Empirical Studies of Structural Phenomena Using a Curated Corpus of Java Code

by

Hayden P. Melton
B.E.(Hons), Software Engineering

Submitted in fulfilment of the requirements for the degree of

Doctor of Philosophy

Deakin University

May 2017

DEAKIN UNIVERSITY
ACCESS TO THESIS - A

I am the author of the thesis entitled

Empirical Studies of Structural Phenomena using a Curated Corpus of Java Code

submitted for the degree of

PhD (Information Technology)

This thesis may be made available for consultation, loan and limited copying in accordance with the Copyright Act 1968.

'I certify that I am the student named below and that the information provided in the form is correct'

Full Name: Hayden Melton
(Please Print)

Signed: [illegible]

Date: 9th Dec 2016

DEAKIN UNIVERSITY
CANDIDATE DECLARATION

I certify the following about the thesis entitled (10 word maximum)

Empirical Studies of Structural Phenomena using a Curated Corpus of Java Code

submitted for the degree of PhD (Information Technology)

a. I am the creator of all or part of the whole work(s) (including content and layout) and that where reference is made to the work of others, due acknowledgment is given.

b. The work(s) are not in any way a violation or infringement of any copyright, trademark, patent, or other rights whatsoever of any person.

c. That if the work(s) have been commissioned, sponsored or supported by any organisation, I have fulfilled all of the obligations required by such contract or agreement.

d. That any material in the thesis which has been accepted for a degree or diploma by any university or institution is identified in the text.

e. All research integrity requirements have been complied with.

'I certify that I am the student named below and that the information provided in the form is correct'

Full Name: Hayden Melton
(Please Print)

Signed: [illegible]

Date: 9th Dec 2016

Abstract

Empirical studies of structural phenomena are performed using a curated corpus of Java code that was conceived, designed and evolved by the author for the purposes of conducting this research. What is found is that long dependency cycles involving many source files are quite prevalent in real-world Java software, despite much advice in the instructional literature on object-oriented design to avoid them. Approaches are proposed to quantify the extent of dependency among the source files in such cycles, including schemes for classifying such dependencies on the basis of their pathology (e.g., whether two classes are intrinsically dependent on one another), and schemes to quantify their connectedness. Specific refactoring techniques are proposed for breaking these cycles, and a tool inspired by the poka-yoke paradigm in manufacturing is proposed to ensure cycles are not created by software engineers in the first place. As for the cause of cycles (and large transitive dependencies in general), further empirical studies are performed in an attempt to link the use of non-private static members to them, and to quantify the extent to which default implementations of interfaces—which may be associated with large transitive dependencies—appear in dependency injection schemes. The thematic contributions of the work are (1) that empirical studies of *just* structural attributes (and not external quality attributes) of software can provide us with new and useful insights into the practice of software design that may in turn help to focus efforts in the areas of tool support and empirical validation of design principles. And (2), that a carefully curated corpus of real software is needed to ensure these insights are convincing. The more concrete contributions of this work are its results relating to cycles, the corpus it yielded that is now publicly available and in wide-use, and the various tools developed along the way.

Acknowledgements

I consider myself extremely lucky to have had Professor John Grundy as my supervisor here at Deakin University. Besides being an absolutely top notch academic he is a thoroughly decent, kind and understanding human being. He helped me to get this thesis over the finish line. I must also thank Associate Professor Ewan Tempero who supervised me at the University of Auckland, where the vast majority of this research was actually conducted. He helped me to get funding for this research, gave me the latitude to find my own research topic and once I had done so helped me to publish the papers contained in this thesis. Without those publications, there would not be a finish line to cross. I must also thank Dr. Hong Yul Yang, Dr. Mohamed Almorsy Abdelrazek, all of my coauthors, those who gave me feedback on this research as it was being conducted, and those who (surely unbeknownst to them) have renewed my confidence in the meaningfulness of this work by citing it and building upon it in their own research.

Publications

1. Hayden Melton and Ewan Tempero. Identifying refactoring opportunities by identifying dependency cycles. In Proceedings of the 29th Australasian Computer Science Conference-Volume 48, pages 35–41. Australian Computer Society, Inc., 2006.

2. Gareth Baxter, Marcus Frean, James Noble, Mark Rickerby, Hayden Smith, Matt Visser, Hayden Melton, and Ewan Tempero. Understanding the shape of Java software. In ACM Sigplan Notices, volume 41, pages 397–412. ACM, 2006.

3. Hayden Melton. On the usage and usefulness of OO design principles. In Companion to the 21st ACM SIGPLAN symposium on Object Oriented programming systems, languages, and applications, pages 770– 771. ACM, 2006.

4. Hayden Melton and Ewan Tempero. The CRSS metric for package design quality. In Proceedings of the thirtieth Australasian conference on Computer science-Volume 62, pages 201–210. Australian Computer Society, Inc., 2007.

5. Hayden Melton and Ewan Tempero. An empirical study of cycles among classes in Java. Empirical Software Engineering, 12(4):389–415, 2007.

6. Hayden Melton and Ewan Tempero. JooJ: Real-time support for avoiding cyclic dependencies. In Proceedings of the thirtieth Australasian conference on Computer science-Volume 62, pages 87–95. Australian Computer Society, Inc., 2007.

7. Hayden Melton and Ewan Tempero. Towards assessing modularity. In Assessment of Contemporary Modularization Techniques, 2007. ICSE Workshops ACoM'07. First International Workshop on, pages 3–3. IEEE, 2007.

8. Hayden Melton and Ewan Tempero. Static members and cycles in Java software. In First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007), pages 136–145. IEEE, 2007.

9. Hong Yul Yang, Ewan Tempero, and Hayden Melton. An empirical study into use of dependency injection in Java. In 19th Australian Conference on Software Engineering (aswec 2008), pages 239–247. IEEE, 2008.

10. Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. The Qualitas corpus: A curated collection of Java code for empirical studies. In 2010 Asia Pacific Software Engineering Conference, pages 336–345. IEEE, 2010.

Table of Contents

Chapter 1 Introduction

It is widely accepted that software structure (as it manifests in source code) affects software quality [Dij68][Par72][SMC74][Lak96][Boo91][Ber93][FP96]. The need to obfuscate code as an anti-piracy measure is perhaps the best example of this. By rearranging the program's structure by removing otherwise descriptive names of classes and methods, altering the structure of looping and conditional constructs and so on, as obfuscation tools do, the program being obfuscated becomes essentially incomprehensible to a human being [CN09]. This makes it nearly impossible for a person with nefarious intentions (e.g., to steal parts of the program for use in other programs, or to circumvent parts of the code intended to check for license files) to reverse engineer an obfuscated program to these ends.

