	java.lang.Throwable	5	290	Exceptional event for JVM		X					X						JVM specification
1366	eclipse_SDK-4.3	org.eclipse.core.commands.common.EventManager	10	542	Observable entity	X								X				

There is no strong relationship between the semantic characteristics presented in Table 34 and the shape or usage characteristics described in the other sections. Most (45) of the large hierarchies show a Fragmented or Balanced Reuse invocation profile, with only the most generic hierarchies (container and adapter) having Common Interface Polymorphism. Most (36) of the hierarchies also show a mixture of overriding and new methods.

Summary of Large Hierarchies: [KF 5.32] Reuse and substitutability appear to be strong influences for some large hierarchies, but not all. [KF 5.33] Many large hierarchies have strong external motivators such as design patterns, other design structures, or structures in the domain, which may determine final shape more than practitioner choice. [KF 5.34] There is no indication that the external motivations correspond to the empirical properties of hierarchies, indicating that understanding of both is required to assess a hierarchy.

5.6 Discussion

This section discusses the results in relation to the research questions and surrounding literature and relates the findings to previous work.

5.6.1 RQ5.1: How is inheritance used?

Inheritance is present in large amounts in the corpus *[KF 5.1]* – inheritance is in common use and is thus important to software construction and design. The average levels of definition-using-inheritance are like those found in previous studies *[KF 5.2]* - around *three-quarters* of types are defined using some form of inheritance. However, systems achieve similar levels of definition-by-inheritance in different ways *[KF 5.4]* - smaller systems have proportionally more type-definition using inheritance, but prefer interfaces; while larger systems define proportionately fewer types using inheritance, but are more likely to use concrete inheritance when they do. Variation between systems calls into question the value of corpus-wide averages *[KF 5.8]*.

Use of inheritance enables logical substitutability and optional reuse. Based on method invocations to hierarchy members, just over half of hierarchies are only accessed via the root type *[KF 5.28]* indicating substitutability is a key motivator for inheritance use. However, the number of novel methods present indicate that reuse is also a common motivation for inheritance use *[KF 5.15]* since these novel methods must be accessed via a direct reference to the raw type or indirectly via other methods.

Much of inheritance use is trivial. Many types are only extended one or two times in most cases *[KF 5.3]* indicating that low-level reuse is common, but that few individual types are extended (directly or indirectly) many times. Most hierarchies are shallow *[KF 5.5]*, and most hierarchy members offer a small number of methods *[KF 5.11]*, with most hierarchy members having < 10 methods. Most inheritance hierarchies are also simple – many variations on a single type (fan), or repeated specialisation without branching (line) *[KF 5.18]*. However, there are major differences within the population *[KF 5.8]*, with much more variation in width than depth *[KF 5.7]*, and some outlier hierarchies growing very deep and wide *[KF 5.6]*.

Outlying hierarchies are noticeably large and wide compared to most hierarchies and guidance on depth *[KF 5.6]*. Deeper hierarchies are more common as system grow larger, while hierarchy width appears to be less constrained by system size *[KF 5.9]*, growing by an order of magnitude over depth in many cases.

5.6.2 RQ5.2: How can inheritance hierarchies be characterised?

The most obvious influence on large hierarchies are patterns from the problem or solution domain *[KF 5.12, 5.33]* 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-use strategy.

Relatively few hierarchies contain over half of all hierarchy members *[KF 5.21, 5.22]*. This skews the apparent presence of large branching hierarchies – it is not the case that these large hierarchies are common, but they *are* more likely to be encountered. So, strategies or guidance for dealing with large hierarchies are, in fact, targeting exceptions – it is not clear if these guidelines are equally applicable to smaller hierarchies.

Hierarchies can be categorised by shape (line, fan, line-branch, branch-line, sub-trees) – which are present in similar proportions across the examined systems *[KF 5.17]*. Many hierarchies are simple (line and fan) *[KF 5.18]* reflecting again that much inheritance is trivial *[KF 5.5, KF 5.3]*, and that this distribution of inheritance use is common. The high presence of *fan* shaped hierarchies indicates that hierarchies can grow arbitrarily wide *[KF 5.19]*. Indeed, deeper hierarchies without branching are rare *[KF 5.20]*, and very deep hierarchies always branch. The different shapes grow in different ways, so any notion of size must account for the type of hierarchy being considered. Similar distributions of type-extension are found in medium and large systems indicating that some guidance may be less useful when assessing smaller systems *[KF 5.4]*. Many of the largest hierarchies appear to be strongly guided by an external abstraction *[KF 5.32, 5.33]*.

Method invocations indicate that just over half of hierarchies are invoked exclusively via references to their root type *[KF 5.28]*. Again, some larger hierarchies show inconsistent patterns of invocations – with references to subtypes being used in some cases *[KF 5.29]*. In addition, there is a substantial amount of casting in the examined systems, much of which involves inheritance hierarchy members *[KF 5.30]*. Root types involved in casting have twice as much casting as non-root types *[KF 5.31]*. The fact that some attempts to 'program with abstractions' must be bypassed, indicates some mismatch between client and abstraction.

Large hierarchies have less consistent patterns of method invocation among members, indicating varying degrees of non-polymorphic access *[KF 5.29]*. Overriding declines with depth in a hierarchy, while novel methods continue to be added *[KF 5.13, 5.16],* indicating that initial subtypes tend to refine root behaviour, and deeper subtypes tend to add behaviour. In contrast, hierarchy members with no-novel methods are usually in shallow hierarchies *[KF 5.14]*. While method invocation patterns indicate reuse- and polymorphism-motivations as noted above, upon closer examination, some large hierarchies are not easily classified in this way *[KF 5.32]*.

Only one system appears to be built in an inheritance-avoiding style (azureus), and this still contains a large amount of inheritance. Azureus does have a great deal of interfaces – it not clear if these are included as a separate phenomenon from the low amount of inheritance, or if the extra interfaces substitute for some use of inheritance. As noted previously, there are synergies between inheritance and interface use as interfaces and their implementations can be propagated cheaply within inheritance hierarchies.

The use of abstract classes to form structures within hierarchies is common *[KF 5.23]*, but not universal. This abstract structure is generally observed to be continuous within a hierarchy or not present at all *[KF 5.25]*, though in very larger hierarchies, this structure becomes fragmented *[KF 3.26],* indicating lack of consolidation.

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 smaller structure with inconsistent patterns of overriding may indicate a poorly defined abstraction. Assessing local regularity may be more useful than comparing numerical values *between* hierarchies. This may provide guidance for interpreting the purpose of a hierarchy, allow consolidation of common functionality, facilitating extension, and determining if composition is a better option.

5.6.3 RQ5.3: To what extent can inheritance usage patterns be related to Design Quality where quality is defined via Guidance, Metrics, Code Smells, Entity Population Models, and Modelling?

Practically all the guidance regarding inheritance is in the form of 'what not to do' so evidence of compliance would be in the form of minimising unjustified breeches of these guidelines. Closer examination reveals that there is variation in the use of inheritance between and within systems *[KF 5.2]*. Using simple, direct measurements it is possible to illustrate that that most inheritance use is trivial *[KF 5.5, 5.18, 5.22]*. Inheritance represents an effort-saving language feature and the presence of many trivial hierarchies indicates that there is little design risk (perceived or actual) for most inheritance hierarchies.

Guidance regarding depth does not reflect usage of inheritance 'in the wild' *[KF 5.6]*. Most hierarchies are well below the suggested limits on depth (DIT 3-5) *[KF 5.5]*, although outliers grow very deep and wide *[KF 5.6]* so depth limit suggestions are not suitable for either of these groups. Definition by inheritance (interfaces and inheritance) only displays a 'regular' rate of use when considered at the highest level *[KF 5.2]*, but in general interfaces and inheritance are not assessed together – guidance treats these as 'separate tools'. It appears that system size may restrict some properties *[KF 5.9]* with small systems accommodating wider, but not deeper hierarchies. Additionally, some consistent patterns of inheritance use only emerge in medium or large systems *[KF 5.4]* indicating that models may have to be adjusted for system size in some cases.

Guidance regarding consolidation of wide hierarchies is not followed in many cases *[KF 5.10]*. The wide hierarchy smell is linked to 'lack of consolidation' or missing intermediate types – however the range of hierarchy widths observed varies wildly *[KF 5.19]*. In addition, the 'fan' shape is common *[KF 5.18]* – indicating that extracting intermediates is either not possible, or regularly left undone.

In line with the dependency inversion principle and depending towards abstractions, abstract classes are often present in hierarchies, and often as a continuous 'backbone' within the hierarchy *[KF 5.23]*. There appears to be somewhat of an all-or-nothing approach to use of abstract classes *[KF 5.25]* – possibly indicating practitioner style (Chow and Tempero, 2011). Large hierarchies make proportionately less use of abstract classes *[KF 5.24]* - since larger hierarchies tend to be composed of proportionately more leaves, and leaves are generally concrete classes. Additionally, as hierarchies grow very large, the abstract backbone tends to become fragmented *[KF 5.26]*, while this fragmentation is not severe *[KF 5.27]*, this does indicate non-compliance with DIP.

Some hierarchies are constrained to mirror highly specified external structures *[KF 5.12]* such as document standards. Similarly, there are clear examples of large design-pattern driven hierarchies *[KF 5.33]*. There is no indication that these external motivators correspond to internal measurements indicating that understanding of both is required to assess a hierarchy *[KF 5.34]*, and more importantly that a strong motivation can override lower level guidance.

Casting is common in the examined systems, including a notable number of casts involving inheritance hierarchy members *[KF 5.30]*. The casts commonly involve root types – root types involved in casting show higher levels of casting *[KF 5.31]*. This indicates that the root abstractions are commonly being bypassed, and may mean that reuse has come at the cost of poor abstraction. Though some designs use casting to translate between API/library types and more local types.

There is a rule of thumb noted by Beck that three similar classes suggest a missing super-class (Beck, 2014). Since most types that are extended are only extended once or twice *[KF 5.3]*, this this rule is evidently not the cause of many instances of inheritance use, but creating a common super-type from three classes seems reasonable given that it is often done for one or two types.

It is obvious that simple inheritance is used in many cases *[KF 5.1, 5.5, 5.18]*. From the literature, the focus on inheritance related problems are related to size and complexity, and conceptual/modelling problems. Some of these smells cannot manifest in small hierarchies (deep or wide hierarchy), or are unlikely to (multipath or cyclic hierarchy). It is evident that practitioners are freely using inheritance when it is in the form of simple hierarchies.

There is also evidence that the structure of many of the larger hierarchies are strongly guided by a design pattern or well-defined domain entity, such as a document standard, or communication protocol *[KF 5.12, 5.33]*. These hierarchies can grow very large but are self-consistent because the external model has a suitable structure such as hierarchical, whole-part, or composite. It would be difficult to justify not using inheritance in these cases, however the guidance indicates otherwise. This suggests that a clear external motivation or model may preclude recourse to guidance, as the way to proceed is 'obvious'.

Meyer's taxonomy for inheritance (Meyer, 1996) describes three broad categories of inheritance use, depending on the model inspiring the inheritance hierarchy – model inheritance (is-a relationships in the problem domain), software inheritance (solution domain relationships), variation inheritance (similarity or difference relationship). There is some evidence that model and software inheritance are present in high amounts in the large hierarchies examined *[KF 5.33]*, 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 (English et al., 2005).

5.6.4 Presence of Inheritance

It is confirmed that all the observed systems are using large amounts of inheritance *[KF 5.1],* with levels similar to those in large code surveys (Tempero et al., 2008). However, inheritance appears to be used in different ways. These system level inheritance counts hide internal differences - for example *JDK* and *aoi*, have similar proportions overall of non-inheriting types (28% and 29% respectively). When examined in more detail, the systems have different internal breakdowns, with *aoi* favouring extension (class inheritance), while *JDK* is more evenly split between extension and implementation (class inheritance and interface implementation). Systems with broadly the same amount of *abstraction* achieve this in different ways.

Some inner proportions of the systems vary independently of system size (number of user defined types), while other features appear to be consistent across systems *[KF 5.3, 5.4, 5.9, 5.17]*. Some previous work has suggested that some large-scale trends in systems such as coupling are unavoidable (Potanin et al., 2005; Taube-schock et al., 2011), in contrast Chow and Tempero note that "*developer culture*" can have a large impact (Chow and Tempero, 2011). While variation can be attributed to developer variation – it may also be the case that similarities between systems are due to consistency in developer style or training.

Similarly, while most of the systems display all the categories of use – width and depth *[KF 5.5, 5.7]*, shape *[KF 5.17]*, abstract classes *[KF 5.25]*, the actual proportions of use vary. For example, the smaller systems have fewer abstract classes overall which may be expected as fewer types means less common code to consolidate. Gizmoball and JHotDraw systems are known to be intensely designed (Ferreira et al., 2012). However, this has not resulted in any noticeable common features at this level of analysis. This indicates that more nuanced definition of 'good design' may be required.

Most of the systems in the corpus have similar ranges for use of the *extends* keyword *[KF 5.3]* – if the frequency of extension is compared, all systems have frequently extended types in the top 10% (5-172 extensions), and most into the top 1% (18-172 extensions). Despite being very large, *eclipse* has similar proportions to, say, *argouml* which is an order of magnitude smaller. Most systems contain at least one very highly extended type - indicating that all examined systems contain at least one hierarchy that conflicts with the guidance on width (Suryanarayana et al., 2015).

Other than being a little top heavy, the framework-like *JDK* is not distinct in the distribution of extended types, and has the same proportions as *aoi*. Frameworks are often treated separately in software corpus studies on the basis that they are structured differently from *full applications*. While frameworks are not full applications, they often contain a detailed domain model – this may be sufficient to make a framework *application-like* for the purposes of some static analyses, such as this one.

Guidance indicates that abstract types should be used to contain common code among subtype *peers*. However, there is also advocacy for a *wait and see* approach to determine if shared code is truly a common abstraction or simply coincidental. If, as is indicated, most types that are extended, are not extended very much, then confirmation for the proposed abstract class (in the form of more sibling types) may never arrive. This may give some credence to Foote's proposal of automatic (and pre-emptive) abstract classes for any non-leaf role in a hierarchy (Johnson and Foote, 1988).

5.6.5 Depth of Inheritance

Overall, hierarchies with depth one or two account for over 79.9% of all hierarchies *[KF 5.3]*, so from this point of view, all deeper hierarchies could be viewed as outliers - as Tempero notes "*most inheritance is shallow*" (Tempero et al., 2008). It follows that some attempts to characterise the overall approach to inheritance used in a system might be dominated by this bedrock of shallow inheritance. Similarly, assembling corpuses of many systems may produce mean levels that reflect most common use - but not necessarily the diversity of inheritance use. Depth alone does not capture the range of complexity which can exist at each depth.

The maximum values for hierarchy depth found in this study match many of the maximum values identified in previous studies in both Java and C++ systems discussed in Chapter 2. Given the notable depth of the *eclipse* system, and the popularity of the Qualitas Corpus, this may be the reason for some similar maximum values. Other studies report depth ranges around 1-6, except for *eclipse* and *JDK*, systems in the corpus presented here have a span of depths in this range. This indicates that the "*rule of thumb*" of five (SonarQube, 2013), or six suggested by the guidance ((Suryanarayana et al., 2015)) is adhered to in most cases – though it is unclear if this is a natural limit or a result of practitioner restraint.

Most hierarchies are very shallow *[KF 5.5]*, so a limit of six gives does not address size issues that might occur if most hierarchies double or even triple in size. Additionally, there are a substantial number of large (depth >6) well-formed hierarchies *[KF 5.12, 5.33]* which do not comply with this guidance. This guideline is based on observation, and maintainability considerations. Experimental work on maintainability investigated inheritance depths of 3 and 5 (Cartwright, 1998; Daly et al., 1996; Harrison et al., 2000), which suggest that depth cannot be reliably related to maintainability issues. This indicates that factors other than depth affect maintainability. Indeed, work on hierarchy design (Dvorak, 1994) suggests that the arrangement and relatedness of members within a hierarchy is a more important consideration.

5.6.6 Hierarchy Width

As with depth, most hierarchies are narrow *[KF 5.18]*, although most of the systems have some extremely wide hierarchies – small systems appear to accommodate hierarchy width more easily than hierarchy depth *[KF 5.9]*. Even *azureus*, which has low inheritance use overall, has a few wide hierarchies.

Width is not discussed much in guidance as such – this may be because increasing re-use or polymorphism *must* increase the number of leaves in a hierarchy. Guidance does, however, have something to say about *duplication*, both generally (Don't Repeat Yourself), and specifically in relation to hierarchy members. Johnson and Foote suggest that common features in subtypes should be extracted to intermediate subtypes (Johnson and Foote, 1988), reducing the "*un-factored hierarchy*" smell (Suryanarayana et al., 2015). This *does not* decrease the number of leaf-types as such, but it may simplify their implementations and reduce error, while increasing hierarchy depth. Use of inheritance for such consolidation of common behaviour may be considered 'convenience reuse', rather than proper 'is-a' relationships. Both Liskov and Meyer discourage this type of reuse - which violates the "meaningful classification" requirement proposed by Suryanarayana et al., who identify this violation with the "*broken hierarchy*" smell (Liskov, 1988; Meyer, 1996; Suryanarayana et al., 2015). This study also observes that overriding tends to appear in shallower parts of hierarchies – indicating that shared re-definition of root-behaviour may be being 'pushed up' within the hierarchy *[KF 5.16]*.

This study identifies that wide inheritance structures are not only common *[KF 5.5, 5.18]*, but that in some cases decedents of a single node accounts for most or all of the width of a hierarchy. This indicates that abstractions with many direct descendants is a more common design need than chains of repeated specialisation.

Tempero compares inheritance structures to binary trees to provide a baseline for discussing branching behaviour (Tempero et al., 2008) – this is not an appropriate model. The width of observed hierarchies does not increase regularly with depth as a binary-tree does. Width, especially in very wide hierarchies, is the result of a few nodes with very many direct descendants.

It appears that the natural growth of hierarchies prefers width over depth by an order of magnitude *[KF 5.7]*. Given the lack of restrictions on the Java language, this must to be due to some limit of process (or practitioners), rather than feasibility. Alternatively, it may be that increases in width indicate reuse of existing abstractions is *easy*, while increasing depth requires some extra effort. This echoes the preferential attachment model of system growth where "*existing nodes link to new nodes with probability proportional to the number of links they already have*" (Baxter et al., 2006). However, this appears to conflict with code smell guidance which recommends a limit of nine immediate types before some consolidation action is considered (Suryanarayana et al., 2015).