Where things are less well agreed, with respect to software structure, is the precise relationship that specific structural phenomena in source code (e.g., coupling and cohesion, visibility of a module's functions, and so on) have on specific software quality attributes, such as maintainability, understandability, reusability, and so on [FP96]. In the software measurement community the former are often referred to as *internal attributes*, and the latter are often referred to as *external attributes*.[1] Some researchers—especially in the empirical software engineering community—hold the view that the only empirical studies worth doing are those that take measurements of internal attributes and seek to correlate those with measurements of external attributes [Par03].

It is my position that the work in the publications described herein demonstrates that this view held by some in the empirical software engineering community is short-sighted, and that we can advance knowledge in the field through carefully conducted empirical studies of *just* internal attributes. Before I argue my position in this however, I first describe the genesis of this work.

[1] Fenton and Pfleeger define *internal attributes* as "those that can be measured purely in terms of the product, process or resource itself. In other words, an internal attribute can be measured by examining the product, process or resource on its own, separate from its behavior" and *external attributes* as "those that can be measured only with respect to how the product, process or resource relates to its environment. Here, the behavior of the process, product or resource is important, rather than the entity itself." [p.74,FP96].

1.1 The Genesis of this Work

Upon my graduating (with First Class Honours) from the University of Auckland with the Bachelor of Engineering in Software Engineering I took a job as a software engineer at a large—at least by New Zealand standards—locally-owned software company. My experiences at this company were, unequivocally, the impetus for the research in this thesis.

What struck me as a very serious problem, during my almost year long stint at this software company, was how difficult it was, as a software engineer, to make changes to the software products they developed. Whether I was fixing a bug, adding a new feature, or trying to refactor some code, it was seldom immediately clear which source files I needed to edit to accomplish each task. I felt as if I spent a lot of time trying to figure out which source files I needed to edit for each change, and after I had figured out which those were, I was left to wondered why the code had been structured the way it had, and not in a different and "better" way.

The experience I had at this software company was in stark contrast to that which I had while completing programming course work at the University of Auckland. I am not saying that the course work at the University of Auckland left me ill-prepared to work as a professional software engineer, but I am saying that the experience dealing with code in industry was quite different. While University assignments often involved modifying code written by academic staff with PhDs in Computer Science, that had small code bases, and had generally well thought out designs, real code bases in industry were much, much larger and perhaps as noted by Foote and Yoder had evolved in unforeseen ways over many years, leaving them poorly structured and therefore unnecessarily difficult for the software engineers working on them to implement new features and to fix bugs [FY97].

My initial view, in dealing with the code bases of several products at this company was that their designs were too "highly coupled". Even a seemingly trivial new requirement, or seemingly trivial bug, would require quite extensive investigation of which source files required modification, and usually there were a slew of such files per bug or requirement. At this time, as a recent graduate, I had only a fairly informal view of coupling that I later learned to be consistent with that of Fowler's view on it:

that two source files are coupled if changing one necessitates a change in the other [Fow01]. The problem was that it was not entirely clear to me what structural attributes, as they manifest in source code, caused two or more source files to be coupled with respect to change. And this was even after taking course work involving the teaching of Design Patterns, Object-Oriented Design, and so on, as an engineering student at the University of Auckland.

After almost a year at this company, I ended up being awarded a scholarship to undertake PhD research, and would end up leaving it to rejoin the University of Auckland this time as a PhD candidate in Computer Science, with the goal of trying to find answers to the questions I had from my experience in industry on coupling as it manifests in source code, and coupling as it relates to two or more files being modified in relation to the fix for a single bug, or implementation of a single feature. This was not exactly the final direction of the research—and the importance of describing what was tried in one's research but did not work is duly noted[2]—so I will do so below.

My initial idea for relating coupling as it manifests in source code (for now let's call this "structural coupling") to coupling as it manifests as groups of source files being changed together to fix bugs or implement new features (also for now let's call this "change coupling"), was to download a version control repository for a Java project from the open source project hosting website SourceForge. From the project's inception, to its most current revision, I would then repeatedly check out files by their timestamp and commit comment in order to infer change coupling (groups of files changed together would likely have the same commit comment, and approximately the same timestamp). In order to infer structural coupling I would compile the code after each such checkout and run a tool to analyze the Java bytecode to infer various forms of structural coupling among the classes in the project's source files, so I would know the state of various forms of structural coupling (which may have changed due to the checkout) in the program prior to the next modification of its source code.

[2] See http://www.deakin.edu.au/students/research/your-thesis-and-examinations/thesis-structure-options, section 9

There are many different forms of structure coupling in object-oriented software like Java: Bidve and Khare review the various frameworks that have been proposed over the years to categorize these many forms of coupling in object-oriented systems [BK12]. My initial approach instead of building a tool to measure coupling myself, was to look for existing tools that were in the public domain and leverage those. In that way, by collecting a large number of different forms of coupling from such tools, I could try and identify certain forms of structural coupling that were correlated with changes. Most of the tools I found though, operated on Java byte-code (and not Java source-code, because the former is generally easier to analyze) so after each checkout I also had to (re)compile the project's source code.

My investigations had shown, with the computing resources available to me at the time (namely an Intel Pentium 4 3.2 GHz with 1GB of RAM desktop computer, the standard provided to each graduate student in Computer Science by the University of Auckland at the time), that recompiling the entire code base after each check out was going to take far too long, if I wanted to analyze the entire multi-year check-in history of the project. I postulated I might only have to recompile the classes whose source code had actually changed after each check out, but to my surprise in researching the matter I found a work by Lagorio showing that due to the complexities of the name-binding rules in Java, one must actually recompile not only the files that were changed in the check-in, but also all files that *might,* based off names appearing in their source, have transitive compilation dependencies on the files changed in the check-in [Lag04].

I set about to conduct an experiment on the latest revision of one of the SourceForge projects I had downloaded—a popular file sharing application implementing the BitTorrent protocol called *Azureus*—and to my surprise found that a single change in almost any source file in the Azureus code base required, according to Lagorio's algorithm, recompilation of almost its entire code base. Having expected to only have to recompile a few other source files in the project if any given source file was altered, I resolved to determine why this might be. I then also wondered if this problem of having to recompile almost all source files after each change might also true of the other Java projects on SourceForge.