Tempero describes the deeper hierarchies as *"tall and skinny"* (Tempero et al., 2008). This is reminiscent of Smalltalk guidance proposed by Foote and Johnson, who recommend that "*Class hierarchies should be deep and narrow*" (Johnson and Foote, 1988) – on the basis that a deeper hierarchy structure reuses more code as common functions are *pulled up* the hierarchy. Very wide *fan* structures *[KF 5.18]* observed in the corpus are directly at odds with these arguments from design (see section 5.5.5.1 Shape).

This modest corpus shares some characteristics regarding amount and range of inheritance with larger and more diverse corpora that have been examined in previous large scale studies. It is likely that use of a larger corpus would provide more robust mean figures and perhaps provide more of a continuum of system profiles.

5.6.7 Public Interfaces and Method Novelty

Design guidelines do not have anything specific to say about the size of public interfaces in inheritance hierarchies (root or subtype), even the large class smell is more concerned with the number of class variables (Fowler et al., 1999). However, general object-oriented guidelines encourage that public interfaces expose only necessary behaviour and that objects have a *single responsibility* (Martin, 2005b).

This study confirms that most inheritance hierarchy members have relatively few public methods *[KF 5.7]*. It is thought that aspects of design which are *visible* to a designer (such as the number of methods in a public interface) are less likely to *get out of hand*, due to their obviousness (Baxter et al., 2006). Inheritance hierarchies can contain irregular structures and indirect relationships where cause and effect may be many steps apart – it may be difficult to 'get the whole picture' by looking at one part or member of a hierarchy [KF 5.8, 5.10, 5.24, 5.34].

Only around 6% of hierarchy members retain the same public interface as their root type *[KF 5.13]*, indicating that pure variations on a type represent a minority of the uses for inheritance. This is surprisingly low as polymorphism-driven inheritance is often the textbook example of inheritance hierarchies. The addition of novel (non-root signature) methods in subtypes is common - these *novel* methods begin to dominate after a single level of inheritance. So, in many cases after depth two, the majority of public methods presented to the practitioner may not be part of the *essential* abstraction captured by the hierarchy. It is conceivable that this may obscure the true purpose of a hierarchy when a practitioner encounters the leaf of a deeper hierarchy.

If no methods are being added, subtypes can alter behaviour in other ways - by constructor based configuration, implementing abstract methods, or overriding methods. Hierarchies with no additional methods represent 903 (3.9% of total) hierarchy members. Which means that, a maintainer will rarely encounter a hierarchy member implementing only root methods – when they do that hierarchy member is likely to be at depth one.

By considering the count of methods in hierarchy members, there are a handful of hierarchy members with hundreds or even thousands of distinct methods, however, these are the exception. There is little common ground among these – *JDK* graphic components, *eclipse* OS interface, and key modelling concepts in a game (e.g. unit, player) *freecol*. These represent implementation, facility, and subtype inheritance use respectively from Meyer's taxonomy (Meyer, 1996).

A major complication of this analysis is that the novel behaviours (methods and attributes) may be entirely suitable conceptual expansion to the root type, so simply being *novel* does not mean that the new behaviour represents a conceptual drift from the root type. A high proportion of method novelty may indicate that a root type is conceptually very abstract. This would necessitate adding novel methods to specialise the root. However, since new public methods represent behaviour outside the polymorphic interface offered by the root of a hierarchy, more novel methods indicate that the new behaviour is not related to the polymorphic use. This corroborates that idea that overriding (and thus polymorphic) refinement takes place in shallower parts of hierarchies, while deeper elements add behaviour (see 5.6.2).

5.6.8 Hierarchy Structure

Classifying inheritance hierarchies by overall shape gives a population breakdown that is repeated across most of the systems in the corpus. Most (12 of 14) systems following in order line, fan, sub-trees, branch-line, line-branch *[KF 5.17]*. The popularity of *line* is likely due to the shallowness of most hierarchies. Line structures indicate repeated specialisation – without knowledge of the purpose of the specialisation it is still possible to link this to incremental modification (Taivalsaari, 1996), or local replacement (Liskov, 1988).

Some *line* hierarchies (e.g. *java.awt.KeyboardFocusManager*) indicate that they are *versioning* hierarchies, where the super-types represent 'older' versions of the class, and the single leaf-type is the 'current' version. This is compliant with the OCP (R. Martin, 1996) and indicates a further use for inheritance in design – balancing backwards compatibility with change. Versioning might be considered a sub-set of reuse where some constraint requires preservation of multiple versions of the same type in a design.

Fan structures indicate one super-type with at least two, but often many, direct descendants *[KF 5.10]*. These structures could relate to *grouping* super-types - where the super-type defines some commonly required behaviour, but where "*no attempt is made to relate*" the subtypes (Liskov, 1988). Lack of relation would explain the lack of common behaviour that can be extracted to intermediate subtypes. However, this grouped behaviour is also often used for easy substitutability – so use of the hierarchy members by the rest of the system *[KF 5.28]* is necessary to identify if use of subtypes is solely polymorphic.

Subtrees comprise the largest and most complex hierarchies, the prevalence of very wide branches in large hierarchies *[KF 5.20]* indicates that these highly branching nodes may have particular properties that require examination independently of the containing hierarchy.

In terms of complexity – it may be argued that a *fan* can become arbitrarily wide before introducing too much complexity, because the relationships that must be considered are between one root type and one child type. With a single layer of inheritance and a stable base class, there is a lot of *horizontal freedom* to add more leaves at the same level, without adding much complexity. However, finding and extracting common behaviour is likely to require more effort as the number of subtypes increases.

Johnson and Foote recommend deeper structures to maximise reuse – "*A well-developed class hierarchy should be several layers deep. A class hierarchy consisting of one superclass and 27 subclasses is much too shallow.*" (Johnson and Foote, 1988) The writers imply that common elements are common in a large *fan* hierarchy – "*it is often* *possible to break a method into pieces and place some of the pieces in the superclass and some in the subclasses.*"

The prevalence of fan structures indicate that this is not always done – in some cases there may be no common behaviour – for example in grouping hierarchies (e.g. Serializable). The refactoring literature does make an exception for *protocol* inheritance where there are likely to be many subtypes of a single super-type (Suryanarayana et al., 2015). However, it could be argued that any super-type or interface describes a protocol so this notion, again, requires subjective assessment.

Extracting a common super-type is a trade-off. On one hand, there may be duplicate code among the candidate subtypes. On the other hand, adding a common super-type creates a relationship between subtypes and an expectation of substitutability. If reduced duplication is required *without* substitutability, then composition would be more suitable for meeting this need. This may be one aspect of the guidance to "*prefer composition over inheritance*" (Gamma et al., 1994).

It is not clear if any of the common *fan* structures *[KF 5.18]* are *is-a*, or only exist to indicate common support for "*needed operations*"(Liskov, 1988). Manual examination reveals that there are '*is-a'* relationships in some fan hierarchies (e.g. *AbstractBrowserMessageListener* in azureus). These appear at first appear to be *grouping* hierarchies which Meyer and Liskov do not consider to be 'proper use' of inheritance (Liskov, 1988). However, the subtypes are true variations on a type based on an implementation abstraction. This illustrates *software inheritance* which describes "*relations within the software itself rather than in the model*" (Meyer, 1996). This kind of inheritance can be difficult to identify as the concept represented has no analogue in the problem domain.

It is notable that the large size of these hierarchies is mainly due to the multiple branches *[KF 5.20]* – some of the internal fans alone constitute large hierarchies *[KF 5.19]*. Once again this indicates that the wide hierarchy smell is often not addressed or is not considered to be a problem. This may indicate that the guidance overestimates the problem of wide hierarchies – much like the apparent low-impact of the fragile base class problem in practice (Sabané et al., 2016).

5.6.9 Large Hierarchies

Large sub-tree hierarchies illustrate the skew in the inheritance hierarchy population *[KF 5.21]*. The largest 48 (of 377 total) sub-tree hierarchies are all over depth 5 and contain half of the members in sub-tree hierarchies *[KF 5.21]*. The literature on code smells considers depth 6 to be the limit for *reasonable* hierarchies, this is cited from Riel, which is based on experience and practice. Although Riel does note that "*…some designers have noted that this is due to a lack of tools.*" (Riel, 1996; Suryanarayana et al., 2015).

Large hierarchies are proposed as sources of possible complexity or confusion in the literature, however, on examination there is high level consistency within the largest hierarchies *[KF 5.12, 5.23, 5.33]*. Inheritance hierarchies do not exist in a vacuum – they are subject to constraints and responsibilities which limit and guide their growth:

- 12 of the 48 are constrained to model some external standard, such as a document specification – indicating that the hierarchy is likely to be stable and well specified.
- 25 of the 48 have a clear *design pattern* structure which may constrain growth to be regular.

Just over half of the large hierarchies examined show design pattern influence (usually Composite pattern). There is no small irony in the fact that the most obvious structures in the large inheritance hierarchies are design patterns which urged practitioners to "*Favour 'object composition' over 'class inheritance'.*" (Gamma et al., 1994).

On closer examination - most complex hierarchies have recognisable high-level design motivations – recovering these motivations require understanding the design *[KF 5.33, 5.34]*. There is no indication that very deep hierarchies should be associated with poor design although they may be inherently unwieldy due their size. While there may be some individual poor examples, treating depth as a concern does not reflect use of inheritance in practice.

The presence of these existing design constraints in many of the complex hierarchies may allow easier detection of unsuitable leaf-types. For example – if there is a clear composite structure for most of a hierarchy – it may be worth re-considering if a non-compose-able leaf is being added. This might be considered a special case of the LSP (Liskov, 1988). Furthermore, structural motifs such as design patterns can often be detected automatically.

5.6.10 Abstract Classes

Just over half (54%) of the hierarchies examined contain abstract classes, but there is no observed proportional increase in abstract types with an increase in hierarchy size *[KF 5.24]* – this may be explained by the fact that most hierarchies are dominated by the fringe of leaf nodes (in terms of hierarchy member count).

Where there are abstract types, however, they are often grouped together in a contiguous subtree within the containing hierarchy. While there are a few instances of fragmented or *out of order* (less to more abstract) relationships, these are uncommon. Additionally – all large hierarchies observed have a substantial contiguous structure of abstract classes. This indicates that, regardless of design pressures and local style, a *backbone* of abstract classes emerges when large hierarchies are being constructed *[KF 5.23]*. Consequently, the 'completeness' of the abstract structure of a hierarchy may be one indication of its 'health'. The presence of abstract classes was gathered, but not reported during earlier work (Tempero et al., 2008), there is little other work to compare this with.

This appears to corroborate guidance offered by Johnson and Foote that "*the top of the class hierarchy should be abstract*" and confirm the usefulness of the dependency inversion principle in practice (Johnson and Foote, 1988; R. C. Martin, 1996). Most hierarchies have an all-or-nothing approach to abstract class use (0% or 100%) – with few in-between *[KF 5.25]*. Depth appears to be accompanied by the use of abstract classes, but some *sub-tree* hierarchies show a fragmented abstract class structure. It is not clear if the abstract class backbone in these deeper, branching types of hierarchy are aggregating or disintegrating over time.

5.6.11 Method Invocation

While there is no direct means of identifying the intent of the practitioner – there are some hints as to the eventual usage of inheritance structures by the rest of the design. Based on method invocations by the rest of the design, hierarchies that are primarily polymorphic (public root methods only, invoked on root or other hierarchy member) in terms of use can be identified – these account for 54% of hierarchies. Most of these hierarchies (47%) are accounted for by fan and line hierarchies indicating that purely polymorphic hierarchies are usually shallow.

The remaining hierarchies show varying levels of direct access to public methods that are not root-defined, indicating that the use of inheritance may be motivated more by the desire for reuse or occasional compliance (the super-type is a partial interface). Of the 758 hierarchies which have the *fan* shape, 38% (287) are accessed (method invocations) exclusively via the root type of the hierarchy which suggests that it may be possible to isolate archetypical modes of inheritance use which may simplify comprehension and navigation, even in very wide hierarchies.

While this is not practical for all hierarchies, it may provide a way to *triage* hierarchies during review or familiarisation to illustrate to the practitioner how *internally consistent* a hierarchy is. This would mesh will with the principle of least astonishment – to confirm to the practitioner, at a glance, that there are no unusual corner cases or inconsistencies in a hierarchy.

5.6.12 Casting

Casting is common in all systems indicating, perhaps, that more guidance is required for this mechanism. This indicates that circumventing the root abstraction is relatively common - despite being discouraged by guidance.

Only 6% of casts are casts into a hierarchy - the ‘cast from’ types in these cases are related to hierarchy members but do not have access to the hierarchy – references to very general types such as *java.lang.Object,* or to interfaces implemented by hierarchy members.

More interestingly perhaps is the traffic within the hierarchy (21 % of hierarchy member facing casting) – since inheritance members may be freely substituted for one of their super-types without a cast, casting within the hierarchy may indicate casting *down* or *sideways* within the inheritance hierarchy. This can be considered a *breach of encapsulation* as the programmer is ‘looking past’ an abstraction to get information that has been deliberately hidden. Switching on type is usually associated with the Missing Hierarchy code smell (Suryanarayana et al., 2015), in this case a hierarchy is present – so this may indicate that behaviour is not properly distributed within the hierarchy (i.e. Unfactored Hierarchy smell).

Finally – some systems have a few instances of casting from inside a hierarchy to a type outside of that hierarchy. Notable outliers are *aoi* and *java* (and *eclipse* to some extent). These casts indicate that access to the inheritance hierarchy member via its native interface is not sufficient at the point of use. Examination of these casts indicates that they are casts from a more general type within the application or java libraries and are cast to *translate* them back to the local implementation e.g.

```
artofillusion.object.MeshVertex -> artofillusion.object.TriangleMesh.Vertex
java.lang.Thread -> org.eclipse.jface.operation.ModalContext.ModalContextThread
```

These examples show that different parts of the host systems are operating at different levels of abstraction, where the more general type is useful for some operations, but the abstraction must be broken at some point retrieve the more specific local type. The high proportion of this category in the java libraries may reflect that operations offered by the language libraries operate on partial abstractions e.g. generic operations on collections, traversing syntax trees.

So, casting is not a rare or special event, as the general tone of the guidance suggests – casting can be error prone "*uncheckable run-time dependency*" (Gurp and Bosch, 2001), and be based on unsafe assumptions (Eichberg et al., 2015). There has not been a great deal of research into the use of casting – so there is no easy way to compare these findings with previous work. It appears that the types and hierarchies in these systems are unable to meet the requirements made of the abstractions. It is not clear, if this is a general weakness of Java, OO design, or if high rates of casting indicate poor choice of abstractions.

5.6.13 Conclusion

Inheritance is more common than the current guidance such as 'program to an interface' would indicate. In line with previous work, this study confirms that a high proportion of types are defined using inheritance.

The systems all deviate from high level trends averages in different ways as might be expected from conventional metrics, there is no sense of uniform distance from the average, and growth of inheritance use is not consistent between systems. This indicates that system level measures or averages cannot meaningfully represent the population.

However, using various features of inheritance hierarchies and their members, this study found that there *are* distinct structural motifs that show remarkable similarity between systems. These are based solely on design structure and require no manual classification – an issue which often plagues this sort of design analysis.

Several distinct sub-populations of inheritance were observed – based on shape, method invocation patterns, and method introduction and overriding. These may map to use-intention or design needs. More importantly, these sub-populations may facilitate rapid identification and familiarisation of specific hierarchies.

Comparing the guidance to the findings highlights a tension between 'ideal' and pragmatic use of inheritance. This tension results in exclusions and caveats in guidance, for example – where convention or library design force the practitioner to use inheritance in a specific way or reasonable breeches of guidance (e.g. wide hierarchy smell vs protocol inheritance). Much of this everyday use of inheritance is excluded from studies and guidance as *improper* use.

Unfortunately, it is not always clear how to distinguish between exceptional or improper uses of inheritance (which are necessary, or at least cost effective) and badly formed 'proper' inheritance (which guidance aims to prevent).

The further mismatch between guidance and practice is in the population distribution. This study has found that most uses of inheritance are trivial – consisting of small, well defined hierarchies which fall well within the limits of guidance. Much of the guidance and feedback relating to inheritance discusses the size and complexity of hierarchies. There is little guidance for assessing quality in small hierarchies. This illustrates that the guidance is focused on avoiding rare large hierarchies, rather than providing advice for most inheritance use cases.

At the other end of the scale – all systems examined have at least one large hierarchy. Examination of the largest hierarchies indicate that many of these have internally consistent designs. The growth of large hierarchies is often moderated by higher level considerations such as modelling, or design patterns, indicating that these supersede implementation level guidance.

Much of the guidance relates to easily observable qualities of inheritance hierarchies, such as depth or width. However, most trivial hierarchies fall well within these limits, and the limits appear to be ignored by the well-designed larger hierarchies. This leaves the case where hierarchies grow past trivial size, but lack strong 'external' motivation or constraints – which may indeed be where guidance is required. Analysis has shown that many hierarchies, especially larger hierarchies, have an 'internal consistency' - both in terms of extension of an abstraction, and use of that abstraction by the rest of the system. This may be used to ensure that hierarchies that confound conventional size metrics can still be assessed. Inconsistencies in these larger structures may indicate different motivations for extensions. Ultimately these must be assessed subjectively, but it may be valuable to identify inconsistencies to assist in inspection, navigation, and familiarisation.

Returning to the ‘gold standard’ for inheritance use – the presence of an ‘is-a’ relationship. This is fundamentally a semantic distinction and is this currently beyond the reach of automated inspection. Each step in a hierarchy may represent one or more reasons to inherit. While it may be impossible to recover this motivating information – it would also be useful to have some guidance based on the information that *is* present i.e. hierarchy structure.

Structural regularities in inheritance hierarchies hint that some low-level good practices may be identifiable without full knowledge of the design considerations involved. With this in mind, inheritance hierarchies can be constructed to be as *regular* as possible to assist in detecting what design needs are being met by a particular inheritance hierarchy.

6 Conclusions

This thesis investigates what design guidance is perceived as useful in practice, and how high-level, generic guidance is balanced with low-level, project-specific concerns. Production software was also examined for evidence of these compromises. Specifically, how popular guidance relates to practitioner attitude, and the use of interfaces and inheritance in open source OO systems.

The contribution made by this thesis is that it helps close the gap between object-oriented design guidance and practice. It achieves this by i) discovering how practitioners approach design quality ii) discovering how interfaces are used in open-source Java systems, iii) discovering how inheritance is used in open-source systems, and iv) relating use back to practitioner sentiment and guidelines.

The literature indicates that there is serious interest in design quality and its measurement, but that the enduring overarching design principles are difficult to capture and measure in relation to specific programming languages or structures. In addition, there are varying opinions on ‘pure’ versus ‘pragmatic’ application of design principles in the literature, which further complicates an already subjective area.

RQ1.1: What does it mean for software to be of high quality, or well-designed? Can definition(s) of ‘good design’ be derived from the attitudes and beliefs of experienced practitioners, and the structure of mature software systems?

Good design is framed in terms of impact on development and maintenance. A large part of recognising quality is the judgement of practitioners (individuals and as a group). Principles are valued over rules. Time constraints heighten sensitivity to the rework cost of poor design decisions.

An investigation into the opinions of practitioners confirmed that popular design guidelines are well-known, and considered important by practitioners. However, it highlighted that software construction remains highly people-oriented as there is a marked reliance on personal and peer knowledge. Furthermore, the well-regarded guidelines are those which provide generally-applicable high-level object-oriented advice, while allowing for practitioner judgement. As guidance becomes more specific – such as for inheritance and design patterns – opinion becomes split on its usefulness. It is notable that much of the negative feedback for the more controversial guidance was about misapplication, especially by novices. This was seen to carry the risk of a high potential rework cost. This may explain the strong aversion to such prescriptive advice – as a key objective in software design is to postpone expensive or hard-to-undo decisions.

This indicates that general principles are more portable than specifics among practitioners – possibly because principles are used to support decision making in the absence of specific experience or complete specification. This is echoed by the fact that many of the general software design principles pre-date object-oriented design.

As noted in section 2.3.1, metrics often have some root in design principles, but can lose their connection to the original practitioner-facing concern during conversion to measurement. Without a strong link to principles, metrics may be a difficult to interpret into actionable feedback. This may go some way to explain the low uptake of software quality metrics. Contrast this with the popularity of code quality frameworks such as Sonar among respondents, which prompted positive comments from practitioners. These frameworks combine simple, metrics with many hand-crafted rules of thumb – this was perceived to be more like 'expert advice' by some respondents. These frameworks are also highly customisable and can be used to enforce consistent 'local style' or practice.

RQ1.2: How are programming language constructs used in practice? Specifically, mature and ubiquitous abstractions - such as interfaces and inheritance.

There is some evidence of 'textbook' use, and much use is simple. Outliers and 'misuse' are wide-spread indicating a pragmatic approach. High level decisions override general guidance – often in a specific dimension, indicating specifically overridden guidance, rather than general 'bad design'.

Chapter 4 investigates a well-known guideline – Program to an Interface – which was popularised by an influential design manual (Gamma et al., 1994). This guideline is at the level of design or implementation advice, but captures the intent behind higher level advice such as 'depend towards abstractions'. Support for interface-type abstraction is common in modern programming languages.

An analysis of 11 open source systems revealed that interface use is relatively low. Though there is a wide variety of use levels between systems. One issue that arose was the lack of a 'baseline' of interface use. It is not clear that all systems which vary in size and problem domain *should* have the same level of interface use. Indeed, survey feedback indicates that overuse of interfaces is perceived to create an unnecessary maintenance overhead.

Many user-defined types are not accessible via an interface at all, indicating that interfaces are not added to types as a matter of course. Analysis of interface 'impact' highlighted a major difference in implementation styles, with some systems having many 'low impact' interfaces which captured a low number of interactions. In one system (azureus), there appeared to be a 'rote' approach to interface creation, the system appears to compensate for a low use of inheritance with a high number of interfaces. This is in contrast to the remaining systems where interface definition appeared to be more ad hoc.

Systems that showed very high interface 'impact' had many dependencies captured by a low number of interfaces. This indicates that at least some systems can be arranged such that many of the type-type interactions can be mediated by interfaces without including interfaces that might be regarded as unnecessary. It is not clear if all systems can be economically arranged such that most of the type-type interactions can be mediated by meaningful interfaces (or other abstractions).

The analysis of interfaces highlighted common patterns of use at the individual interface level that are consistent with language or pattern conventions. While these have value in that they are easily recognisable and provide conventional architectures – they are common enough that they skew the entity population model for interfaces. By corollary, these patterns of use are internally consistent or defined by convention, so they would not be improved by moving them toward global 'norms' or averages. This highlights that comparison with a 'norm' is only useful if there is no overriding concern that can be used to assess or define an element. This suggests that interfaces could be assessed by identifying the (relevant) attributes of interest of their sub-population, deferring to global norms only when there is no better information. This provides the opportunity to acknowledge non-negotiable design constraints and preserve the important aspects of the specific structure, while avoiding un-actionable 'advice'.

For example, an interface may represent some external standard or protocol. This gives a way to ensure the cohesion of the interface by deferring to the external definition, at the cost of losing control over aspects such as the interface size and complexity. By contrast, a set of interfaces that represent the client sub-groups of a server object may be relatively volatile, evolving independently, but their main 'reason to exist' may be to provide module boundaries, so they could be assessed primarily in this capacity. These examples illustrate that there may be a different ranking of concerns in different circumstances, with some aspects being almost entirely subsumed by a higher-level design decision.

This is echoed in survey responses which warned against obscuring ‘essential complexity’ through over-design, or destroying locality simply to meet some generic thresholds. This is especially pertinent because, as noted above, the guideline thresholds do not appear to capture or explain the distribution of structures found in the examined systems.

Chapter 5 examined inheritance use in open source object-oriented systems. In line with previous work studying similar corpora, inheritance is used to define around three-quarters of types. However, much inheritance is trivial – indicating that ‘use of inheritance’ should not be equated with ‘problems arising from large hierarchies’. Despite this fact, many inheritance guidelines and metrics focus on large structures. Indeed, some hierarchies are so large they are as complex as small applications – it appears unreasonable to attempt to assess these extremes with the same instruments.

Considering the gross properties of inheritance hierarchies across the systems highlights different aspects of inheritance which may be used to assess the hierarchies in ‘inheritance-specific’ terms. Classical ‘strictly polymorphic’ use of inheritance was confirmed – with around half of the hierarchies accessed exclusively via their root type. However, a high degree of novelty in some hierarchy members was also detected, indicating that many hierarchies prioritise reuse over substitutability. Again, this indicates that these different hierarchies would require different assessment. For example – if a hierarchy leans more towards novel methods and reuse, assessment may focus more on consolidation of common code. Contrast this with ‘completely substitutable’ hierarchies which must be LSP compliant. This is evidence that Meyer’s pragmatic approach is more reflective of reality that Liskov’s restrictive view of inheritance use.

Hierarchy shape and the pattern of method invocation on hierarchy members also hints at different uses of inheritance. The invocation of methods on hierarchy members can provide clear indication of whether a hierarchy is used as a ‘pure’ LSP-compliant hierarchy, or if it exists more for reuse. Similarly, identifying hierarchy members or sub-hierarchies with markedly different invocation patterns may identify where refactoring opportunities. This view of inheritance hierarchies recognises that there may be different kinds of ‘self-consistent’ hierarchies, which perhaps should not be mixed.

For example – a line hierarchy where inheritance is used for incremental modification (versioning) of a type. This use of inheritance can be repeated without issue as the newest leaf is always the 'latest version' of the root type. However, if the hierarchy were to branch, this would indicate a new purpose for the hierarchy, indicating more reassessment may be required. In this case – it may be useful to highlight the 'self-sameness' of existing structures as an aspect of quality to highlight change as a departure from what has already been done. These may allow a practitioner to judge if a proposed modification to a hierarchy is 'more of the same' or changes the overall role of the hierarchy. Additionally this would leverage local (system) examples, which are more representative of local development style and tolerances.

As was discussed with interfaces above, there are sub-populations of inheritance hierarchies with different design constraints which in turn present different 'degrees of freedom'. It is important to understand which aspects of these structures are 'fixed' by other design decisions to avoid duplicate assessment and to avoid, as noted in the survey response, obscuring the essential properties of the solution by trying to make everything uniformly compliant with general guidance.

RQ1.3: How does practice impact the structure of software? Specifically – is there any evidence within systems that guidance is being followed? Can (or should) systems be fully compliant with all guidance?

Design reflects the pressures of practice – high-level decisions justify 'designed' structures and architecture, while volatile specifications or uncertainty leads to deferred design decisions – simpler structures, repetition, and unconsolidated design. Some sub-populations of structures can be identified which may represent common trade-offs, which might be assessed in their own terms.

Practitioners have indicated a desire to produce high quality systems – pragmatically described as easy to work with, or unsurprising to the examiner. This suggests that given enough time and resources, most parts of most systems might resemble the *Lego hypothesis* (Potanin et al., 2005). However, the survey respondents also indicated there is insufficient time and resources to achieve this.

With limited effort to spend on design quality, the key motivating qualities of each structure must take priority. Consequently, some components of a software system may indistinguishably represent: quickly implemented compromise, work in progress, or domain specific solution. This also suggests that what a practitioner considers to be a ‘good’ solution varies with available time and information. How then is quality to be recognised?

There is evidence of archetypical uses of interfaces and inheritance. These indicate that there are common design trade-offs or ‘modes of use’ that are acceptable to practitioners across systems, examples include – inheritance for versioning, inheritance for reuse, inheritance for substitutability, interface for decoupling monolithic class, empty interface for grouping other interfaces to facilitate runtime checks, not to mention design patterns, and common architectures which inform interface use and inheritance structures.

Firstly – the fact that these are trade-offs indicates that a design or elements of designs cannot be compliant with all guidance, all the time. This appears to be especially true where there is ‘essential complexity’ or some overriding high level decision. All guidelines might be considered, but some are set aside or deliberately deprioritised. Second - it is not clear how to *detect* the difference between deliberate setting aside of a guideline in favour of some other constraint, versus a place where a guideline could be applied, but has not.

These issues may explain why tools based on hand-crafted sets of rules have more positive response than metrics. They capture what is already known by practitioners. For example – it is possible to exclude specific elements from specific rules in Code Sonar where the outlier is a ‘justified’ violation to avoid repeated warnings.

Overall, an entity population model consisting of a normal distribution over the examined population aspect could not capture most of the aspects of design examined. However, sub-populations can be identified, which may allow definition of key features for these groups, and provide hints as to which guidelines are most relevant for assessment given limited time.

6.1 Research Goal Response

Goal: To close the gap between object oriented design guidance and practice.

Practitioner responses indicate that design guidelines can provide direction when there is no more specific guidance such as prior experience. Examination of systems indicate that most structures are simple, and that complexity is often related to modelling the problem domain, or the result of architectural choices. This indicates that the drive to comply with guidance should not override domain or architecture specific design decisions. If this is the case, this should inform the interpretation of general guidance based assessment (and metrics).

Collect information on the usefulness object oriented design guidance to practitioners.

A survey of practitioner attitude highlighted that experience is considered to be the most important factor affecting design quality. Many of the popular design guidelines were considered useful by respondents. Guidelines provide a 'fall back' when there is uncertainty or no specific prior experience. There was less support for more procedural guidance such as design patterns. There is some feeling that misapplied guidance can incur rework cost.

Analyse object-oriented systems for evidence of compliance with design guidance.

There is evidence of compliance with guidance but current guidance does not provide a clear picture of what compliance would look like at a system level – so compliance assessment must be done on individual interfaces or hierarchies. For interfaces, much use falls outside the normal usage (marker, constant) promoted by 'program to an interface', but these are present in all systems examined. Interfaces examined in context often show evidence of a high-level design motivation that justifies the addition of an interface, but is not motivated by a specific guideline. Similarly, examination of inheritance use revealed that most hierarchies are trivial – making size and complexity based guidance less relevant to these structures. The few large hierarchies usually have a domain or architectural motivation which overrides general guidance or limits.

Develop advice for guideline application or refine existing guidelines based on actual use and practitioner attitude (rather than arguments from theory).

Guideline compliance is variable. While guidance indicates what to do, it does not appear to capture when to do it. This is most obvious as there are many concrete dependencies in each system that are not mediated by interfaces. Additionally, observations indicate that it may not be the case that all uses of a give structure can be tied back to a guideline. If this is the case, guideline-based assessment may not be suitable in all cases.

6.2 Lessons Learnt

While instructive, construction of a tool for this study was time-consuming and raised additional validity threats that had to be addressed. Other researchers are strongly encouraged to consider all existing tools before embarking on creation of a new suite. In addition to the time and effort involved, this can impact reproducibility if the tools developed are not available or usable by other researchers. While there were well defined research questions, the volume of data produced and the exploratory nature of the study did entail longer periods of analysis than planned. This has yielded some of the more interesting findings in this report, and was mitigated somewhat by the iterative approach to tool development.

Scaling the size of the experiment to the available resources in terms of available hardware and person-hours is important if the work is to finish in a reasonable amount of time. Some of the systems selected for analysis have order-of-magnitude differences in size, which required careful consideration to present data in a representative format.

6.2.1 Observations and Recommendations

One of the goals of the empirical software engineering field is to discover 'Entity Population Models'. This would allow the measurement of a specific design elements such as inheritance hierarchies to be 'situated' within the population. However, the findings presented here raise question about some underlying assumptions of this view.

The proportions and usage of language features vary dramatically within systems – with many structures being simple or small. This indicates that the high-level goal of 'simplicity' holds much of the time. While the 'Lego hypothesis' that systems are constructed of many small regular components (Potanin et al., 2005) may not hold all of the time – it does describe most of the elements in the systems examined, most of the time.

On examination, many 'outliers' – are remarkably uniform and well defined. These often have some overriding justification specific to the application, indicating information is available that is more specific and authoritative than generic guidance. The question then becomes how to assess these structures other than using system level averages or thresholds.

Survey responses indicated that a large component of design assessment is in the 'eye of the beholder', varying with available time (what is economical), experience (what is easy to read), familiarity (domain models/processes). This information is not easily recovered or captured.

However, it was found that there are traces of context in source code – types of interfaces, naming conventions, recognisable design patterns, ties to external standards, invocation patterns and abstract structures in hierarchies. These provide hints where structures might be self-consistent, and where the local conventions and patterns have been 'broken' or might be restored. These might require hand-crafted rules for each category identified – however rule based tools are already seen as valuable and this approach would allow practitioners to include their 'approach' on these structures in the tooling.

Comprehending the design of software takes a substantial proportion of the time, and thus cost, of system maintenance and evolution. The findings presented here suggest that interface and inheritance use can be characterised by recognising context-specific patterns of use, which may provide some benefits to practitioners and design teaching.

It appears possible to identify some archetypical *kinds* of use which may assist in rapid comprehension of large structures by leverage existing knowledge of common patterns and structures. Decision-making might be assisted by presentation of the existing design trade-offs that can be detected in a code base. Finally, self-consistency within hierarchies and in use of a hierarchy by the rest of the system may indicate a novel notion of quality and how well an abstraction serves the rest of the system *without the need to necessarily understand the design motivation*.

6.3 Limitations

This work examines interface and inheritance use in OO systems with a view to determine how these are used and what aspects of use are assessable using the available information.

This work may be applicable in other OO languages. However, this would require more information about what kinds of structures are possible in these languages and what guidance is popular among the practitioners using each language. While it is likely that the high-level principles remain, specific guidance may differ.

As discussed in Chapters 4 and 5, the number of system examined here is smaller compared to large corpus studies. However, the populations examined were the interfaces and inheritance hierarchies within these systems, which provided many more units of comparison than the number of systems. Additionally, as stated previously – the goal of this study was to investigate these structures as they are used, rather than validate population norms over many systems. As such the interfaces and inheritance hierarchies examined here are likely to be representative of *some* similar structures in other systems. However, it is impossible to say what other categories of use may exist in the wider population.

Longitudinal analysis is common in the related work, indeed the corpus chosen for the inheritance study (Chapter 5) uses elements of the *evolution* set of its source corpus. Longitudinal analysis may have highlighted if the different user patterns identified in this study (both for interfaces and inheritance) have the same origins, or if these use patterns perhaps converge from different starting configurations. Additionally, it would be informative to determine if the new ways of examining these structures show stable patterns over time. This would highlight if the new methods of examining interface and inheritance use can be linked to system lifecycle properties such as aging or refactoring efforts.

6.4 Future Work

The findings presented above identify a variety of topics for further research.

The findings confirm that design guidelines associated with coupling, cohesion, size, complexity and programming to an interface are considered useful by practitioners. Following up on survey comments – practitioners identify a relationship between ease of testing, testing practices, and design quality – this relationship between quality and testability requires further investigation.

This work has identified that many concrete types are not accessible via an abstraction. To fully appreciate 'trade-off' a study should be carried out to consider all the places in design where interfaces (or other abstractions) are *not* used, and whether interface placement is appropriate, or at least consistent with placement in the rest of the system (i.e. can we identify the interface placement criteria in each system).

Systems evolve over time - assessment may be informed by the history of each interface – specifically to examine how these relate to 'kinds' of interface, or interface use. Additionally, evolution of the abstract backbones within hierarchies may give insight into developer practices – if these accumulate over time, this may indicate that they fall out of refactoring effort naturally. Indicators or checklists may be devised to assist in this process.

The use of inheritance for versioning is clear in isolated line hierarchies – but it is conceivable that this kind of inheritance is present, currently undetected, in other hierarchies. The balance between code *churn* and open-closed principle-style extension is not clear and would require an evolutionary study to examine in more detail.

Practitioner style appears to have a strong influence on the variability between systems. It may be useful to attempt to quantify style factors such as tolerance for concrete dependencies, complexity avoidance, or interface placement. This may identify exceptions to design rules, or set thresholds to withhold advice that is not of interest to a specific practitioner. This kind of model may also be used to highlight practitioner habits and set goals for improvement.