So in attempting to perform the study I had initially intended—i.e., to correlate forms of “structural coupling” with “change coupling”—I had stumbled upon a problem where the structure of some Java projects required that their entire code bases be recompiled after each change. This was the starting point for my *actual* research, as opposed to my initial *intended* research, and as I found the answer to one question, naturally led to another. And this is how the research progressed. One might characterize it as being “curiosity driven”. In the commentary of each of the publications in this research provided in this chapter I describe the path of this research and the connection between the publications by identifying the question each publication seeks to answer, and explaining why the answer to that question naturally leads to another question which is answered in a subsequent publication.

1.2 Open Issues in Software Structure

In the interests of keeping the preceding discussion high-level I have so far alluded to software structure (and the concepts it subsumes such as coupling, internal attributes, and so on) without actually defining it (or them). In this section I define them and explain why, despite structure being an active topic of research in the field of software engineering for over 50 years now—the earliest significant reference I can find discussing software structure is that of J.C. Emery from 1962 [Eme62]—controversies relating to them persist.

1.2.1 The Meaning of Structure

According to the Oxford English Language Dictionary the structure of a thing is “the arrangement of and relations between the parts or elements of something complex”. Software is certainly complex and comprises interrelated parts. As Bass *et al.* discuss in their book on Software Architecture, a software system has not just one but rather a plurality of structures, and the specific manifestation of structure of interest is dependent on one’s goal in assessing that structure [BCK98]. For instance, in assessing the performance of a software system, where that system is geographically distributed and connected by bandwidth constrained network links, an appropriate structure to analyze might be one where the “relations” are network connections, and the “parts” are the separate process comprising that system that communicate over those network connections. Similarly, if one is interested in structure as it relates to a bug where the wrong value has been written to a global variable the "parts" might be the statements in code that either directly or transitively cause data to be written to that variable, and "relations" might be the flow of data through these statements towards the global variable of interest. The analysis pertaining to this form of structure is often referred to a *program slicing* [Tip95].

Sometimes, with reference to the definition of structure, the precise nature of the "relation" among the parts is less clear. Earlier I described a technique whereby code obfuscation tools made code effectively incomprehensible by replacing descriptive class and method names with nonsensical ones. For instance, the classes "JavaLexer" and "JavaParser" in a program for compiling Java code might be renamed "A" and "B", respectively. Is merely renaming identifiers in this manner this a structural change? The answer is "yes" because a there is a relationship between these two classes, implied by their respective names, deriving from the concepts in the domain (metrics seeking to exploit these relationships among words in identifiers have been proposed by Stein *et al.* [SEG+06], and others). Textbooks on compilers describe the process of *lexing* occurring before the process of *parsing*, so from these names alone (assuming they accurately describe the functionality of their respective classes) one can determine a "happens before" relationship, which may aid in a maintenance task such as adding new keywords and constructs to a programming language or fixing a bug. Indeed, Anslow *et al.* [ANMT08] have done some work done along these lines in the area of studying the English names in identifiers that appear in the *Qualitas Corpus*.

If it is accepted that there are many manifestations of structure in the context of software systems, what is the ongoing the controversy in this area of research to which I have alluded? One answer is that there is still no consensus on which specific attributes of structure affect which specific attributes of software quality. And certain studies have shown that it may be more complicated than this even, because there may be additional factors such as choice of programming language paradigm [Hat98] and programmer experience [USH+16][AS04] that play a significant role determining the specific relationship between an attribute of code, and a software quality attribute. Another answer is that there is no consensus on the methods by which we should seek to establish a relationship between a structural attribute and an external quality attribute. Finally—and perhaps most fundamentally of all—the precise nature of many widely-discussed concepts thought to be structural attributes (such as coupling) remains contentious.

1.2.2 Linking Structural Attributes to External Attributes

On the issue of linking structural attributes of source code to software quality attributes such as maintainability, testability, reusability and so on, consider a study by Arisholm and Sjoberg [AS04]. In this study the authors conduct an experiment where the effort to perform maintenance tasks is measured—some subjects being asked to make changes to a program with a centralized control structure, and some other subjects being asked to make the same changes to essentially the same program except with a delegated control structure. What the study finds is that less experienced subjects find the former style of control structure easier to work on, and more experience participants find the latter style of control structure easier to work on. This does seem to indicate that other factors in addition to structure may play a significant role in determining external quality attributes such as maintainability.

A more recent study by Uesbeck *et al.* seeks to link a relatively new language feature in C++, namely Lambda Functions (or simply "lambdas"), to maintainability when compared to using an older feature: iterators [USH+16]. What it finds—perhaps unsurprisingly—is that experience has a major effect on completion time for such maintenance tasks, whether lambdas or iterators are used. The newer language feature, lambdas, however, are found to be more burdensome with respect to maintenance efforts.

What, from a practical perspective though, do the results of the study of Arisholm and Sjoberg and separately the study of Uesbeck *et al.* tell us though? That a software company with less experienced staff—perhaps because they are not willing to pay for experienced ones, or perhaps because the experienced ones don't stay long because they are eventually poached by other companies willing to pay them more highly—should implement their software using only a centralized control pattern? Or that universities should do a better job teaching their graduates the delegated control style? And that universities (and employers, by way of on the job training) are doing a poor job of teaching the concept of lambdas?

Parnas is fairly critical of the types of studies performed in empirical software engineering, along the lines of that performed by Arisholm and Sjoberg and separately by Uesbeck *et al.*, though he does not mention these studies explicitly (his publication precedes these) [Par03]. Essentially, he argues that not much can be learned from watching poorly trained subjects perform software engineering work, because, by definition they are poorly trained. If the delegated control style is "better" than the centralized style, software engineers should simply be better trained to use it. He further argues, essentially, that it should not be necessary to show a style to be better via an experiment involving subjects, and that if there are good, rational arguments for why it is better, we should accept them. He notes, himself having received formal training as an electrical engineer, that no study has ever been conducted to prove the usefulness of Ohm's law in designing circuits, but the electrical engineering community widely accepts that it is a useful technique for doing so.

Parnas also notes that he himself did not conduct any empirical study of his seminal work on Information Hiding, and intimates that the simple example of two differently structured Key Word In Context (KWIC) programs in this work is sufficient to show his theory to be sound [Par72].