A key question raised is whether inheritance use can usefully be described by a limited set of structural patterns. Initial patterns suggested from this study are documented in *Appendix H - Structural Patterns for Inheritance Hierarchies*. Investigation of this question would involve automated identification of these patterns of use in a larger corpus. Validation with practitioners may involve incorporating the proposed patterns either as guidance or tools.

A further practitioner-facing problem is that of reassessing a design after a change. Changes may provide enough information to inform factors such as distribution of functions in abstract super-types, or reassessing complex execution paths. This is the sort of automated *heavy lifting* that might bring non-obvious or early problems to the designer's attention. In this vein – it may also be useful to investigate what factors practitioners consider when assessing inheritance (including how much of a hierarchy they examine) and how this information can be presented in a clear and timely manner at the time of decision making.

Lastly, there is ever-important possibility of replicating these studies with other practitioners and corpora, including refinements that have been discussed.

It is important that many of these suggestions involve mirroring or understanding practitioner behaviour, as most key principles and guidance are derived from early experience reports. Similarly, practitioner concerns underpin early work on metrics such as the 'viewpoints' which inspired the classic K&C suite.

7 References

Abdeen, H., Sahraoui, H., Shata, O., 2013a. How we design interfaces, and how to assess it. IEEE Int. Conf. Softw. Maintenance, ICSM 80–89. doi:10.1109/ICSM.2013.19

Abdeen, H., Shata, O., 2014. Type Variability and the Completeness of Interfaces in Java Applications. Int. J. Softw. Eng. Appl. 5, 1–7. doi:10.5121/ijsea.2014.5301

Abdeen, H., Shata, O., 2012. Metrics for assessing the design of software interfaces. Int. J. Advenced Res. Comput. Commun. Eng. 1, 1–8.

Abdeen, H., Shata, O., Erradi, A., 2013b. Software interfaces: On the impact of interface design anomalies, in: 2013 5th International Conference on Computer Science and Information Technology. IEEE, pp. 184–193. doi:10.1109/CSIT.2013.6588778

Agner, L.T.W., Soares, I.W., Stadzisz, P.C., Simão, J.M., 2013. A Brazilian survey on UML and model-driven practices for embedded software development. J. Syst. Softw. 86, 997–1005. doi:10.1016/j.jss.2012.11.023

Al Dallal, J., 2013. Object-oriented class maintainability prediction using internal quality attributes. Inf. Softw. Technol. 55, 2028–2048. doi:10.1016/j.infsof.2013.07.005

Al Dallal, J., Briand, L.C., 2010. An object-oriented high-level design-based class cohesion metric. Inf. Softw. Technol. 52, 1346–1361. doi:10.1016/j.infsof.2010.08.006

Al Dallal, J., Morasca, S., 2014. Predicting object-oriented class reuse-proneness using internal quality attributes, Empirical Software Engineering. doi:10.1007/s10664-012-9239-3

Amálio, N., Glodt, C., 2014. A tool for visual and formal modelling of software designs. Sci. Comput. Program. 98, 52–79. doi:10.1016/j.scico.2014.05.002

Bajeh, A.O., Basri, S., Jung, L.T., Almomani, M.A., 2014. Empirical validation of object-oriented inheritance hierarchy modifiability metrics, in: Proceedings of the 6th International Conference on Information Technology and Multimedia. IEEE, pp. 189–194. doi:10.1109/ICIMU.2014.7066628

Baker, A., van der Hoek, A., Ossher, H., Petre, M., 2012. Guest Editors' Introduction: Studying Professional Software Design. IEEE Softw. 29, 28–33. doi:10.1109/MS.2011.155

Baldwin, C., MacCormack, A., Rusnak, J., 2014. Hidden structure: Using network methods to map system architecture. Res. Policy 43, 1381–1397. doi:10.1016/j.respol.2014.05.004

Basili, V., Briand, L., Melo, W., 1996. A validation of object-oriented design metrics as quality indicators. Softw. Eng. IEEE … 22.

Basili, V.R., 1992. Software Modeling and Measurement: The Goal/Question/Metric Paradigm. University of Maryland Technical Report. UMIACS-TR-92-96 (1992).

Baxter, G., Frean, M., Noble, J., Rickerby, M., 2006. Understanding the shape of Java software. ACM SIGPLAN … 41, 397. doi:10.1145/1167515.1167507

Beck, K., 2014. Software Design: Why, When & How. JavaZone 2014 Conference.

Beck, L.L., Perkins, T.E., 1983. A Survey of Software Engineering Practice: Tools, Methods, and Results. IEEE Trans. Softw. Eng. SE-9, 541–561. doi:10.1109/TSE.1983.235114

Bernhardt, G., 2011. The Unix Chainsaw. YouTube.

Binkley, A.B., Schach, S.R., 1996. A comparison of sixteen quality metrics for object-oriented design. Inf. Process. Lett. 58, 271–275. doi:10.1016/0020-0190(96)00059-2

Bloch, J., 2008. Effective Java, 2nd ed. Addison-Wesley Professional.

Booch, G., 2011. Draw Me a Picture. IEEE Softw. 6–7.

Boxall, M. a. S., Araban, S., 2004. Interface metrics for reusability analysis of components. 2004 Aust. Softw. Eng. Conf. Proceedings. 40–51. doi:10.1109/ASWEC.2004.1290456

Brachman, 1983. What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks. Computer (Long. Beach. Calif). 16, 30–36. doi:10.1109/MC.1983.1654194

Brekelmans, B.L., 2014. What programmers do with inheritance in Java and C#. Universiteit van Amsterdam.

Briand, L.C., Melo, W.L., Wust, J., 2002. Assessing the applicability of fault-proneness models across object-oriented software projects. IEEE Trans. Softw. Eng. 28, 706–720. doi:10.1109/TSE.2002.1019484

Briand, L.C., Wüst, J., 2002. Empirical Studies of Quality Models in Object-Oriented Systems, in: Advances in Computers. pp. 97–166. doi:10.1016/S0065-2458(02)80005-5

Briand, L.C., Wüst, J., 2001. Integrating scenario-based and measurement-based software product assessment. J. Syst. Softw. 59, 3–22. doi:10.1016/S0164-1212(01)00045-0

Brooks, F.P., 2010. The Design of Design: Essays from a Computer Scientist, 1 edition. ed. Addison Wesley.

BSI, 2011. BSI Standards Publication Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. BSI Stand. Publ.

Buse, R.P.L., Zimmermann, T., 2012. Information needs for software development analytics. Proc. - Int. Conf. Softw. Eng. 987–996. doi:10.1109/ICSE.2012.6227122

Cai, Y., Wang, H., Wong, S., Wang, L., 2011. Architecture Recovery Based on Design Rule Hierarchy (No. DU-CS-11-03), DU-CS-11-03.

Cartwright, M., 1998. An empirical view of inheritance. Inf. Softw. Technol. 40, 795–799. doi:10.1016/S0950-5849(98)00105-0

Charette, R.N., 2005. Why software fails. IEEE Spectr. 42, 42–49. doi:10.1109/MSPEC.2005.1502528

Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object oriented design. IEEE Trans. Softw. Eng. 20, 476–493. doi:10.1109/32.295895

Chow, J., Tempero, E., 2011. Stability of Java interfaces: a preliminary investigation. Proc. 2nd Int. Work. … 38–44.

Coad, P., Yourdon, E., 1991. Object-oriented design. Yourdon Press.

Collberg, C., Myles, G., Stepp, M., 2007. An empirical study of Java bytecode programs.

Softw. - Pract. Exp. 37, 581–641. doi:10.1002/spe

Conley, C.A., Sproull, L., 2009. Easier Said than Done: An Empirical Investigation of Software Design and Quality in Open Source Software Development, in: 2009 42nd Hawaii International Conference on System Sciences. IEEE, pp. 1–10. doi:10.1109/HICSS.2009.174

Coplien, J., 2016. Symmetry in Design, Domain-Driven Design Europe. http://dddeurope.com, Brussels.

Daly, J., Brooks, A., Miller, J., Roper, M., Wood, M., 1996. Evaluating inheritance depth on the maintainability of object-oriented software. Empir. Softw. … 1, 109–132. doi:10.1007/BF00368701

Daly, J., Miller, J., Brooks, a., Roper, M., Wood, M., 1995. A survey of experiences amongst object-oriented practitioners. Proc. 1995 Asia Pacific Softw. Eng. Conf. 137–146. doi:10.1109/APSEC.1995.496962

Darcy, D.P., Kemerer, C.F., 2005. OO metrics in practice. IEEE Softw. 22. doi:10.1109/MS.2005.160

Devanbu, P., Zimmermann, T., Bird, C., 2016. Belief & evidence in empirical software engineering, in: Proceedings of the 38th International Conference on Software Engineering - ICSE ’16. ACM Press, New York, New York, USA, pp. 108–119. doi:10.1145/2884781.2884812

Dijkstra, E.W., 1968. Letters to the editor: go to statement considered harmful. Commun. ACM 11, 147–148. doi:10.1145/362929.362947

Dvorak, J., 1994. Conceptual entropy and its effect on class hierarchies. Computer (Long. Beach. Calif). 27, 59–63. doi:10.1109/2.294856

Dyer, R., Rajan, H., Nguyen, H.A., Nguyen, T.N., 2014. Mining billions of AST nodes to study actual and potential usage of Java language features, in: Proceedings of the 36th International Conference on Software Engineering - ICSE 2014. ACM Press, New York, New York, USA, pp. 779–790. doi:10.1145/2568225.2568295

Eichberg, M., Hermann, B., Mezini, M., Glanz, L., 2015. Hidden Truths in Dead Software Paths, in: ESEC/FSE 2015. p. TBA.

Eisenbach, S., Drossopoulou, S., 2002. Observing the Dynamic Linking Process in Java.

Emam, K. El, Goel, N., Rai, S., 2000. Thresholds for object-oriented measures. Softw. Reliab. Eng. … 24–37.

English, M., Buckley, J., Cahill, T., 2005. Applying Meyer’s taxonomy to object-oriented software systems, in: Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation. IEEE Comput. Soc, pp. 35–44. doi:10.1109/SCAM.2003.1238029

Ferreira, K.A.M., Bigonha, M.A.S., Bigonha, R.S., Mendes, L.F.O., Almeida, H.C., 2012. Identifying thresholds for object-oriented software metrics. J. Syst. Softw. 85, 244–257. doi:10.1016/j.jss.2011.05.044

Fontana, F.A., Dietrich, J., Walter, B., Yamashita, A., Zanoni, M., 2016. Antipattern and Code Smell False Positives: Preliminary Conceptualization and Classification, in: 2016 IEEE

23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). IEEE, pp. 609–613. doi:10.1109/SANER.2016.84

Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D., 1999. Refactoring: Improving the Design of Existing Code (Object Technology Series), 1 edition. ed. Addison Wesley.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1994. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

Garlan, D., Allen, R., Ockerbloom, J., 1995. Architectural mismatch or why it's hard to build systems out of existing parts, in: Proceedings of the 17th International Conference on Software Engineering - ICSE '95. ACM Press, New York, New York, USA, pp. 179–185. doi:10.1145/225014.225031

Garousi, V., Coşkunçay, A., Betin-Can, A., Demirörs, O., 2015. A survey of software engineering practices in Turkey. J. Syst. Softw. 108, 148–177. doi:10.1016/j.jss.2015.06.036

Garousi, V., Zhi, J., 2013. A survey of software testing practices in Canada. J. Syst. Softw. 86, 1354–1376. doi:10.1016/j.jss.2012.12.051

Gartner Inc., 2010. Gartner Estimates Global "IT Debt" to Be $500 Billion This Year, with Potential to Grow to $1 Trillion by 2015 [WWW Document]. http://www.gartner.com. URL http://www.gartner.com/newsroom/id/1439513 (accessed 8.28.17).

Gill, N.S., 2011. Inheritance Hierarchy Based Reuse & Reusability Metrics in OOSD. IJCSE 3, 2300–2309.

Girba, T., Lanza, M., Ducasse, S., 2005. Characterizing the Evolution of Class Hierarchies, in: Ninth European Conference on Software Maintenance and Reengineering. IEEE, pp. 2–11. doi:10.1109/CSMR.2005.15

Gorschek, T., Tempero, E., Angelis, L., 2010. A large-scale empirical study of practitioners' use of object-oriented concepts, in: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10. ACM Press, New York, New York, USA, p. 115. doi:10.1145/1806799.1806820

Gößner, J., Mayer, P., Steimann, F., 2004. Interface utilization in the JAVA Development Kit. … 2004 ACM Symp. … 1310–1315.

Gurp, J. Van, Bosch, J., 2001. Design, implementation and evolution of object oriented frameworks: concepts and guidelines. Softw. Pract. Exp. 277–300.

Halbert, D.C., O'Brien, P.D., 1987. Using Types and Inheritance in Object-Oriented Programming. IEEE Softw. 4, 71–79. doi:10.1109/MS.1987.231776

Harrison, R., Counsell, S., Nithi, R., 2000. Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. J. Syst. Softw. 52, 173–179. doi:10.1016/S0164-1212(99)00144-2

Hitz, M., Montazeri, B., 1996. Chidamber and kemerer's metrics suite: A measurement theory perspective. IEEE Trans. Softw. Eng. 22, 267–271. doi:10.1109/32.491650

Hitz, M., Montazeri, B., 1995. Measuring Coupling and Cohesion In Object-Oriented Systems. Proc. Int. Symp. Appl. Corp. Comput. 25–27. doi:10.1.1.409.4862

Hoffman, D., 1990. On criteria for module interfaces. Softw. Eng. IEEE Trans. 16.

Holub, A., 2003. Why extends is evil [WWW Document]. Java World - Java Toolbox. URL http://www.javaworld.com/article/2073649/core-java/why-extends-is-evil.html (accessed 1.22.15).

Hunt, A., Thomas, D., 1999. The Pragmatic Programmer: From Journeyman to Master.

IEEE, 2014. IEEE Standards Definition Database [WWW Document]. IEEE Stand. Assoc. URL http://dictionary.ieee.org/

ISO/IEC, 2010. ISO/IEC/IEEE 24765:2010 - Systems and Software Engineering - Vocabulary, 2010th ed. IEEE.

Jabangwe, R., Börstler, J., Šmite, D., Wohlin, C., 2014. Empirical evidence on the link between object-oriented measures and external quality attributes: a systematic literature review. Empir. Softw. Eng. 1–54. doi:10.1007/s10664-013-9291-7

Joel Spolsky, 2005. Making Wrong Code Look Wrong [WWW Document]. Joel Softw. URL http://www.joelonsoftware.com/articles/Wrong.html (accessed 1.22.15).

Johnson, R.E., Foote, B., 1988. Designing Reusable Classes. J. Object-Oriented Program. 1, 22–35.

Kay, D.A., Ram, S., 2003. Dr. Alan Kay on the Meaning of “Object-Oriented Programming” [WWW Document]. URL http://www.purl.org/stefan_ram/pub/doc_kay_oop_en

Kegel, H., Steimann, F., 2008. Systematically refactoring inheritance to delegation in java, in: Proceedings of the 13th International Conference on Software Engineering - ICSE ’08. ACM Press, New York, New York, USA, p. 431. doi:10.1145/1368088.1368147

Keller, M., 2016. Bug 486067 [WWW Document]. Bugzilla – Full Text Bug List. URL https://bugs.eclipse.org/bugs/show_bug.cgi?format=multiple&id=486067 (accessed 9.20.16).

Kelly, T., Buckley, J., 2009. Cognitive levels and Software Maintenance Sub-tasks, in: PPIG 2009 - 21st Annual Workshop. Psychology of Programming Interest Group, p. . doi:10.1.1.222.7014

Kitchenham, B., 2010. What’s up with software metrics? – A preliminary mapping study. J. Syst. Softw. 83, 37–51. doi:10.1016/j.jss.2009.06.041

Kitchenham, B., Pfleeger, S., 1996. Software quality: the elusive target [special issues section]. IEEE Softw. 13, 12–21. doi:10.1109/52.476281

Kitchenham, B., Pfleeger, S.L., Fenton, N., 1995. Towards a framework for software measurement validation. IEEE Trans. Softw. Eng. 21, 929–944. doi:10.1109/32.489070

Lanza, M., 2003. Object-Oriented Reverse Engineering. Institut fur Informatik und angewandte Mathematik.

Lanza, M., Marinescu, R., Practice, O.M., 2006. Object-Oriented Metrics in Practice, Design. Springer-Verlag Berlin Heidelberg. doi:10.1007/3-540-39538-5

Larman, C., 2004. Applying UML and Patterns – An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd ed.), 3rd ed. Prentice Hall.

Larman, C., 2001. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process. Prentice Hall PTR.

Lehman, M.M., Ramil, J.F., 2003. Software evolution—Background, theory, practice. Inf. Process. Lett. 88, 33–44. doi:10.1016/S0020-0190(03)00382-X

Liskov, B., 1988. Keynote address - data abstraction and hierarchy. ACM SIGPLAN Not. 23, 17–34. doi:10.1145/62139.62141

Martin, R., 2000. Design principles and design patterns. Object Mentor.

Martin, R., 1996. The Open-Closed Principle. C++ Rep. 1–14.

Martin, R.C., 2016. The Future of Programming [WWW Document]. X/UP. URL https://youtu.be/ecIWPzGEbFc (accessed 7.27.16).

Martin, R.C., 2009. Clean Code: A Handbook of Agile Software Craftsmanship, 1 edition. ed. Prentice Hall.

Martin, R.C., 2005a. PrinciplesOfOod [WWW Document]. butunclebob.com. URL http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod (accessed 5.18.15).

Martin, R.C., 2005b. SRP: Single Responsibility Principle [WWW Document]. URL https://docs.google.com/open?id=0ByOwmqah_nuGNHEtcU5OekdDMkk (accessed 5.12.15).

Martin, R.C., 2002. Agile Software Development, Principles, Patterns, and Practices, (15 Oct. 2. ed. Pearson.

Martin, R.C., 1996. The Dependency Inversion Principle. C++ Rep. 8, 61–66.