Why is it then that many in the empirical software engineering community are so adamant that the studies of the form undertaken by Arisholm and Sjoberg (and Uesbeck et al) are the only *true* way to show connection between internal attributes of software and external quality attributes? Fenton and Pfleeger shed some light on this, arguing that many of the tools, techniques and technologies adopted by the software engineering community have been so on the basis of hype, marketing and folklore (as opposed to adoption on the basis of results from a scientific study, like what might happen in say adoption of a new drug) [FP96]. An empirical study by Hatton on defect density of object oriented programming compared to the older style of procedural programming seems to illustrate this point [Hat98].

Hatton's study on defect density of the two programming paradigms—although published several years before Parnas' work criticizing studies performed in the field of empirical software engineering—seems to be the rebuttal of Parnas' criticisms.

Hatton intimates that while the rational arguments for why object-orientation better reflect the way we as humans think about the world—and therefore may be a more natural way for software engineers to express those thoughts in source code—the empirical evidence for defect density and time taken to perform corrective maintenance was higher in the objected-oriented system as compared to the procedural one.

Interestingly, arguments for adoption of technologies by hype, marketing, folklore (if one is aligned with the views of many in the empirical software engineering community) or by rational argument (if one is more aligned with Parnas' views) persist, even to the present day. The following quote is excerpted from the introductory chapter of recent book by Odersky *et al.* on the Scala programming language [OSV16]: "Fewer lines of code mean not only less typing, but also less effort at reading and understanding programs and fewer possibilities of defects". While one can appreciate Odersky wanting to evangelize the programming language he has created, without (at the very least) elaboration on these claims they do seem facile, and indeed very reminiscent of those that Hatton describes were used to justify the adoption of object-oriented programming. To be sure, *facile* is exactly the right word to describe the argument in this quote, because while Scala may indeed result in a reduction of lines of code, the complexities of its type system, program flow constructs and so on, may actually make it more difficult to understand and more susceptible to the introduction of subtle bugs.

All of this leads us to the question of which is the right approach: that of Parnas, or that of those in the empirical software engineering community like Arisholm and Sjoberg? The work of Kitchenham *et al.* seems to suggest that there is merit in both approaches, and that they are not mutually exclusive [KDJ04]. Kitchenham *et al.* borrow the same-named concept from the field of medicine and apply it to software engineering to come up with *Evidence-Based Software Engineering*. This approach seeks to amalgamate expert opinion (e.g., the rational arguments in the style of Parnas'), with results of empirical studies (e.g., the results of Arisholm and Sjoberg) by, among other things, critically weighting evidence in terms of its credibility. The approach, as promising as it might seem at first sight, is as noted by Kitchenham *et al.* not without its critics, even in other fields where it has existed for quite some time. And sometimes, like in the case of a three-wheeled vehicle as a compromise between a car and a motorbike, the outcome is the worst of both alternate possibilities—it being both less stable than a car, and less agile than a motorbike.

Given the above my conclusion for this section is that there is *not* a consensus on the method by which one should attempt to show a connection between a structural attribute, and a software quality attribute (such as maintainability, reusability, understandability and so on).

1.2.3 The Nature of Structural Attributes

On the issue of the nature of widely-discussed structural attributes such as *coupling* consider the following. In one of the more recent treatments of coupling, Fowler says of it, that two things are coupled if changing one necessitates changing the other [Fow01]. He further notes that two things might be coupled even if there is seemingly no dependency (as far as the compiler or execution of the program is concerned) between them. The example he uses is when code that would otherwise belong in a well-factored, single method, is instead duplicated throughout the program. Any maintenance task to fix a bug in that code would involve modifying that code in each of the source files it appears, and by Fowler's definition, this would imply all those source files were coupled to one another. It is worth noting, Fowler's definition of coupling is not just an anomaly, many authors use the term this was as indicated in the survey paper of Kagdi *et al.* [KCM07]. Indeed the IEEE Standard Glossary of Software Engineering Terminology [IEE90] defines coupling as "the manner and degree of interdependence between software modules" and Fowler's view of it does seem consistent with this.

Where things are inconsistent with respect to this view of coupling though, is in the literature on software measurement (see, e.g., Fenton and Pfleegers' book, and additional references therein [FP96]). This literature seems to exclusively define coupling as an *internal attribute* of software. An *internal attribute* of a thing is one that can be measured from knowledge of the thing alone. An *external attribute* of a thing is one that's measurement necessitates knowledge not only of the thing itself, but knowledge of the environment in which the thing exists. The classic example of an external attribute of software is *reliability*, because it depends not only on the correctness of the software's implementation, but also on the specific features of it utilized by its user, the hardware on which it runs, and any software in the stack (e.g., the compiler, the operating system and so on) on which it depends.

There are many reasons why Fowler's definition of coupling is not consistent with it being an internal attribute of software. One is that whether coupling exists between two or more modules depends on the specific bug being fixed or requirement being implemented. Although not mentioned by Fowler, this is abundantly clear in Parnas' seminal work—which modules change in his KWIC program depend on which one of the future requirements he enumerated is considered. Those future requirements do not manifest in the source code of the program. Further, Parnas' KWIC program is very simple[3], and it has been my observation that in real-software there are oftentimes several choices of source files that can be modified to implement a given requirement. A professional software engineer will evaluate each such choice and the risks associated with it (e.g., the risk of introducing a regression when considering generalizing existing code vs. that of leaving the existing code as-is and writing less invasive additional code that may involve some degree of duplication to implement a new feature). In this respect, coupling as it is defined by Fowler is an external attribute because it depends not only on the specific future requirement, but also on the expert judgment of a software engineer implementing it especially in the cases where that feature can be implemented in the code in a plurality of distinct ways.

It is certainly true that the definition of words can change over time (so much so that this is a topic of study known as *semantic change* in the field of Linguistics), but the works of Fowler, and those cited by Fenton and Pfleeger, and separately those cited by are Kagdi *et al.* are all contemporaneous to one another. It follows then that there is no consensus of the precise nature of coupling. This is a problem, because as Wand and Weber have noted, we cannot expect the state of knowledge in a field to advance quickly if the fundamental concepts and terms in that field (like coupling) remain poorly defined [WW90]. Eden and Kazman too, similarly caution us on the dangers of fundamental terms becoming "mere platitudes" in the field of software engineering [EK03].

[3] This is not a criticism of Parnas' work—likely, and appropriately, the KWIC program was deliberately selected as a simple example for pedagogical reasons.

Given the discussion above on the controversies that exist in the area of software structure, how might we address them? It is my position—and that taken in this work—that measurement of *just* internal attributes of real software systems is an excellent starting point for addressing these controversies.