Mayer, T., Hall, T., 1999a. A Critical Analysis of Current OO Design Metrics. Softw. Qual. J. 8, 97–110. doi:10.1023/A:1008900825849

Mayer, T., Hall, T., 1999b. Measuring OO systems: a critical analysis of the MOOD metrics, in: Proceedings Technology of Object-Oriented Languages and Systems. TOOLS 29 (Cat. No.PR00275). IEEE Comput. Soc, pp. 108–117. doi:10.1109/TOOLS.1999.779004

Meyer, B., 1996. The many faces of inheritance: a taxonomy of taxonomy. Computer (Long. Beach. Calif). 29, 105–108. doi:10.1109/2.494093

Murphy, B., 2011. The difficulties of building generic reliability models for software. Empir. Softw. Eng. 17, 18–22. doi:10.1007/s10664-011-9184-6

Niculescu, M., Dugerdil, P., Canedo, B.M., 2015. Measuring Inheritance Patterns in Object Oriented Systems, in: Proceedings of the 8th India Software Engineering Conference on XXX - ISEC '15. ACM Press, New York, New York, USA, pp. 130–138. doi:10.1145/2723742.2723755

Ó Cinnéide, M., Hemati Moghadam, I., Harman, M., Counsell, S., Tratt, L., 2016. An experimental search-based approach to cohesion metric evaluation. Empir. Softw. Eng. 1–38. doi:10.1007/s10664-016-9427-7

Oracle, 2014a. What Is an Interface? [WWW Document]. The Java^{TM} Tutorials. URL https://docs.oracle.com/javase/tutorial/java/concepts/interface.html

Oracle, 2014b. Interface Serializable [WWW Document]. URL http://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html

Page-Jones, M., 1992. Comparing techniques by means of encapsulation and connascence. Commun. ACM 35, 147–151. doi:10.1145/130994.131004

Palmer, S., Coad, P., 2004. The Coad Letter: Modeling and Design Edition [WWW Document]. Borl. Dev. Netw. URL https://web.archive.org/web/20041125095103/http://thecoadletter.com/article/0,1410,29697,00.html (accessed 4.6.17).

Palomba, F., Zanoni, M., Fontana, F.A., Lucia, A. De, Oliveto, R., 2016. Smells Like Teen Spirit: Improving Bug Prediction Performance Using the Intensity of Code Smells. 2016 IEEE Int. Conf. Softw. Maint. Evol., Lecture Notes in Electrical Engineering 339, 244–255. doi:10.1109/ICSME.2016.27

Parnas, D., 1972. On the criteria to be used in decomposing systems into modules. Commun. ACM.

Perepletchikov, M., Ryan, C., Frampton, K., 2007. Cohesion Metrics for Predicting Maintainability of Service-Oriented Software, in: Seventh International Conference on Quality Software (QSIC 2007). IEEE, pp. 328–335. doi:10.1109/QSIC.2007.4385516

Perepletchikov, M., Ryan, C., Tari, Z., 2010. The Impact of Service Cohesion on the Analyzability of Service-Oriented Software. IEEE Trans. Serv. Comput. 3, 89–103. doi:10.1109/TSC.2010.23

Pfleeger, S., Kitchenham, B., 2001. Principles of survey research: part 1: turning lemons into lemonade. ACM SIGSOFT Softw. Eng. … 26, 16–18.

Potanin, A., Noble, J., Frean, M., Biddle, R., 2005. Scale-free geometry in OO programs. Commun. ACM 48, 99–103. doi:10.1145/1060710.1060716

Radjenović, D., Heričko, M., Torkar, R., Živkovič, A., 2013. Software fault prediction metrics: A systematic literature review. Inf. Softw. Technol. 55, 1397–1418. doi:10.1016/j.infsof.2013.02.009

Raymond, E.S., 2003. The Art of Unix Programming, Revision 1. ed. Addison-Wesley.

Reenskaug, T., Coplien, J.O., 2009. The DCI Architecture : A New Vision of Object ‐ Oriented Programming [WWW Document]. Artima Dev. URL http://www.artima.com/articles/dci_vision.html

Riaz, M., Mendes, E., Tempero, E., 2009. A systematic review of software maintainability prediction and metrics, in: 2009 3rd International Symposium on Empirical Software Engineering and Measurement. IEEE, pp. 367–377. doi:10.1109/ESEM.2009.5314233

Riehle, D., 1996. Describing and composing patterns using role diagrams. WOON 137–152.

Riel, A.J., 1996. Object-Oriented Design Heuristics, 1 edition. ed. Addison Wesley.

Robbins, N.B., Heiberger, R.M., Court, C., Hall, S., 2011. Plotting Likert and Other Rating Scales. Jt. Stat. Meet. 1058–1066.

Rocha, H., Valente, M.T., 2011. How Annotations are Used in Java: An Empirical Study. SEKE 426–431.

Romano, D., Pinzger, M., 2011. Using source code metrics to predict change-prone java interfaces. Softw. Maint. (ICSM), 2011 … 303–312.

Romano, D., Raemaekers, S., Pinzger, M., 2014. Refactoring Fat Interfaces Using a Genetic Algorithm, in: 2014 IEEE International Conference on Software Maintenance and Evolution. IEEE, pp. 351–360. doi:10.1109/ICSME.2014.57

RosettaCode, 2015. Abstract type [WWW Document]. Rosetta Code. URL http://rosettacode.org/wiki/Abstract_type#Scala

Sabané, A., Guéhéneuc, Y.-G., Arnaoudova, V., Antoniol, G., 2016. Fragile base-class problem, problem? Empir. Softw. Eng. 1–46. doi:10.1007/s10664-016-9448-2

Schocken, S., 2012. Taming complexity in large-scale system projects, in: Proceedings of the 43rd ACM Technical Symposium on Computer Science Education - SIGCSE ’12. ACM Press, New York, New York, USA, p. 409. doi:10.1145/2157136.2157259

Simons, C., Singer, J., White, D.R., 2015. Search-Based Refactoring: Metrics Are Not Enough, in: Barros, M., Labiche, Y. (Eds.), Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Lecture Notes in Computer Science. Springer International Publishing, Cham, pp. 47–61. doi:10.1007/978-3-319-22183-0_4

Sjøberg, D.I.K., Anda, B., Mockus, A., 2012. Questioning software maintenance metrics, in: Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM ’12. ACM Press, New York, New York, USA, p. 107. doi:10.1145/2372251.2372269

SonarQube, 2013. Rules | Inheritance tree of classes should not be too deep [WWW Document]. Rules. URL https://sonarqube.com/coding_rules#q=inheritance%7Clanguages=java (accessed 12.1.16).

Steimann, F., 2007. The Infer Type refactoring and its use for interface-based programming. J. Object Technol. 6, 99–120. doi:10.5381/jot.2007.6.2.a5

Steimann, F., Mayer, P., 2005. Patterns of Interface-Based Programming. J. Object Technol. 4, 75–94.

Steimann, F., Siberski, W., Kühne, T., 2003. Towards the systematic use of interfaces in JAVA programming. … Pract. Program. Java 13–17.

Stevenson, J., 2014. Research proposal: Objective evaluation of object oriented design quality, in: ACM International Conference Proceeding Series. doi:10.1145/2601248.2613080

Stevenson, J., Wood, M., 2018. Inheritance Usage Patterns in Open-Source Systems, in: ICSE ’18 Proceedings of the 40th International Conference on Software Engineering. ACM Digital Library, pp. 245–255. doi:10.1145/3180155.3180168

Stevenson, J., Wood, M., 2017. Recognising object-oriented software design quality: a practitioner-based questionnaire survey. Softw. Qual. J. doi:10.1007/s11219-017-9364-8

Strachey, C., 2000. Fundamental Concepts in Programming Languages. Higher-Order Symb.

Comput. 13, 11–49. doi:10.1023/A:1010000313106

Suryanarayana, G., Samarthyam, G., Sharma, T., 2015. Hierarchy Smells, in: Refactoring for Software Design Smells. Elsevier, pp. 123–192. doi:10.1016/B978-0-12-801397-7.00006-0

Systa, T., Muller, H., 2000. Analyzing Java software by combining metrics and program visualization, in: Proceedings of the Fourth European Conference on Software Maintenance and Reengineering. IEEE Comput. Soc, pp. 199–208. doi:10.1109/CSMR.2000.827328

Taenze, D.H., Ganti, M., Podar, S., 1989. Problems in Object-Oriented Software Reuse, in: Cook, S. (Ed.), Proceedings of the 1989 European Conference on Object-Oriented Programming. Cambridge University Press, Nottingham, pp. 25–38.

Taivalsaari, A., 1996. On the notion of inheritance. ACM Comput. Surv. 28, 438–479. doi:10.1145/243439.243441

Taube-schock, C., Walker, R.J., Witten, I.H., 2011. Can We Avoid High Coupling? ECOOP 2011 – Object-Oriented Program., Lecture Notes in Computer Science 6813, 204–228. doi:10.1007/978-3-642-22655-7

Tempero, E., Counsell, S., Noble, J., 2010a. An empirical study of overriding in open source java, in: Mans, B. and Reynolds, M. (Ed.), Conferences in Research and Practice in Information Technology Series. Australian Computer Society, Inc. Darlinghurst, Australia, Australia ©2010, Brisbane, Australia, pp. 3–12.

Tempero, E., Craig, A., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., Noble, J., 2010b. Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies, in: 2010 Asia Pacific Software Engineering Conference (APSEC2010). pp. pp336-345. doi:dx.doi.org/10.1109/APSEC.2010.46

Tempero, E., Noble, J., Melton, H., 2008. How do Java programs use inheritance? An empirical study of inheritance in Java software. … 2008–Object-Oriented Program. 667–691.

Tempero, E., Yang, H.Y., Noble, J., 2013. What Programmers Do with Inheritance in Java, in: Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). pp. 577–601. doi:10.1007/978-3-642-39038-8_24

Torchiano, M., Tomassetti, F., Ricca, F., Tiso, A., Reggio, G., 2013. Relevance, benefits, and problems of software modelling and model driven techniques - A survey in the Italian industry. J. Syst. Softw. 86, 2110–2126. doi:10.1016/j.jss.2013.03.084

Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., Poshyvanyk, D., 2015. When and Why Your Code Starts to Smell Bad, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. IEEE, pp. 403–414. doi:10.1109/ICSE.2015.59

Veerappa, V., Harrison, R., 2013. An Empirical Validation of Coupling Metrics Using Automated Refactoring, in: 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement. IEEE, pp. 271–274. doi:10.1109/ESEM.2013.37

Venners, B., 2002. Josh Bloch on Design [WWW Document]. artima Dev. URL http://www.artima.com/intv/bloch13.html

Venners, B., 2001. A Conversation with Java’s Creator, James Gosling [WWW Document]. JavaWorld. URL http://www.artima.com/intv/gosling36.html (accessed 6.26.16).

Verner, J., Sampson, J., Cerpa, N., 2008. What factors lead to software project failure? 2008 Second Int. Conf. Res. Challenges Inf. Sci. 71–80. doi:10.1109/RCIS.2008.4632095

Voigt, J., Irwin, W., Churcher, N., 2010. Class Encapsulation and Object Encapsulation: An Empirical Study. ENASE2010 5th Int. Conf. Eval. Nov. Approaches to Softw. Eng. 22-24 Jul 2010 171–178.

Weck, W., Szyperski, C., 1996. Do We Need Inheritance?, in: Workshop on Composability Issues in Object-Orientation at ECOOP ’96.

Weyuker, E.J., 2011. Empirical Software Engineering Research - The Good, The Bad, The Ugly, in: 2011 International Symposium on Empirical Software Engineering and Measurement. IEEE, pp. 1–9. doi:10.1109/ESEM.2011.66

Yamashita, A., Moonen, L., 2013. Do developers care about code smells? An exploratory survey, in: 2013 20th Working Conference on Reverse Engineering (WCRE). IEEE, pp. 242–251. doi:10.1109/WCRE.2013.6671299

Yau, S.S., Collofello, J.S., MacGregor, T., 1978. Ripple effect analysis of software maintenance. IEEE Comput. Soc. Second Int. Comput. Softw. Appl. Conf. 1978 COMPSAC 78 60–65. doi:10.1109/CMPSAC.1978.810308

Yourdon, E., Constantine, L.L., 1975. Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. Yourdon Press.

Yue, K.-B., Yang, T., Ding, W., Chen, P., 2004. Open courseware and computer science education. J. Comput. Sci. Coll. 20, 178–186.

Zhang, C., Budgen, D., 2013. A survey of experienced user perceptions about software design patterns. Inf. Softw. Technol. 55, 822–835. doi:10.1016/j.infsof.2012.11.003

Zhang, C., Budgen, D., 2012. What Do We Know about the Effectiveness of Software Design Patterns? IEEE Trans. Softw. Eng. 38, 1213–1231. doi:10.1109/TSE.2011.79

8 Appendices

8.1 Appendix A – Terminology

8.1.1 Literature Review Terminology

Clean Code – A popular set of implementation guidance and methods based on industry experience (Martin 2009), advocating 'software craftsmanship'. Includes: clarity of design including naming, small size, simplicity, modularity, documentation, test driven development.

Entity Population Model – Allows us to define what *normal values* are for a measurement, allowing a measurement to be interpreted. (Kitchenham et al., 1995)

Native Type – the set of all method signatures defined by an objects operations. (Gamma et al., 1994)

Type – a particular interface, defined by a name and a set of (possibly abstract) method signatures. (Gamma et al., 1994)

Ripple effect –a change to one part of a system causes new change(s) to be required elsewhere in the system (Yau et al., 1978)

8.1.2 Inheritance Chapter Terminology

The java language definition recognises two main categories of entity which may be referred to:

- **Primitive types** – byte, short, int, long, char, float, double, boolean
- **Reference types** – pointers to objects – class, abstract class, interface, array, enum

In this chapter, 'primitives' will be used to refer to the first type of entity.

When discussing object references:

- 'class' will be referred to as '**concrete class**' to avoid ambiguity
- '**type**' will be used in place of 'concrete class, abstract class, or interface' where the specific reference type is not important.
- The terms 'concrete class', 'abstract class', and 'interface' will be used separately and specifically, where disambiguation is required.

Terms that have already been defined that are mentioned in this chapter:

DIT_{MAX} – Depth of Inheritance Tree – DIT is the longest path from the root to a leaf in an inheritance hierarchy, where the root of the hierarchy is at DIT zero. DIT was first proposed in the C&K metrics suite (Chidamber and Kemerer, 1994). DIT_{MAX} is the largest DIT value across all members in an inheritance hierarchy. In this study, interfaces are not included in DIT calculations.

BIT_{MAX} – Breadth of Inheritance Tree – derived from BIT (Harrison et al., 2000), this value indicates the number of hierarchy members at the widest point in an inheritance hierarchy.

Key findings are summarised at the end of each section in the form **[*KF chapter.finding]*** for ease of reference. e.g. *[KF 3.1] is key finding one, in chapter three.*

8.2 Appendix B - Object-Oriented Concepts

Object-oriented design, and software design in general, has a rich vocabulary of concepts. These are useful when discussing different properties of designs and source code. These concepts are defined and discussed for reference in this appendix

8.2.1 Classes

A class is a description of a category of things. In object-oriented (OO) software, a class is a *blueprint* which can be used to construct *objects*. An object constructed from a class at runtime is known as an *instance* of that class. Different instances of a class lead separate existences and may change independently from each other. For example, a program may contain objects which represent files – these objects may all be instances of a *File* class. This means that the file objects have the same kinds of properties, such as a stored file path, but have different individual values for those paths.

Classes facilitate encapsulation, which is the "… *process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation*" Booch et al. in (Voigt et al., 2010).

8.2.2 Types

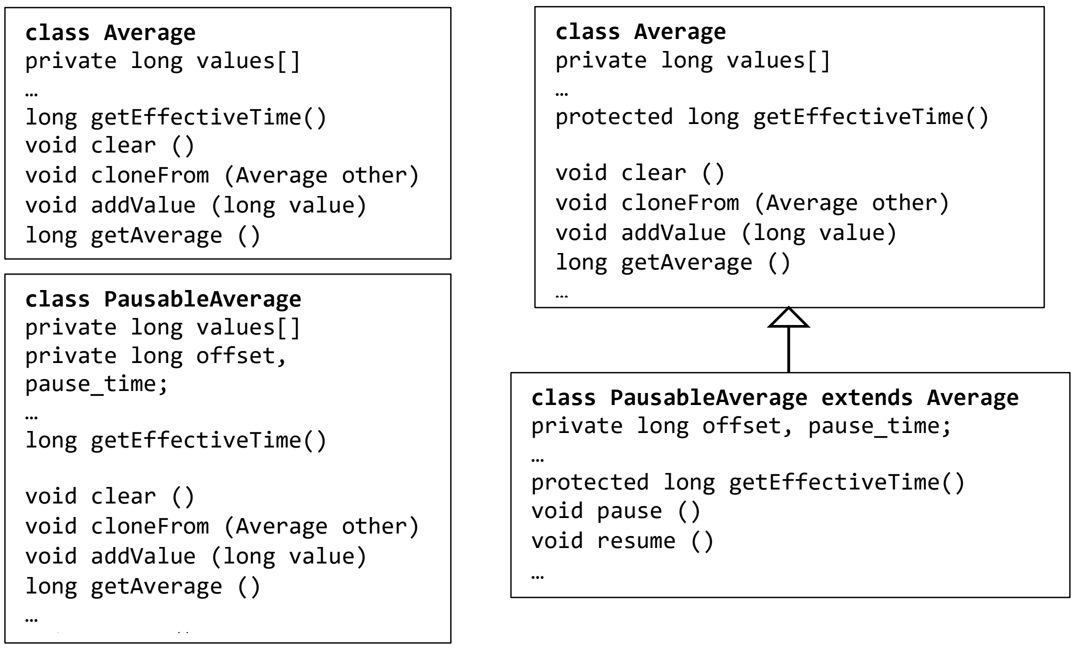
The term *type* has different technical meanings in areas related to computer science. A definition relating to object-oriented design is provided in the Design Patterns manual - *"A type is a name used to denote a particular interface. We speak of an object as having the type "Window" if it accepts all requests for the operations defined in the interface named "Window." An object may have many types,…"* (Gamma et al., 1994). Where a native interface is "*The set of all signatures defined by an object's operations…*"(Gamma et al., 1994). Note that while instances of the same class (discussed above) share a common type, objects may have types other than the type of the class of which they are an instance. Much of the effort in object-oriented design is to define types and their interactions.

8.2.3 Inheritance

This section describes a key design feature of object-oriented programming languages – inheritance, which is "*a semantic notion by which the responsibilities (properties and constraints) of a subclass are considered to include the responsibilities of a superclass, in addition to its own, specifically declared responsibilities.*" (ISO/IEC, 2010). The section focuses on class- or concrete-inheritance where both implementation and interface are inherited. Abstract inheritance is covered in the next section.

Inheritance allows a practitioner to define a new feature in terms of code that already exists - potentially reducing the amount of new code that must be introduced. This is more precisely defined by Taivalsaari - "*subclassing – an implementation mechanism for sharing code and representation*" (Taivalsaari, 1996). Figure 54 shows an example of inheritance, note that language-specific details have been omitted for clarity. On the left, two similar classes are defined separately, which requires the duplicate definition of multiple identical behaviours. While on the right, the class *PausableAverage* has been defined using class inheritance, by extending the class *Average*.

Definition by extension means that the child class, (*PausableAverage*) immediately gains access to all the behaviour of the parent class (*Average*). It is often possible to find existing classes which have behaviour similar to that which is required. This allows rapid development by adapting what already exists and avoids code duplication.



```
class Average
private long values[]
…
long getEffectiveTime()
void clear ()
void cloneFrom (Average other)
void addValue (long value)
long getAverage ()
```

```
class PausableAverage
private long values[]
private long offset,
pause_time;
…
long getEffectiveTime()

void clear ()
void cloneFrom (Average other)
void addValue (long value)
long getAverage ()
…
```

```
class Average
private long values[]
…
protected long getEffectiveTime()

void clear ()
void cloneFrom (Average other)
void addValue (long value)
long getAverage ()
…
```

```
class PausableAverage extends Average
private long offset, pause_time;
…
protected long getEffectiveTime()
void pause ()
void resume ()
…
```

Figure 54 - Simple inheritance (reuse), in org.gudy.azureus2.core3.util.Average and org.gudy.azureus2.core3.util.PausableAverage, note that method definitions have been omitted for clarity

Note that in this thesis, *subtype* and *super-type* are used in preference to *child* and *parent*, these terms are normally interchangeable. In Figure 54 the *Average* type provides a protected *getEffectiveTime()* method which is used for internal calculations, this method is redefined in *PausableAverage* to account for paused time. This allows methods inherited from the parent class to behave consistently in the new class. Note that this re-definition of an inherited method is called *overriding*.

Consequently – the amount of additional testing, review, and maintenance required for the new behaviour may be reduced if the reused components have already been validated.

8.2.4 Interfaces

In the design literature, an interface is the "*…set of all signatures defined by an object's operations is called the interface to the object*"(Gamma et al., 1994). Note that 'object' indicates that this is a run-time description. Interfaces define types without specifying implementation details. This allows method or function signatures to be defined as a contract of behaviour with no practical details of *how* the specified behaviour is to be implemented. This is of practical benefit, as it allows the expected or required behaviour of a type to be specified at a high level before implementation begins. Additionally, different implementations of an interface may be defined at a later time.

The IEEE Standards Definition Database (IEEE, 2014) contains 27 definitions of the word *interface*, 16 of which could be applied to interfaces as they exist in software without any difficulty or stretch of the definitions. Table 35 shows some of the more relevant definitions for the topics presented here.

While some languages offer a construct that consists entirely of signatures (abstract interfaces), it is possible to discuss the interface of a concrete type as all the public method signatures defined in that type. The concept of an abstract type or protocol appears in many modern programming languages as can been seen in Table 37 (see section 8.3 Appendix C), 44% of the entries in a survey of modern languages (RosettaCode, 2015) have native support for interfaces, with a further 37% having the capacity to simulate interfaces in some way - giving a total of 81%.

8.2.4.1 Interfaces in Java

Interfaces are defined in the Java programming language as structures which "*…allow the description (definition) of a type with the understanding that the type is entirely abstract and represents a behavioural contract.*" (Oracle, 2014a). In terms of use – interfaces are described as *"a contract between the class and the outside world, and this contract is enforced at build time by the compiler"* (Oracle, 2014a). These definitions describe the constraints interfaces place on implementers and the guarantees they provide for clients, but offer no guidance as to when and how interfaces should be used.

Table 35 - Selected definitions of the term "interface" from (IEEE, 2014), emphasis added to match source.

A shared boundary between two functional entities. *A standard specifies the services in terms of the functional characteristics and behavior observed at the interface. The standard is a contract in the sense that it documents a mutual obligation between the service user and provider and assures a stable definition of that obligation.* IEEE Std 1003.0-1995 IEEE Guide to the POSIX Open System Environment (OSE)
A shared boundary that specifies the interconnection between two units or systems, hardware or software. In hardware, the specification includes the type, quantity, and function of the interconnection circuits and the type and form of signals to be interchanged via those circuits. In software, the specification includes the object type and, where necessary, the name or instance handle of specific objects copied or shared between the two systems. IEEE Std 1226-1998 IEEE Trial-Use Standard for A Broad-Based Environment for Test (ABBET) Overview and Architecture IEEE Std 1671™-2010 IEEE Standard for Automatic Test Markup Language (ATML) for Exchanging Automatic Test Equipment and Test Information via XML
The declaration of the meaning and the signature for a property or constraint. The interface states "what" a property (responsibility) knows or does or what a constraint (responsibility) must adhere to. The interface specification consists of the meaning (semantics) and the signature (syntax) of a property or constraint. IEEE Std 1320.2-1998 IEEE Standard for Conceptual Modeling Language Syntax and Semantics for IDEF1X97 (IDEFobject)

Java interfaces are, by default, visible only in the package where they are declared. This allows the segregation of local (package) interfaces and global (public) interfaces that are visible to any class in any package. Interfaces can take on most of the same roles as any other reference type – variable declarations, method parameters, method return types, use in parameterised types, etc. The main difference being that no objects can be created (instantiated) from interfaces. To obtain an interface-typed object, an object must be created from a class that implements that interface.

A Java interface can contain zero or more (abstract) method signatures – meaning that the methods have only a method signature and no definition. Interfaces in Java may also contain static variables, inner class definitions including enumerated types, and more recently default implementations (of methods).

Interfaces take part in three main types of interaction in Java programs:

1. Implementation by a more concrete class – i.e. an abstract class or concrete class
2. Use as a type – e.g. as a type in a variable declaration statement
3. Interface Inheritance - extension by another interface

Of these relationships, the second is perhaps most relevant to design work – as it allows clients of an interface to depend on an abstraction, rather than a specific implementation. Figure 55 illustrates a possible interface relationship – the objects on the right implement the *ProgressObserver* interface, meaning that these objects can be used anywhere the *ProgressObserver* type is declared.

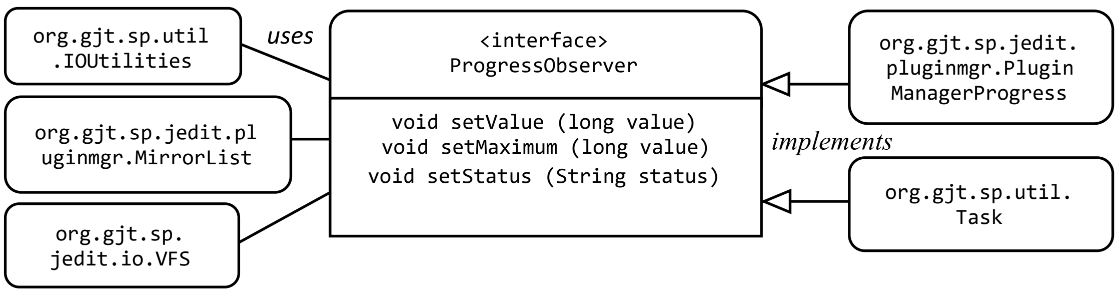

org.gjt.sp.util .IOUtilities
uses
org.gjt.sp.jedit.pluginmgr.MirrorList
org.gjt.sp. jedit.io.VFS
<interface> ProgressObserver
void setValue (long value)
void setMaximum (long value)
void setStatus (String status)
implements
org.gjt.sp.jedit. pluginmgr.Plugin ManagerProgress
org.gjt.sp.util. Task

Figure 55 - Example relationship between client, interface and implementer (Types from *jedit (5.1.0)*)

Each of the client objects may refer to objects of type *ProgressObserver* and may also refer any of the methods specified in the *ProgressObserver* interface at design time, before implementations of *ProgressObserver* type have been defined. As Liskov notes – "… *a programmer cares only about what it* [a method] *does and not how it is implemented.*" (Liskov, 1988)

Conversely, any class may implement the *ProgressObserver* interface, so long as that implementing class provides the methods described in the interface. This involves matching the method signatures exactly (method name, return type, parameters). At compile time, errors will result if a client attempts to call a method not defined by a type (or interface). Similarly, a concrete class implementing an interface will not compile if it does not provide adequate implementations for methods it has promised to implement.

At runtime, methods called on interface types are resolved via dynamic-binding (sometimes called late-binding) of the method call to whatever object is implementing the interface. This allows the specific implementer of an interface to be chosen or changed at runtime. This illustrates that static (compile time) dependency is between client and interface, not client and interface-implementer.

Note that the use of an interface ensures logical compatibility in the client-interface and interface-implementer relationships. Interface use makes no guarantee about the behaviour of the implementing class in terms of what operations are carried out when methods are executed. This is no more or less of a guarantee of *good behaviour* than a dependency on a concrete type.

8.2.4.2 Abstract Classes

Abstract classes are non-instantiable partial implementations. These can be used as a code-sharing mechanism to reduce duplication in a hierarchy by storing common implementation details in a shared ancestor, without creating full classes. Abstract classes are used to consolidate common behaviour in hierarchies.

Abstract classes can define methods and other functionality as a class, but may also leave some details undefined by including *abstract methods* as an interface does. Abstract classes are implemented by *extension* (class inheritance) and occupy a mid-point between the class-inheritance motivation of code-reuse and interface-inheritance motivation of decoupling type specification from implementation as discussed in *Design Patterns* (Gamma et al., 1994).

8.2.4.3 Marker Interfaces

So-called *marker* or *tag* interfaces contain no methods; these interfaces may indicate that an object has a capability, but do not enforce this via definition of abstract methods.

A key example of this use of interfaces is the *Cloneable* interface in Java. This interface must be implemented by any object which overrides the *clone()* method defined in the universal root type, *Object*. In this case, the interface is used as a flag by the Java Virtual Machine (JVM) to detect that a specific library method has been overridden. Without the interface, Java's native implementation of the clone operation from *Object* will indicate an error (*CloneNotSupportedException).*

Implementing the *Cloneable* interface has nothing to do with inheriting the *clone()* method signature, since all classes already inherit this signature automatically from *Object* (which does not implement the *Cloneable* interface). Cloning a class can be complex, indeed, the default implementation in the *Object* class is empty, and throws an exception if called. It would be more conventional for developers to *opt in* to define an implementation in their classes.

The reason that *Cloneable* is considered *broken* is summarised by Joshua Bloch - "*…making something Cloneable doesn't say anything about what you can do with it. Instead, it says something about what it can do internally*" (Venners, 2002).

The use of the *Cloneable* interface described fulfils none of the usual functionality of an interface i.e. hiding implementer from client or enforcing implementation on an implementer. Arguably, this use of interfaces is being used to fill in a gap where another language-level mechanism might suffice, such as annotation which was introduced to Java later.

Similarly, the *Serializable* interface has *"no methods or fields and serves only to identify the semantics of being serializable"* (Oracle, 2014b). In the simplest cases, use of this marker interface allows the programmer to depend on the default methods of the serialisation subsystem of the JVM to save and restore objects. However, implementing the *Serializable* interface gives the option to provide more detailed instructions to the JVM in the form of customised read and write methods for a given type.

In summary – marker interfaces may be used as flags for the JVM or compiler to trigger the interest of specific subsystems concerned with operations that may be considered to sit *outside* the programming language.

More recently new ways of adding meta-data to classes such as annotation have been introduced, though it is still thought by some that the use-case for marker cases remain as they are better described by interfaces than annotation. The reason is that subclasses of a 'marked' type are also marked and that the marker interfaces allow runtime checking while annotations do not (Bloch, 2008).

8.2.4.4 Other Uses of Interfaces

A peripheral use of interfaces is as containers for *constant* values. Modern Java practice is to use class references to access constant values. However, it is not uncommon to grant access to constant values by implementation of an interface carrying only constant values, and no method signatures. These interfaces are what Liskov identifies as *convenience inheritance* (Liskov, 1988). However, there is no way to *turn off* the polymorphism granted by implementing the constant interface which may lead to misinterpretation of a design. Specifically – there is no intention that all the implementers and inheritors of a constant interface would be interchangeable in any meaningful way.

The matter of correct interface use is further confused by the continued expansion of features interfaces are used to support. Interfaces are used in the definition of annotations which allow the addition of meta-data to classes. Interfaces have also been incorporated in to the newer functional aspects of Java.

Finally, a facility for *default implementation* has also been added to interfaces – while this feature was defined to have a specific role in maintaining backward compatibility, there is no constraint on how programmers will use this feature in their designs, which may further muddy the waters in terms of how interfaces should be used. Specifically, adding executable code to interfaces conflicts with the aim of separating specification from implementation.

8.2.5 Polymorphism

Polymorphism (meaning *multiple forms*) is a core concept in OO programming. This mechanism allows a single object to have types other than the type of the class of which it is an instance. Conversely, it allows a component or function to interact with different objects if they share a common type.

Polymorphism is historically a description applied to a function or operator, sometimes called *ad hoc* polymorphism (Strachey, 2000), such as the '+' operator which behaves differently in many languages based on input as shown in Table 36. These examples rely on the operator having the capacity to deal with various input types correctly.

A further important property is subtyping, which is a "*substitutability relationship: an instance of a subtype can stand in for an instance of its super-type*" (Taivalsaari, 1996). Since a subtype can be substituted in place of its super-type, code which depends on the behaviour of a super-type may operate on the subtype, reducing the number of changes required to the design/codebase to accommodate the new behaviour (in the form of a new subtype).

Table 36: Example operations with polymorphic operator '+' in Java

Expression	Evaluates To	Operation
"abc" + "def"	*"abcdef"*	String concatenation
"abc" + 1	*"abc1"*	String concatenation with automatic conversion of integer (second parameter) to String
100 + 200	*300*	Integer addition
2 + 3.14	*5.14*	Floating point addition with automatic widening conversion of (first parameter) integer to double

This is shown in Figure 57 where a class accepts a super-type parameter at construction. The parameter is specified as *Average* however the *PausableAverage* subclass defined in Figure 54 may be substituted for its super-type.

```
class AverageWrapper {
    private Average avg;
    public AverageWrapper (Average a) {
        avg = a;
    }
}
```

```
AverageWrapper aw1 = new AverageWrapper (new Average());

AverageWrapper aw2 = new AverageWrapper (new PausableAverage());
```

Figure 56 – Simple Inheritance (polymorphism)

This has two effects. Firstly, *AverageWrapper* class has no direct way of telling if the parameter object is of type *Average*, or some subtype of *Average*. However, the parameter is guaranteed to have the same set of defined behaviours (method signatures) as *Average*. Though some of these behaviours may have different effects in a subtype. Secondly, *AverageWrapper* only has access to the behaviours defined by the parameter type (*Average*). If a subtype is passed as a parameter and it has additional behaviours, such as *pause()* (see Figure 54), this additional behaviour is not visible to *AverageWrapper*.

This principle of substitutability is formalised in the Liskov Substitution Principle (LSP) (Liskov, 1988). Liskov asserts that uniformity of operations (interface signatures) is not sufficient to maintain substitutability of subtypes, it must be the case that *"*[the same] *operations do the same things"*. Liskov illustrates this point by comparing a stack and queue that have the same operational signatures, but behave differently in practice.

8.2.6 Dynamic Binding

Dynamic binding is a mechanism which allows the specific implementation of a type or method call to be *looked up* at runtime. This allows increased run-time flexibility. Gamma et al. present an object-oriented approach to polymorphism where they stress the importance of *dynamic binding*, which is the ability to "*substitute objects that have identical interfaces for each other at run-time*" (Gamma et al., 1994). This is also the definition of polymorphism in the glossary of their book, Design Patterns.

8.2.7 Reuse versus Polymorphism

Reuse and polymorphism are discussed as 'two sides' of inheritance. These concepts are not unique to inheritance. Reuse addresses an implementation-level concern. Reuse reduces repetition and thus error, and speeds implementation. The usefulness of this approach is illustrated by the similar libraries of functions to emerge in most general-purpose programming languages. Re-use is a key driver of the *composition* technique where small units of code (functions, objects, etc.) are built into larger components much like 'off the shelf' components.

Polymorphism on the other hand relates to abstraction, which is more design-facing. Polymorphism allows substitutability between similar implementations. If implementations are identical there would be no need to have polymorphism. So, there must be some tolerance for 'acceptable variation' in each group of substitutable types. It is interesting to note that polymorphism also provides reusability of the dependent code, as operations across many similar types only need to be written once. Polymorphism can also be achieved with *abstract types* (interfaces or abstract classes in Java) which may contain partial implementation details.

Inheritance complicates the issue by providing both polymorphism and code reuse. Class-inheritance can be used to extend a type without altering dependent code, and class-inherited subtypes are fully substitutable for their parent. This allows designers to provide new variations of classes with minimal changes to a design – *incremental modification* (Taivalsaari, 1996). This cannot be achieved with interfaces. Furthermore, a variation on an existing type is likely to share much of the original's behaviour, so reuse of super-type code is highly appropriate for new subtypes.

Substitutability can be achieved without sub-classing (reuse). Interfaces allow the designer to provide enough information to program with, without needing to provide implementation details. Interfaces indicate a contract of behaviour, definition of a *role*, or permit partial access to another type. Interfaces may be used in an inheritance hierarchy to define a root public interface or to capture the idea that all members will play additional role(s) indicated by the interface(s).

Dvorak investigated *where* in a hierarchy types are attached – he notes "*Without a shared view of the domain … developers will classify concepts according to their own viewpoint*" leading to "*increasing conceptual inconsistency as we travel down the hierarchy*"

(Dvorak, 1994). This is identified as *conceptual entropy*. Dvorak further notes that "*the effects of identifying sub-classes and super-classes are not local to those entities*" – highlighting the potentially non-local effects of hierarchy membership.

A key difference between interfaces and inheritance is when they can be introduced, and the (effort) cost of introduction. Inheritance can be used to create a 'modified' version of a class or concrete class and preserve substitutability even after the initial stages of design. In contrast, adding new interfaces to allow substitutability may involve many changes within a design.