1.2.4 The Case for Measuring Just Structural Attributes

It is widely accepted that measuring a thing forces us to formalize our otherwise only intuitive notions of that thing, and that measurement is a key part of science [FP96]. By actually measuring the various forms of coupling that exist and are meaningful, we can see that it is perhaps best described as a *general concept* with many different manifestations rather than an internal or external attribute of software. In order to perform the kind of empirical studies of coupling many in the empirical software engineering community want to see, besides measuring the external quality attribute, we must also be able to measure the thing present in source code we are attempting to correlate with that—so this approach is not really inconsistent with those efforts. Further, a careful reading of study of Parnas' seminal work on information hiding reveals he too was using measurement to make his argument [Par72]. In particular, for two different designs of the KWIC program, he counts (i.e., measures) the number of modules that would require changing for each of the requirements under consideration—a smaller number of modules changed being superior to a larger number changed. It follows then that my approach to measuring internal attributes is not really inconsistent with the views of Parnas either.

One of the insights of the research contained herein, that might also go some way toward addressing the controversies described above is the introduction of the notion of *activities* [MT07e]. An activity may help us to "bridge the gap" between a thing that exists in source code, and an external quality attribute. For instance, the activity of recompilation after a change in Java (and in C++) involves recompiling all the source code that transitively depends on the one that was modified. It therefore makes no sense to try and *empirically* link transitive compilation dependencies to number of files requiring to be recompiled. As another example, if our approach to (i.e., activity for) reuse is to copy a source file without modifying editing it, to ensure it will compile in the new system we also need to copy all the other source files on which it transitively depends. Again, there is no sense in empirically establishing a link between compilation dependencies and the files that need to be copied, because it exists in our definition of the activity. There are many more examples in this vein described in the works contained herein, and I term recompilation and this specific form of reuse (and the others) *activities*.

When there is a sound theoretical link between a thing in source code, and what I have termed an *activity*, it does not make sense to empirically validate that link. What does make sense though, is to empirically determine the relevance of that activity to the external software quality attributes. So, for the recompilation activity, to what extent are software developers waiting on recompilation when maintaining a system? Is the "copy source code without modification from one system to another" activity the appropriate approach to reuse? Relating to Parnas' seminal work too [Par72], is minimizing the number of modules requiring change empirically linked to reduced effort for making that change? (In large part, it is the sound theoretical link to these so-called *activities* that ultimately caused me to focus my efforts in this research on compilation dependencies among source files, and not on other perhaps more sophisticated forms of coupling where there is no sound theoretical link to any such activity).

1.2.5 Summary

To summarize: despite software structure being an active research area in software engineering for over 50 years, many controversies persist relating to the precise nature of fundamental terms in it such as *coupling*, and the methods by which we should seek to show connection between structural attributes of source code and external software quality attributes. In a manner that is not inconsistent with the otherwise opposing views on the methods by which such connections should be shown, I propose measurement of *just* internal attributes in real software systems, with a particular focus on those that have sound theoretical connections to *activities*, which themselves may subsequently be shown to be connected to external software quality attributes.

1.3 Overview of the Publications (Chapters) in this Research

This is a *PhD by Prior Publication*, so the body of this dissertation comprises the publications that have resulted from the research. I have elected to include the publications verbatim (except for formatting changes which were required to meet thesis submission requirements), as they were accepted for publication by the referees at each of their various venues. In this section I provide a short overview of each publication, and explain how together they form a coherent body of work. Where there are coauthors on a publication my specific contribution to that publication is as described in the mandatory coauthor declaration forms appearing in this thesis immediately prior to each publication (chapter).

1.3.1 First Publication [MT06]

In Chapter 2 *Identifying Refactoring Opportunities by Identifying Dependency Cycles* [MT06] I describe a tool *Jepends* that I built that infers compilation dependencies among Java source files in Java projects without necessarily requiring them to be in a state such that they can compile. This tool is important for large scale, multi-project studies because as I discovered when gathering projects for what became the Qualitas Corpus (see Chapter 11) the steps to build a Java project vary greatly from project to project. In addition to inferring the dependencies among the source files of a project using an algorithm derived from Lagorio's [Lag04], the tool collects various metrics about those source files and the dependencies among them.

I further describe how I use to tool to collect some simple metrics from a handful of open source Java applications, reporting specifically on Tomcat and Azureus. I show that the transitive closure of the inward and outward compilation dependencies across source files in Azureus are quite different from those in Tomcat, whereas the distribution of other metrics such as direct dependencies look quite similar. I show that the difference in the distributions of the transitive closure of compilation dependencies in Azureus relative to that in Tomcat, is due to the existence of cycles in the dependency graphs of Azureus.

By counting *simple cycles*—or actually a sampling thereof—I demonstrate in Azureus that it is possible to identify source files using these metrics as candidates for specific forms of refactoring, if the goal is to break dependency cycles among a program's source files. I review some of the literature that advises against dependency cycles. To my knowledge this was the first, albeit very small-scale, empirical investigation of (compilation) dependency cycles among source files in Java.

This paper leads into the next with the speculation that the distribution of some metrics might be invariant from project to project, while the distribution of others might vary from project-to-project. Which are those that might be invariant, and why? The works of Wheeldon et al. [WC03] and Marchesi et al. [MPST04] on the existence of power laws in certain metric distributions in Java and Smalltalk respectively are identified in the related work section of this paper and indeed this is the focus of the next paper.

The claimed contributions of this paper are the tool *Jepends* itself for inferring compilation dependencies among a project's source files, that the tool can also be used to detect and count cycles and direct and transitive compilation dependencies among Java source files (ignoring redundant `import` statements), and that those metrics can be used to identify candidate Java classes for extract-interface refactorings (which is demonstrated on the code base of Azureus), This paper was awarded best paper at the conference in which it appeared.

If one were to retrospectively ascribe a research question (RQ) that this paper sought to answer that question might reasonably be stated as follows:
RQ1: Can compilation dependencies among a Java project's source files (only) be quickly and accurately computed without external libraries, build scripts and so on, and if so what observations can one make about those compilation dependencies in real-software?

1.3.2 Second Publication [BFN+06]

In Chapter 3 *Understanding the Shape of Java Software* [BFN+06] my coauthors and I seek to understand which metrics might be invariant from Java one project to another in terms of their distributions, and which might be project-specific. It is the first significant work published that made use of a sizeable curated corpus of Java software that came to be known as the *Qualitas Corpus*. Among other contributions to the paper, I conceived and developed this corpus and my coauthors acknowledge so in the paper of Chapter 11.

In the paper we find that just because a distribution of a metric appears to be fat- or long- tailed, it doesn't necessarily mean that it obeys a power law (as had previously been found by Wheeldon and Counsel [WC03]); other types of probability distributions might statistically fit just as well or even better. We explain in detail why our use of a sizeable, curated corpus of Java software might yield better results than non-curated corpora where others have found only power laws, and posit from our findings that the distributions of some metrics might, in real software, be unavoidable regardless of how they are designed.

In terms of the contribution of this work—it has subsequently been cited by over 170 papers and there is insufficient space to describe them all here—two particularly important works stand out. One is by Hatton where theories for the causes of power laws in software are proposed, and where he conducts an empirical study very reminiscent of ours except across a slew of programming languages [Hat09]. The other is a PhD thesis by Taube-Schock where he seemingly concludes by analyzing the same corpus that high coupling is unavoidable "all systems in the corpus are scale-free and that that property results in high coupling", and that high coupling may not necessarily be a bad thing despite instructional literature on software design to the contrary [TS12].

If one were to retrospectively ascribe a research question (RQ) that this paper sought to answer that question might reasonably be stated as follows:
RQ2: In real Java software, which structural metrics seemingly have distributions that are invariant from project-to-project, and among those with invariant distributions are they *really* powerlaws?

1.3.3 Third Publication [Mel06]

In Chapter 4 *On the Usage and Usefulness of OO Design Principles* [Mel06] I espouse the benefits of studies of *just* internal attributes of real software systems in a corpus of such. Overall I argue that although such studies by themselves cannot, by definition, draw empirical connections to external quality attributes, they can help us in very meaningful ways. In particular, by alerting practitioners which structural phenomena actually manifest in the systems they work on; to help focus costly research effort on structural phenomena that are thought to be detrimental but that are widespread in real software, to generally be more scientific in our methods (measurement of a thing forces us to formalize our otherwise intuitive understanding of it [FP96]).

The PhD thesis of Oyetoyan has proven my views in this paper valid [Oye15]. He heavily cites my works described in this document as justification for going to the trouble of performing empirical studies of cycles he performed in an attempt to show connections between them and external attributes such as maintainability and change-proneness. Others too (including Oyetoyan) have used these studies to justify the cost and time of building tools to help break cycles [CALN16].

Also in this paper I mention the benefits of making the curated Java corpus I had developed widely available. Again, this came to fruition and is discussed in Chapter 11, with the corpus now in wide use, and a thing of study in its own right (see e.g., the work of Terra et al. [TMVB13] where the objective is to make the corpus' source code automatically compile, and that of Dietrich et al [DSST17] where the objective is to make software in the corpus automatically execute).

This was a Doctoral Symposium paper and allowed me to solicit feedback on the direction of research. The paper, although short, describes the goals of this entire body of research, and the approach taken to it (and why I chose to focus mainly on transitive compilation dependences as opposed to other perhaps more exotic forms of coupling), and identifies some gaps in the pre-existing body of research that justifies this body of research's existence. I elected to include this paper in this body of this thesis for these reasons, and so that in Chapter 12 (where I conclude this work) I could reflect upon the extent to which the goals were achieved and extent to which the approach was successful.

It is perhaps strange to ascribe a research question to this paper since it was a Doctoral Symposium paper and not a research paper per se, but for the purposes of consistency the research (meta-) question this paper sought to answer might reasonably be stated as follows:

RQ3: What is the intended approach, goals, and outcomes of this PhD research?

1.3.4 Fourth Publication [MT07a]

In Chapter 5 *The CRSS Metric for Package Design Quality* [MT07a] I describe how transitive dependencies among classes may have a detrimental effect on dependencies among the packages that comprise a software system. The design advice on avoid cycles at the *package* level is more prevalent than that at the *class* level. In essence, I observe that if the classes comprising an application have large transitive dependencies, then the packages comprising that application cannot be both of reasonable size and free of cycles.

Using my class reachability set size (CRSS) metric, I show that transitive compilation dependencies in several real-world Java applications in the Corpus preclude them from having a good package structure (without even having to analyze the package structure of those applications), no matter how those classes are rearranged among packages. I identify some specific refactoring techniques (specifically, dependency injection and a registry of singletons) that might be used to break these large transitive dependencies. Specific examples of these refactorings on the codebases of Eclipse and Azureus are walked through, and the effects on the transitive dependencies as a result of these specific refactorings are measured.

In reviewing some of the citations of this paper, it seems fair to say that it ignited interest in refactoring to break dependencies among packages—something that had previously received very little attention (see e.g., Laval's PhD thesis which subsequently cites my work [Lav11]). The CRSS metric proposed by me in this paper has also subsequently been studied by Oyetoyan as part of his PhD research [Oye15]: initially in an attempt to correlate it with an external quality attribute [OCC14], and later to extend the work in this paper notably reusing and extending the Jepends tool itself, and improving upon the refactoring techniques identified by me [OCTN15].

In this paper, relating to those two specific techniques I identify for breaking transitive dependencies—and crucially both of which ultimately require an interface to be extracted from an implementation—I coin the phrase “the problem of instantiation”. This alludes to the fact that the implementation of an interface has to be instantiated somewhere, and that if it is also instantiated in the class using the interface then the transitive dependencies induced by the implementation (that are likely “larger” than those induced by just the extracted interface) are not actually broken. This key insight is essentially what led to the papers of Chapters 6, 9 and 10, as described below.

With respect to the paper of Chapter 6 [MT07b], the findings in this CRSS paper led me to question to what extent do cycles contribute to large CRSS values, and what type of cycles should we be measuring? Further, to what extent do cycles appear in the public (cf. private) parts of classes? That of course affects the extent to which extract-interface based refactorings will be successful at reducing transitive dependencies.

With respect to the paper of Chapter 9 [MT07d], the findings of the CRSS paper and this problem of instantiation led me to the question: how else, apart from instantiating a class, might one cause a transitive dependency on things in its implementation? The answer to that is through the use of non-private static members (i.e., methods and fields). The question addressed in the paper of Chapter 9 is to what extent to statics “cause” cycles, and therefore potentially large transitive dependencies.

With respect to the paper of Chapter 10 [YTM08], I sought to answer the question, if dependency injection is so widely-used (and the trade literature at the time seemed to indicate it was), why do so many real-world programs have such large transitive dependencies among their source files? My coauthor and I sought to answer that by investigating both the extent to which dependency injection is used, and the extent to which referencing a “default implementation” in the client class may have been the cause of transitive dependencies on things appearing in the implementation being present in that client class.