8.3 Appendix C – Presence of Abstract Types in Selected Programming Languages

This is a comparison of the presence of abstract types and interfaces in various programming languages from a source at http://rosettacode.org/wiki/Interface, [Accessed: 2015/01/8]. Entries in the table are based on the information provided on the site, not the author's familiarity with the various languages.

Table 37 - Comparison of Support for interfaces and abstract types in various programming languages

	Abstract Type		**Interface**	
Language	*Supported*	*Can be Simulated*	*Supported*	*Can be Simulated*
ABAP	Yes	-	Yes	-
ActionScript	No	Yes	Yes	-
Ada	Yes	-	Yes	-
Agda	No	-	No	Yes
Aikido	Yes	-	Yes	-
Argile	Yes	-	Yes	-
AutoHotkey	Yes	-	No	-
BBC BASIC	Yes	-	No	-
C	No	-	No	Yes
C#	Yes	-	Yes	-
C++	Yes	-	No	Yes
Caché ObjectScript	Yes	-	No	Yes
Clojure	No	-	Yes	-
COBOL	No	-	Yes	-
Common Lisp	No	Yes	No	Yes
Component Pascal	Yes	-	No	Yes
D	Yes	-	Yes	-
Delphi	Yes	-	No	Yes
DWScript	Yes	-	Yes	-
E	Yes	-	Yes	-
Eiffel	Yes	-	No	Yes
Fantom	Yes	-	No	Yes
Forth	No	Yes	No	Yes
F#	No	Yes	No	Yes
Genyris	Yes	-	No	Yes
Go	No	-	Yes	-
Groovy	Yes	-	Yes	-
Haskell	No	Yes	No	Yes
Icon and Unicon	Yes	-	No	-
J	No	-	No	-
Java	Yes	-	Yes	-
Julia	Yes	-	No	-
Lasso	No	Yes	No	-
Logtalk	No	Yes	Yes	-
Lua	No	Yes	No	-
Nemerle	Yes	-	Yes	-
NetRexx	Yes	-	Yes	-
NewLISP	Yes	-	No	Yes
Nim	Yes	-	No	-
Nit	Yes	-	Yes	-
Objeck	Yes	-	No	Yes
OCaml	Yes	-	No	-
ooRexx	Yes	-	Yes	-
OxygenBasic	Yes	-	No	-
Oz	No	Yes	No	Yes
PARI/GP	No	-	No	Yes

Pascal and Object Pascal	Yes	-	Yes	-
Perl	Yes	-	Yes	-
Perl 6	Yes	-	No	Yes
PHP	Yes	-	Yes	-
PicoLisp	Yes	-	No	-
Python	Yes	-	Yes	-
Racket	No	-	Yes	-
REBOL	Yes	-	No	-
Ruby	Yes	-	No	Yes
Scala	Yes	-	Yes	-
Seed7	Yes	-	Yes	-
Sidef	Yes	-	No	Yes
Standard ML	No	Yes	No	Yes
Tcl	No	Yes	No	Yes
Visual Basic	No	-	Yes	-
Visual Basic .NET	Yes	-	Yes	-
zkl	No	Yes	No	Yes
Yes	42	12	28	23
No	21	Cumulative	35	Cumulative
%	67%	86%	44%	81%

8.4 Appendix D - Survey Questions

1	How important is each of the following in ensuring the quality of software design in your development process? *Personal Experience; Design Guidelines; Software Metrics; Design Tools; Peer Review; Team Review; Technical Lead / Expert Review; Other*	Likert
2	Please use this space to provide any details that would help to support or clarify your answers above.	Free Text
3	When making design decisions, how confident are you that you have chosen the best option from the alternatives you have considered?	Likert
4	What types of problem make it difficult to have confidence in a design decision?	Free Text
5	What are the key features that allow you to recognise good design in your own work and in the work of others?	Free Text
6	How important are the following elements in your work? *Functional Correctness; Ensuring Design Quality; Other Tasks*	Likert
7	In the course of your software development work, approximately what percentage of your effort is split between the following tasks? *Ensuring Functional Correctness; Ensuring Design Quality; Other Tasks*	Number (%)
8	What software development methodology most influences your approach to design? *Process based (e.g. Waterfall); Iterative (e.g. RUP); Agile based (e.g. Scrum); Other*	Multi-Choice + Other
9	If you have any other comments about how development methodology affects design quality in your work, please note them here.	Free Text
10	How important is Program to an Interface in your design work?	Likert
11	How important are the following factors when deciding if an Interface should be used? *Avoiding dependency on a concrete type; Multiple possible implementations of a class; A class plays multiple roles in a design; Similar behaviour is required from multiple classes; Composing or decomposing existing interfaces; Containing a foreseeable change (flexibility/hotspot); Variation point for dependency injection or business logic; Other*	Likert
12	Metrics or tools which you find useful for managing Interfaces.	Free Text
13	How important is Inheritance in your design work?	Likert
14	Which factors or indicators do you consider to be most important when deciding to use Inheritance? *Code Reuse; Type Substitution; Future Flexibility; Other*	Likert
15	Which factors do you consider to be most important factors when deciding between Object Inheritance (concrete super class) or Abstract Inheritance (abstract super class)?	Free Text
16	When designing inheritance hierarchies, how important are the following features? *Depth of Inheritance Hierarchy; Width of Inheritance Hierarchy; Dependency Inversion (keeping concrete classes at the fringes of the inheritance hierarchy); Other*	Likert

17	Please use this space to provide any additional details about how you deal with Inheritance, such as any Guidelines, Metrics, or Tools that make this task more manageable.	Free Text
18	How important is the consideration of Coupling in your design work?	Likert
19	Are there any Guidelines, Metrics, or Tools that you find especially useful when thinking about Coupling?	Free Text
20	How important is Cohesion in your design work?	Likert
21	Are there any Guidelines, Metrics or Tools that you find especially useful when thinking about Cohesion?	Free Text
22	How important is Size in your design work? *Method Size; Class Size*	Likert
23	Are there any Guidelines, Metrics or Tools that you find especially useful when thinking about Size?	Free Text
24	How important is Complexity in your design work? *Method Complexity; Class Complexity*	Likert
25	Are there any Guidelines, Metrics or Tools that you find especially useful when thinking about Complexity?	Free Text
26	How important are Design Patterns in your design work?	Likert
27	Please provide any other details on how you deal with Design Patterns in your designs in this space.	Free Text
28	How often do you alter or refactor a design due to the following design considerations? *Program to an Interface; Inheritance; Coupling; Cohesion; Size; Complexity; Design Patterns*	Likert
29	Are there any Guidelines, Metrics or Tools that have not already been discussed that you find particularly helpful when designing software or assessing design quality?	Free Text
30	In which country is your software house or production environment located?	Multi-Choice
31	What is the nature of the software development you have taken part in? (Please select all that apply)	Multi-Choice
32	How many years have you been involved in the software industry as a designer/developer?	Multi-Choice (ranges)
33	What formal training have you completed in relation to your role as a designer? *College; University; Masters; PhD; Apprenticeship; Industry Accreditation; Other*	Multi-Choice + Other
34	What programming language(s) do you use when developing software?	Multi-Choice + Other
35	What types of design problems are you most familiar with?	Multi-Choice + Other
36	What role(s) have you carried out in a software development environment?	Multi-Choice + Other
37	If you have any final thoughts or comments on Design Quality or on the survey in general, please note them here.	Free Text

8.5 Appendix E – Semantic Characteristics of Deep Hierarchies

Table 38 - Survey of Semantic Characteristics for deepest sub-tree hierarchies (DIT 5 – 10)

depth	width	Abstraction	Overall Category	Structure	Framework Hook	Specification/API Tied
5	5	Enable observer pattern	Cross Cutting Concern	Observable		
5	7	AST declaration nodes		Mirror		Java AST
5	8	Mutable Tree nodes		Composite	Yes	
5	8	Start-up configuration				Plugin API
5	11	GUI Elements			Yes	
5	12	Document Object Model		Composite		DOM
5	12	Figure in drawing canvas		Composite	Yes	
5	15	Type Bindings for Compiler		Composite		Compiler Notation
5	15	Writing bytes	Generalisation	Pipe and Filter		
5	17	Context sensitive change proposal	Generalisation	Strategy		
5	17	Reading bytes	Generalisation			
5	22	CORBA Interface Definition Language object representation				CORBA IDL
5	23	Diagram Elements	Generalisation		Yes	
5	25	Change to UI or data element		Context (DCI)		
5	26	Plugin 'hook' implementations	Cross Cutting Concern			Plugin API
5	29	XML document nodes		Composite		
5	29	Skeleton for Collections implementations	Generalisation			
5	32	Message to be Passed	Generalisation	Event/Message		
5	35	Making things observable	Cross Cutting Concern	Observable		
5	37	GUI Elements			Yes	
5	40	XML parser/generator			Yes	XML Specification
5	48	Program administration	Grouping			
5	56	Encryption library component			Yes	
5	64	Platform Neutral Commands			Yes	Mozilla NSL
5	78	Language Processing Filters		Pipe and Filter		
5	290	Exceptional Event for JVM	Generalisation	Exceptions		JVM Specification
6	6	GUI Elements			Yes	
6	9	Java Language Elements	Grouping			

6	10	View in MVC - lightweight proxy for model entity		Composite	Yes	
6	18	Wrapper	Grouping	Adapter	Yes	
6	31	Visitor companion hierarchy	Grouping	Mirror		
6	34	XPath Expressions		Composite		Xpath Standard
6	42	UI Non-menu Components		Composite		
6	57	Save/Loadable entity	Cross Cutting Concern			
6	57	XSLT Expressions		Composite		XSLT Standard
6	62	Part of a form, mainly concerned with saving, validating and updating data	Generalisation			
6	93	Pluggable Look and Feel		Mirror		
6	155	Component in a Project	Cross Cutting Concern			
6	183	Dialogue Page	Generalisation	Mirror		
6	200	Grouping	Grouping	Composite		
7	19	Adapter to implement JobChangeListener	Grouping			
7	20	Has lifecycle start, end, and stages	Cross Cutting Concern		Yes	
7	44	Java Abstract Syntax Tree		Composite		Java AST
7	58	UI Element	Generalisation			
7	186	Adaptable Object	Cross Cutting Concern			
8	16	Adapts model to viewable Widget	Generalisation			
8	135	Window inside main UI	Generalisation			
10	542	Observable Analogue	Cross Cutting Concern	Observable		

8.6 Appendix F – Candidate Component Sub-Hierarchies for a Large Hierarchy

Table 39 - EventManager hierarchy candidate sub-hierarchies

Inner Root	**Members**	**% of containing hierarchy**	**Description**
org.eclipse.jface.resource.ResourceRegistry	5	0.4%	Abstract base class for various registries (works with *IPropertyChangeListener*)
org.eclipse.core.commands.State	9	0.7%	Possibly persisted, state information for things that 'toggle' (works with *IStateListener*)
*org.eclipse.jface.viewers.*BaseLabelProvider	126	9%	Maps model elements to optional images and/or Strings, notes that it implements an *IBaseLabelProvider* interface indicating perhaps that it is also a grouping inheritance root type – which is all the worse as this means that it may be extended many times, constraining all of these leaves to the *EventManager* hierarchy (works with *ILabelProviderListener*)
org.eclipse.core.commands.AbstractHandler	200	15%	Connector for pluggable execution handler (works with *IHandlerListener*), has a direct interface *IHandler2* which has other children e.g. *org.eclipse.ui.internal.handlers.E4HandlerProxy* which implements *IHandler2* and then uses a delegate to pass implementation.
org.eclipse.ui.SubActionBars	6	0.4%	Generic implementation of *org.eclipse.ui.IActionBars* used by a (UI) part to access menu, toolbar and status line (works with *IPropertyChangeListener*). This interface is more specific to the class.
org.eclipse.core.commands.common.HandleObject	8	0.6%	Creates handles for things that may not yet exist so that simple operations can be performed on them (e.g. comparison, listener attachment). Seems incidental this this uses *EventManager* as update mechanism.
org.eclipse.jface.action.AbstractAction	820	60%	Implements *org.eclipse.jface.action.IAction* which represents the non-UI side of a triggered command, together with a further abstract sub-class, *org.eclipse.jface.action.Action*. This is a substantial sub tree with a clear separate responsibility
org.eclipse.ui.part.WorkbenchPart	160	12%	Abstract base implementation of all workbench parts (works with *IPropertyListener*). Implements several other interfaces including interfaces based on *org.eclipse.ui.IWorkbenchPart* which appear to be a more defining super-type.

8.7 Appendix G – Large Hierarchy Summaries

Table 40 - Large Hierarchy Summaries

Hierarchy Root	Member Count	Description	Category
java.lang.Throwable	650	is a feature of the Java exception handling system and the hierarchy is comprised mainly of simple message wrappers. The main purpose of the hierarchy is as *flags* (the type of exception), which is conveyed by defining ever more specialist types in the hierarchy. With this in mind, this hierarchy might expand to an arbitrary size without much increase in complexity as more kinds of (ever more specific) exceptions are defined.	**Documenting**
org.eclipse.osgi.util. NLS	311	NLS documents messages type to facilitate localisation. While the root type holds common functionality, the leaves are very simple and all at depth one.	**Documenting**
org.eclipse.core. runtime. PlatformObject	602	Root is a convenient abstract implementation of *org.eclipse.core.runtime.IAdaptable*. Documentation notes "*In situations where it would be awkward to subclass this class, the same effect can be achieved simply by implementing the IAdaptable interface and explicitly forwarding the getAdapter request*"[19]. Good (not perfect) use of abstract types where there are obvious branches – this is not the case where there are leaves-of-leaves at the fringe of the hierarchy. It is interesting to note that, in the types that implement the *IAdaptable* interface (60 interfaces, 68 classes), some do delegate as the documentation advised, but others implement the functionality locally, or rely on an inherited implementation.	**Grouping. Rule 3 compliant.**
org.eclipse.jface. dialogs.DialogPage	460	Root has a similarly named interface, (*org.eclipse.jface.dialogs.IDialogPage*), which is not implemented elsewhere, however it is extended by other interfaces – so this may be the primary reason for its existence. Mostly using abstract types where there are branches, some sub hierarchies are exception to this, but are otherwise suitable for the hierarchy (e.g. *org.eclipse.ui.dialogs.WizardNewFileCreationPage*).	**Generalisation (is-a). Rule 3 compliant, but not a grouping hierarchy.**
javax.swing. AbstractAction	166	An expanded version of *ActionListener* and is generally sub-classed by inner classes representing actions in UI elements. While the utilitarian nature of the top level abstract class (and interface) suggest grouping, there is little variation in the public interface between types. In addition to having many subtypes within Java, this hierarchy is extended in other applications in the corpus (*argouml*, *columba*, *freecol*, *freemind*, *jhotdraw*). Possible comparison?	**Implementation generalisation (is-a). Rule 3 compliant, not a grouping hierarchy.**
com.aelitis.azureus.ui. common.table.impl. TableColumnImpl	182	Represents table columns in this application – inherits many methods and some constants from multiple interfaces representing views of a column (model, rendering information). Variations on a type – little overriding in leaves, and mostly the same methods (*refresh()*, *fillTableColumnInfo()*). Not clear why the root is not an abstract class.	**Generalisation (is-a).**

[19] http://grepcode.com/file/repository.grepcode.com/java/eclipse.org/4.2/org.eclipse.equinox/common/3.6.100/org/eclipse/core/runtime/PlatformObject.java#PlatformObject

8.8 Appendix H - Structural Patterns for Inheritance Hierarchies

Metrics and models of inheritance have not penetrated in the decision-making space where practitioners *think on their feet*. These tools are generally applied after the fact or part of software lifecycle. Traditional guidelines do affect decision making, but require interpretation – which is a source of disagreement within the design community.

By examining the corpus of inheritance hierarchies, recurring patterns can be identified – these might be described as *what lot of developers usually do, a lot of the time*. Much like other guidance – these patterns are intended to be informative, but also to give practitioners pause before breaking them.

This differs from the metrics approach as there is no intention to summarise all systems in a single average or normal value. These patterns are structural, like inheritance - which makes them more suitable for considering some aspects of inheritance use. More importantly – compliance with the patterns is observed *in the wild* and there is some evidence that deviation from the patterns leads to undesirable outcomes.

8.8.1 Pattern 1

Single parent-child inheritance is allowed - for whatever purpose.
Specifically: Inheritance hierarchies which consist of two members – one super-type with a single subtype.

Reasoning: Many hierarchies are not larger than this - the design cost to undo is small, there may not be enough information at this stage to assess the suitability of inheritance. The complexity that can be introduced by this kind or inheritance is low, so take advantage of the convenience of inheritance.

Population Presence: 40.12% (979 of 2440) of hierarchies examined are in this category, being small, these contain only 8.55% (1958 of 22913) of all hierarchy members examined. Figure 57 indicates the diversity of the root types in these hierarchies.

Many of these extensions are to library types in which case, extension may be the standard mode of access. The figure shows all size two hierarchies broken down by the origin of the root type and the kind of the root. The first column shows that there are just under 300 hierarchy roots that are classes from the Java library. These are broken down further by the pattern of method calls directed to the hierarchy from the rest of the system. Note that even at depth one, the variation in method invocations indicates a variety of uses.