If one were to retrospectively ascribe a research question (RQ) that this paper sought to answer that question might reasonably be stated as follows:

RQ4: In a corpus of real Java software what do the distribution of transitive dependencies among source files look like, and what are the implications in terms of software design quality of these distributions?

1.3.5 Fifth Publication [MT07b]

In Chapter 6 *An Empirical Study of Cycles Among Classes In Java* [MT07b] I perform the first truly in-depth study of compilation dependencies among classes in real world Java applications, using a very mature version of my curated Java corpus, which by this time also included some commercial software. Compilation dependencies are categorized by their pathology—some being worse (and/or harder to break) than others. In combination with this, a minimum edge feedback set approach from graph theory is used to try and quantify the strength of connection in a strongly connected component of Java source files. The effects cycles have on certain *activities* (e.g., integration test ordering, reuse at the level of source code, recompilation, and so on) are described in detail.

If just a single paper could be used to prove my thesis—that carefully conducted empirical studies of *just* internal attributes can advance knowledge in our discipline in a meaningful way—this would be that paper. The novelty of this paper is its very thorough (i.e. careful) treatment of the internal attribute of “cycles” in the compilation dependency relation among source files. The treatment is thorough, in part, because cycles are studied in a corpus of 78 real-world Java projects, that were deliberately chosen to vary along a number of dimensions to achieve some representativeness of Java projects in general. Those dimensions were: open or closed-source, the domain, their origin, their size, and so on. Of the 78 projects, 22 had multiple versions which allowed a “longitudinal” study of cycles, from release-to-release in those projects.

The treatment is also thorough because the origins of the design principle “avoid cycles” are traced back through the literature, and in doing so the specific arguments for why cycles are “bad” are identified. Those arguments are espoused in the paper and by doing so meaningful measurements of cycles that relate to the activities in these arguments could be derived. For instance, rather than counting *simple cycles*, those arguments led me to realize *strongly connected components* were more appropriate to measure. Further, the size of a strongly connected component alone does not indicate the number of dependencies that need to be broken to break that cycle—that led me to the *edge feedback set* metric. Further still, within the edge feedback set metric, inheritance relationships are harder to break than other types of relationships so in calculating the size of that metric with a view to estimating effort to break cycles, that form of dependency was excluded from the edge feedback set.

Finally, cycles among different dependency relations are computed with a view to distinguishing cycles that exist due to inherent relationships among the entities being modeled in the domain (e.g., the mutual relationship between a *node* and an *edge* in a *directed graph*), from unnecessary or "bad" cycles. The way this is achieved is by distinguishing cycles that appear in the public interfaces of a class from those that appear elsewhere (e.g., in the private implementation details of the class).

What is found in the paper is that of the projects in the corpus comprising enough classes to support such a cycle, about 45% have a cycle involving at least 100 classes and around 10% have a cycle involving at least 1,000 classes. What is also found in the longitudinal-style study is that strongly connected components tend to grow in size in subsequent releases of the same project. What is further found is that cycles appearing in the interfaces of classes tend to be much smaller than those appearing in their implementations, which is implies extract-interface style refactorings may be quite successful at breaking large cycles and reducing transitive dependencies.

The impact of this paper is highlighted by the 80+ citations it has received, and the subsequent work in the *exact same area* it seems to have inspired: at least two PhD theses on Cyclic Dependencies in Java [Sha13] [Oye15] and at least one Masters thesis on the same [AM13].

Some questions this paper naturally raises that lead into the subsequent publications are: How, instead of having to break cycles, might software engineers avoid creating them in the first place? And, what is it that causes a software engineer (perhaps only inadvertently) to create a cycle in the first place? These questions are addressed in the papers of Chapters 7 and 9.

If one were to retrospectively ascribe a research question (RQ) that this paper sought to answer that question might reasonably be stated as follows:

RQ5: In a corpus of real Java software to what extent do cyclic dependencies exist and evolve over time, and in terms of software design quality what are reasonable metrics for measuring this?

1.3.6 Sixth Publication [MT07c]

In Chapter 7 *JooJ: Real-time Support for Avoiding Cyclic Dependencies* [MT07c] I argue that the best way to break cycles might be to avoid creating them in the first place. I suggest that the reason that these cycles might come to existence in the first place is because a software engineer working on a system is not cognizant of the overall structure of the system each time s/he makes modifications to individual source files, and that this might inadvertently lead to their introduction.

I describe a novel plugin I built for the Java IDE *Eclipse* that analyzes an entire application for cycles as code is being written, statement-by-statement, so as to provide immediate feedback to a software engineer when a cycle is created. I demonstrate, using the corpus, that the plugin can provide real-time feedback in Eclipse when run on a modest desktop computer at the time, even when running on an application comprising 11,000 Java source files.

The need for the tool described in this paper is largely justified by the widespread existence of large cycles in Java software that I found in [MT07b]. The arguments for why such a tool improves upon the current state of the art (which were batch-style tools) are, as described in the paper, that: (1) code is more resistive to change after it has been written, (2) changing other people's code is hard and (3) as per the poka-yoke approach to manufacturing pioneered by car manufacturer Toyota it is best to fix or prevent mistakes as close as possible to the task that creates them (in this case that "task" is unwittingly writing a new line of code that induces a cyclic dependency).

Others too have cited this work agreeing there is need for better tool support to break cycles. Examples include the PhD thesis of Laval on tool support for breaking cycles among packages [Lav11], and the recent work of Caracciolo et al. where it is noted "Unfortunately, detecting cycles is only half of the work. Once detected, cycles need to be removed and this typically results in a complex process that is only partially supported by current tools. We propose a tool that offers an intelligent guidance mechanism to support developers in removing package cycles. Our tool, Marea, simulates different refactoring strategies and suggests the most cost-effective sequence of refactoring operations that will break the cycle" [CALN16].

One of the main themes of the JooJ paper is proof that it can operate on a large project in real-time. What is very interesting is that nine years later, despite advances in computing consistent with Moore's Law, the ability of tools like this to operate in real-time remains a concern. Again, quoting from the work of Caracciolo et al "Our approach [to identifying the specific refactoring based on a custom profit function] has been validated on multiple projects and *executes in linear time*."[CALN16]

If one were to retrospectively ascribe a research question (RQ) that this paper sought to answer that question might reasonably be stated as follows:
RQ6: Is it computationally feasible to perform whole-program analysis to identify cyclic dependencies in Java code, as that code is being written, in a manner that is tightly integrated with existing Integrated Development Environment (IDE) features?