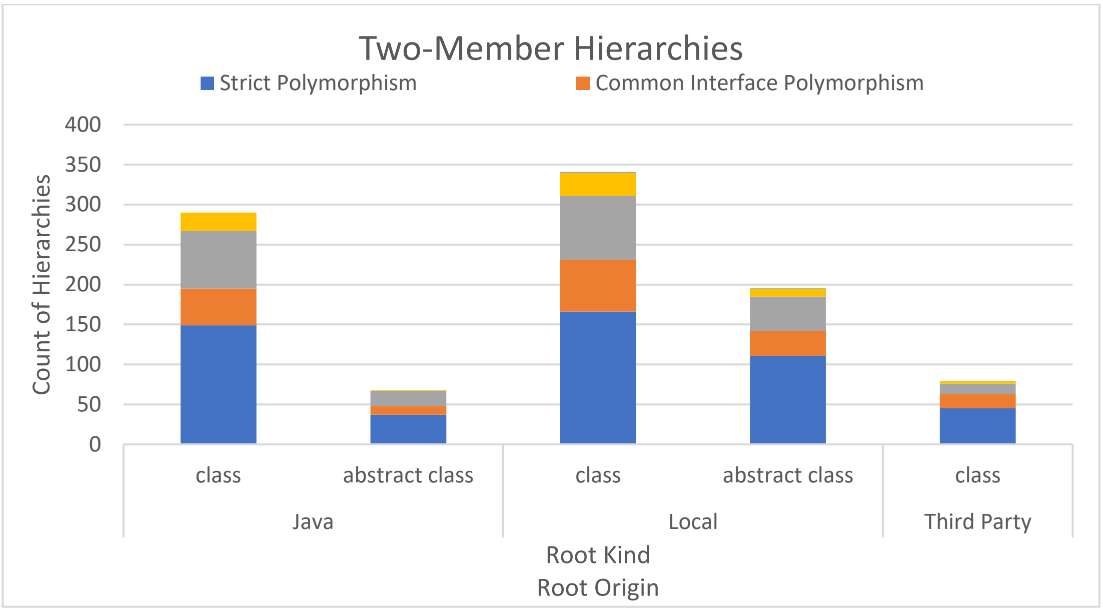


Figure 57 - Breakdown of high level characteristics for 2-member hierarchies

Link to Guidance: Inheritance use is encouraged to reuse existing code, while this can create problems, many of these do not occur until hierarchies grow. There are many recognised uses of inheritance and a hierarchy with only two members could represent any of these – indeed the kind of inheritance may not be entirely clear to the implementer on the initial extension.

An initial extension of a type may indicate the desire to preserve the smallness/single responsibility/cohesion of the super-type while indicating in the design that a new type is required. At minimum, this satisfies concerns about size (not bloating the super-class) and documents the requirement for the new type in the design – which may inform later design changes.

8.8.2 Pattern 2

Non-leaf types should be abstract.

Specifically: The root type and subsequent intermediates before the leaves of the tree should be abstract. This should also be enforced when adding new leaves to the hierarchy. This may involve adding more abstract classes.

Reasoning: The motivation for using inheritance over interfaces is the reuse of implementation. The sole purpose of the intermediate types in a hierarchy is to store implementation – so this should be explicit in the structure. This allows for behaviour reuse, and reduces concrete dependencies. Extracting abstract types by default will force the implementer to document how certain they are about the similarity of leaves. Additionally, this allows separate responsibilities to be stored in separate abstract intermediates, decoupling them from each other. Finally, revision of the inner structure of a hierarchy encourages re-assessment of placement of behaviour in the hierarchy, avoiding the *un-factored hierarchy* smell (Suryanarayana et al., 2015).

Population Presence: Of the hierarchies examined, 54.3% (1325 of 2440) contain abstract classes. Figure 58 shows the presence of abstract types in these hierarchies. The legend indicates how contiguous the inner tree structure is in percent. Green bars indicate continuous abstract hierarchies – these are present across all hierarchy size ranges though they do start to become more fragmented as hierarchies get larger. Blue indicates *almost contiguous* abstract tree structures within a hierarchy. Notably, all very large hierarchies (177-1366 members range) contain abstract types. In large hierarchies, abstract types are generally organised near the root of the tree.

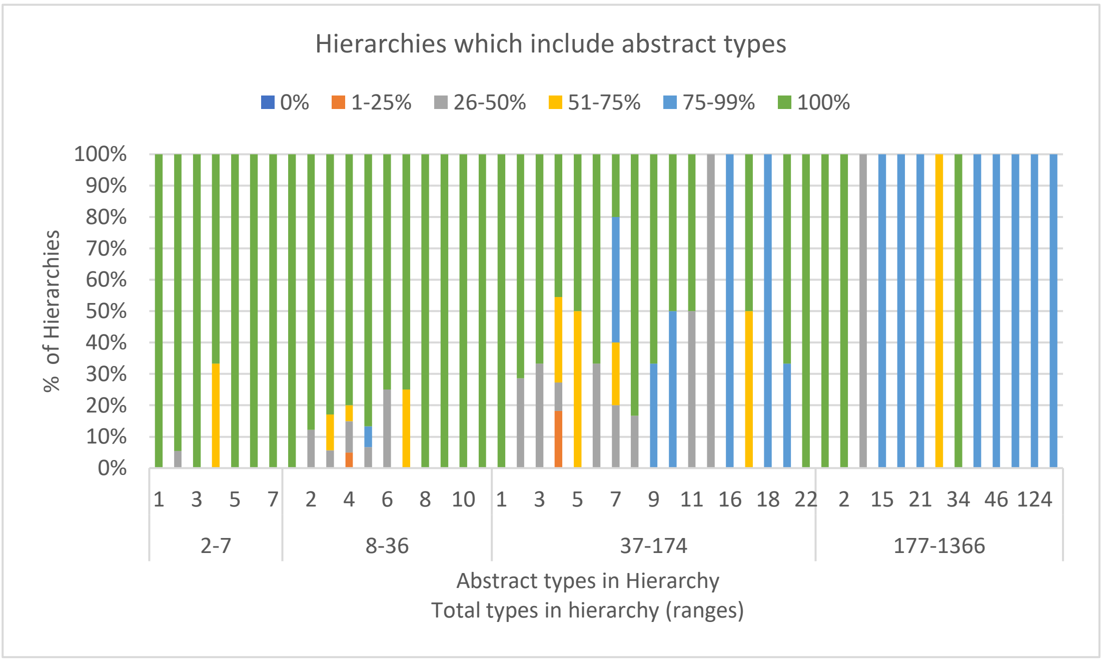


Figure 58 - Hierarchies which include abstract types

The highest level of horizontal expansion is seen in *fan* hierarchies - 65.2% (496 of 760) of which have abstract roots. This rule offers a little more protection to the many subtypes in these wide structures, by ensuring that they depend on an abstract type.

Link to Guidance: This approach excludes the possibility that one concrete type will depend on another, retaining implementation reuse, while reducing direct coupling - this is compliant with the DIP "*depend upon abstractions*" (R. C. Martin, 1996). In addition, this approach moves the change-prone concrete types to the leaves of the hierarchy, encouraging stable intermediates – linking the rule to the OCP (R. Martin, 1996), the approach advocated by the Smalltalk community "*The top of the class hierarchy should be abstract*" (Johnson and Foote, 1988), and Riel's "*All base classes should be abstract*"(Riel, 1996).

8.8.3 Pattern 3

Grouping super-types should be capped with an interface.

Specifically: If using inheritance for a grouping abstraction, add an interface on top of the grouping super-type class to allow type compliance without requiring inheritance of the default implementation – the interface *must* be implemented for compliance, while behaviour *may* be acquired by composition or inheritance.

Reasoning: Grouping super-types are an implementation convenience – so make this decision easy to opt out from (e.g. acquire instead by delegation). This also avoids the need to *stack up* required abstractions by implementing them in sub-trees, when the abstractions are unrelated.

Figure 59 shows two solutions for the same design problem. In both cases, there is an existing hierarchy with an abstract class root A. The second hierarchy with root abstract class (B) is required to comply with the 'native' interface provided by A. In the left panel this is achieved by appending the root of the deficient hierarchy to the existing definition. The attachment could have been made anywhere in hierarchy A to achieve this effect.

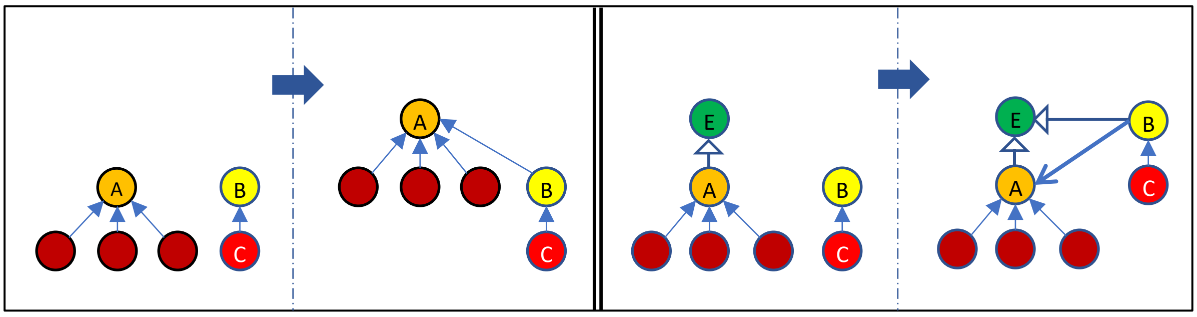


Figure 59 - Illustration of pattern 3 (red = concrete class, yellow/amber = abstract class, green = interface)

In the right hand panel, pattern 3 is in force on the grouping hierarchy. The option now exists for B (and thus C) to comply with the published interface E, while optionally re-using the implementation of this behaviour in A.

Population Presence: Estimating the *at risk* population of grouping hierarchies can be done by counting the hierarchies where the root implements no interfaces AND method invocations from the rest of the design are to root methods: 363 hierarchies with 3029 hierarchy members. In these cases there is evidence that all invocations are polymorphic, and concrete dependency on the root type.

Eclipse contains an example that is especially bad as it creates API commitments that are not easily changed (*org.eclipse.core.commands.common.EventManager*). This hierarchy has 1366 members (depth 10, 542 wide). The abstract root class is an implementation of an Observer-style designed for "*optimising memory and performance … guaranteed to be thread-safe.*"

So, this is definitely a *grouping* hierarchy – the root type tells us very little about the children, other than that they can be observed (notified of events). While there are many direct descendants which benefit from easy access to this behaviour, there are also several large sub-hierarchies within the hierarchy that appear to be attached solely for the sake of convenience. This comes at a price, however. Individual sub-hierarchies each have their own reasons to grow and change – which violates SRP at the hierarchy level. Additionally, by including diverse sub-hierarchies, the overall structure becomes more difficult to understand.

Eight possible sub-hierarchies within the *EventManager* hierarchy are shown in Table 39 (Appendix F), these describe diverse concepts such as – property caches, UI elements, handles for unloaded components, and controller elements – and range in size from 5 to 820 members. Many of these sub-hierarchies are distinct enough (variation on a type hierarchies) to merit considering separation from the top level hierarchy. If all the identified sub-trees were removed by composition, this would leave just 32 members (31 subtypes, 2%) of the original class inheritance hierarchy. In the case of *EventManager*, the developers are aware of the problem with this hierarchy but are unable to make changes due to API commitments. Pattern 3 would prevent exactly this sort of problem.

While not unique to Java, this problem may be more common where there is no option to multiply inherit. This problem may only become obvious after the fact; however vulnerable hierarchies are those where candidate new leaves types are likely to already have other super types.

The next largest hierarchies are summarised in Table 40 (Appendix G) – which shows that this pattern is already in effect in some large hierarchies in eclipse. A good example of this is in the hierarchy with root *org.eclipse.core.runtime.PlatformObject* (602 members). This hierarchy has a root interface, with a default implementation, which notes – "*In situations where it would be awkward to subclass this class, the same effect can be achieved simply by implementing the IAdaptable interface and explicitly forwarding the getAdapter request*". The hierarchy contains a variety of mechanisms for accessing implementations of the root interface – direct inheritance, delegation, local implementation, indirect inheritance. These are all possible (increasing the practitioner's design options) because of the separation of the root interface from the default implementation.

Link to Guidance: Much of the design guidance is pragmatic – it suggests how to prevent future design issues while providing some input for consistency and disciplined design. This rule allows the designer to achieve reuse, without introducing unnecessary concrete dependencies. This form of separation of desirable outcome and mechanism is similar to modularity (Parnas, 1972), depending towards stability or abstraction (R. C. Martin, 1996), or programming to interfaces (Gamma et al., 1994). The choice to use inheritance in this case is itself a *design decision* that the rest of the system can be protected from.

As discussed above – this sort of protocol inheritance is technically an *is*-a relationship in the sense of compliance with the root behaviour and substitutability (LSP). However, *grouping* inheritance adds a protocol to hierarchy members, without much semantic constraint. This pattern recognises this less *defining* relationship by reducing coupling with the default implementation.

8.8.4 Pattern 4

An abstract class root-type without an interface should be used only to represent a 'complete type' sub-typing hierarchy (is-a relationships).

Specifically: A hierarchy root without an interface (ideally an abstract class) should be a well-defined abstraction that contains minimal, very high level common code that is common to all subtypes. This code may even be static methods and constants - essentially an interface-like abstraction with some documentation.

Reasoning: Simple re-use hierarchies can be concrete if there are only two members (as per rule one), but as soon as more than one type relies on the same code, separate mechanism of reuse from the reuse dependency (pattern 2). However, this may leave some doubt as to whether the hierarchy is grouping, or sub-typing.

While pattern 3 warns against committing to a grouping super-type. This pattern captures an alternative situation where there is a clear (modelling or implementation) super-type that defines the essential nature of its subtypes - which are going to be specialisations of that root type.

Population Presence: 561 (23%) of hierarchies have an abstract root and no interface – while this is not confirmation that these are not grouping hierarchies, it does indicate that an abstract-class root is considered *abstract enough* for some purposes by practitioners. For example:

org.apache.tools.ant.ProjectComponent (499 members) - The root describes members as 'components of a project', and defines a few generic facilities such as logging and cloning. Good (not perfect) inclusion of abstract classes at branch points.

org.eclipse.jface.window.Window (361 members) – abstract spine. Many intermediate *hub* types are non-abstract – although documentation indicates this this may be an oversight (e.g. *ViewSettingsDialog* is documented as an abstract class, but it is not).

Link to Guidance: One of the promises of OO design is that it captures meaning from the problem or solution domain. There is some debate over how closely the types defined in the design should be drawn from either domain – as some design elements are clearly implementation abstractions (GRASP – Pure Fabrication (Larman, 2004)). This pattern captures the level of certainty – in this case the documentation is (pragmatically) an abstract class, rather than an interface.

8.8.5 Pattern 5

Inheritance between concrete types is a last resort - when the existing code cannot be changed (OCP) or if the new leaf completely replaces its ancestors (versioning).

Specifically: In some cases, the only mechanism available to allow incremental evolution of a base class is to allow one concrete type to inherit from another. This is reasonable if the new subtype is going to replace its super-type.

Reasoning: The trade-off here may involve modification constraints, reuse, and avoiding complexity. Incremental evolution suggests that each subtype *modifies* the super type while retaining its overall purpose.

In the case where there are going to be a sequence of versions of a class (line shaped hierarchy, no branches), the complexity remains relatively low – there is a clear motivation for each *version*. If branching does occur, then this should be an indication that there is more going on than simple incremental modification.

Similarly, in the case where it is known that only the latest version of the class (the single leaf) is going to be instantiated, the risk of side effects from the interaction of up-or down calls can be well-documented in the hierarchy.

Population Presence: Line hierarchies are common 1043 (43%), but generally not deep (979 at depth 2, 58 at 3, 5 at 4, 1 at 5). Deeper line hierarchies illustrate this use of inheritance. It is not clear if this form of *versioning* is present in more complex hierarchies where it would be more difficult to isolate.

java.awt.KeyboardFocusManager (5 members) While not entirely related to versioning, the documentation indicates that the current leaf type (*DelegatingDefaultFocusManager*) is current version compliant, but delegates to a deprecated class further up the hierarchy.

org.omg.CORBA.portable.Delegate (4 members) The base type implements an standard API (CORBA Object), while later subtype notes "*It extends org.omg.CORBA.portable.Delegate and provides new methods that were defined by CORBA 2.3*" (org.omg.CORBA_2_3.portable.Delegate).

Link to Guidance: Reuse is a commonly stated benefit of using inheritance. However, as noted by Taivalsaari, class inheritance allows a new class to be substituted for its super-type, *incrementally modifying* (Taivalsaari, 1996) the program after it has been written. This is another use mode of inheritance and should be recognised as such.

8.8.6 Pattern 6

There are different kinds of similarity – more examples will clarify what kind of hierarchy you are growing.

Specifically: When common code is identified in sibling leaves, consider if this is co-incidental or real commonality.

Reasoning: The kind of similarity modelled by a changing hierarchy may only become clear as more leaves are added:

- If the hierarchy is grouping - add an interface above the grouping super-type as in Pattern 3
- Only pull out common code to the hierarchy if it is related to the common abstraction, including decomposing common method fragments (Johnson and Foote, 1988).
    - Otherwise, refactor to composition would avoid contaminating the abstract super-type with incidentally common code.
    - Pay attention to the purpose and function of the root abstraction to assess if any sub-hierarchies differ enough to be extracted to new hierarchies.

Population Presence: N/A. Pattern 6 applies as new leaves are being added.

Link to Guidance: This rule represents an evolution concern - conceptual entropy, while the future is often uncertain, attempting to add a new type to an existing hierarchy may be very revealing in terms of how well the design matches the needs of the practitioner. Additionally, each new leaf may broaden or narrow the scope of the root abstraction in the system. This begins to incorporate the notion of *where things belong*, so conceptual entropy is related to cohesion, and the impact of new information.

8.8.7 Patterns Summary

This section reviews the patterns presented above in the context of the corpus used in this study:

8.8.7.1 How much of the corpus is covered by these patterns?

Note that the patterns are not mutually exclusive e.g. a grouping hierarchy should be capped by an interface, but it should also have abstract non-leaf layers. So, some of the populations may overlap. As for being important most of the time – these cover the main motivations for using inheritance, namely polymorphism, reuse, grouping, variation, and evolution.

8.8.7.2 How much evidence there is that the patterns are already being followed?

There are concrete examples for each category noted above. These are isolated using the criteria noted under the *Population Presence* section under each pattern, and are summarised in Table 41. Since these patterns are primarily structural, and derived from the corpus, it may seem obvious that they would be represented in the corpus. Nevertheless, it is notable how few patterns are required to capture most of what is observed in the corpus.

Table 41 - Estimated presence of structural patterns for inheritance in corpus

No	Pattern	Hierarchies	Members
1	Single parent-child inheritance is allowed - for whatever purpose	40%	9%
2	Non-leaf types should be abstract	54%	19%
3	Grouping super-types should be capped with an interface	Few large hierarchies	44%
4	An abstract class root-type without an interface should be used only to represent a sub-typing hierarchy (is-a relationships).	23%	26%
5	Concrete inheritance may be unavoidable – when used to version the root of the hierarchy	Unclear	Unclear