1.3.7 Seventh Publication [MT07e]

In Chapter 8 *Towards Assessing Modularity* [MT07e] I describe the problems with the term *modularity* much like how in this introductory chapter I have described the problems with the term *coupling*. The page limit was quite severely constrained for this workshop, so the paper is very short, but it nevertheless makes several important points that relate back to arguments made in this introductory chapter about needing agreement on the definitions of things. At the time I wrote this I was not brave enough to say that modularity is neither an internal attribute nor an external one, but rather just a general concept (like coupling). All the arguments I have made about coupling in this chapter began with my thinking in this modularity paper, and indeed this paper introduces the term "activity".

In short, I argue that modularity is not well-defined in the field of software engineering. My position is that modularity is the extent to which a thing comprises *independent parts.* Merely increasing the number of parts (e.g., by splitting one of a program's class' into two) does not necessarily increase the extent to which the thing is modular, because those parts also have to be independent of one another.

This leads to my next point that whether two parts can be considered independent or not depends on the specific activity one is undertaking or perspective from which one is assessing modularity. Two such parts may be independent from the perspective of say unit testing, but not from say the perspective of verbatim reuse of source files. Put another way, by simply saying "modularity is improved" by this new programming language feature, or design technique is neither helpful nor meaningful (yet it continues to occur even in publications in this calendar year, 2016). The specific perspectives from which modularity is improved need to be carefully articulated or we risk the term becoming a mere platitude (the dangers of such platitudes are as previously discussed and cited in this introductory chapter [WW90][EK03]).

I also point out that modularity, though often talked about only as a "good" thing (e.g., "our goal is to always to increase modularity"), may not always be as such. For instance, and as pointed out in the paper, it may be harder to change a system that comprises too many independent parts, because finding the right part to change may prove more difficult than in a system with fewer parts. This view is entirely consistent with that of Baldwin and Clark who argue in their highly cited book that designing for modularity is like a buying a kind of financial instrument known as an *option* [BC00]. There is a cost to buy that derivative contract (option), but at some point in the future it may (or may not) provide a savings greater than its cost (in option vernacular, its "premium"). In software engineering terms, as Parnas has intimated, designing our software now for possible future changes costs money now but if those changes do actually happen we will save money and time in the future [Par94].

It is perhaps strange to ascribe a research question to this paper since it was more of a position paper and not a research paper per se, but for the purposes of consistency the research question this paper sought to answer might reasonably be stated as follows:
RQ7: Does it make sense to reason about modularity without a clear definition of it, and even with such does it make sense to do so in isolation without reference to a specific *activity*?

1.3.8 Eighth Publication [MT07d]

In Chapter 9 *Static members and cycles in Java software* [MT07d] I perform yet another empirical study on the curated corpus to collect empirical evidence to support my theory that cycles may be caused by the "overuse" of static members (i.e., non-private static methods and non-private static fields) in Java. In this study I am careful to control for so-called *confounding factors* such as class size by stratifying the dataset along two dimensions: presence or absence of static members and size of class (big or small).

What I find in this study is that both at the application- and corpus-level the results generally seem to support the contention that classes that are accessed statically are more likely to be involved in cycles than those that are not. For the four hypotheses tested in the study (all using the χ^2 test) I obtained only three statistically significant negative results. For the hypothesis pertaining to edges due to access of static members appearing in cycles, only six applications of the 81 examined had a negative result.

This paper is the only one that I am aware of that attempts to correlate these two *internal attributes* with one another (statics and cycles) to gather evidence support a theory that one such internal attribute *causes* another. It would seem to provide evidence that the anecdotal design advice about generally avoiding static members because they are the "globals" in the object-oriented paradigm is sound.

If one were to retrospectively ascribe a research question (RQ) that this paper sought to answer that question might reasonably be stated as follows:
RQ8: Is the use of non-private static members in Java projects a probable cause of dependency cycles among classes in those projects?

1.3.9 Ninth Publication [YTM08]

In Chapter 10 *An empirical study into use of dependency injection in Java* [YTM08], my coauthors and I investigate another structural phenomena that may cause both large transitive dependences and/or cycles—the use of appearance of the default implementations of an interface in the nullary constructor of classes that had otherwise seemingly been designed to implement *dependency injection*. Dependency injection, as the phrase is used in this paper, refers to the passing of an implementation of one class into another through a formal parameter declared as a supertype of the former in the latter's constructor(s). The paper traces the origins of dependency injection and reviews the arguments for the effects it has on quality attributes.

The paper uses what is effectively a program slicing tool built on top of other tools (specifically Jimple, Soot and Indus) by my coauthor Yang to detect instances of dependency injection in a version of the Qualitas Corpus. It does so by way of constructing *use-def chains* which trace through a program the origin of an assignment to a variable. What is found by using this tool to analyze the corpus is that dependency injection is not as widely used as might otherwise have been inferred by reading the trade literature on it (at least in terms of the projects in the corpus studied). In many applications it is not used at all. In some ways, its lack of use makes the second part of the study—the extent to which it is used with a *default implementation*—moot as far as this default implementation causing large transitive dependencies.

One conclusion from this work that relates specifically to breaking cycles and reducing transitive dependencies is that refactoring an existing code base to make wider-use of dependency injection may be an effective technique for doing so. This is because dependency injection was not found to be widely-used, and because my other study found cycles aren't as big in just the public interfaces of classes as in their implementations [MT07b].

If one were to retrospectively ascribe a research question (RQ) that this paper sought to answer that question might reasonably be stated as follows:
RQ9: Is dependency injection widely-used in real Java projects, and if so is it used in a manner that would reduce transitive compilation dependencies?

1.3.10 Tenth Publication [TAD+10]

In the final paper of Chapter 11 *The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies* [TAD+10] my coauthors and I describe the corpus I had, quoting from the paper, originally "conceived and developed" during my time as an PhD candidate at the University of Auckland. The paper describes the design of the corpus so as to make its content readily accessible to other researchers for replication studies, to lower their barriers to entry for performing their own empirical studies and so on. The history, current organization of the corpus and the specific reasoning that led to these things is all described.

Besides what is explicitly described in this paper, the evolution of the corpus is implicit in the publications described above that use it. Almost all of the design decisions relating to the corpus were made exclusively by me and I became strongly influenced by the work of Hunston in Corpus Linguistics in making those decisions [Hun02]. In short, I chose the projects for their corpus to vary along many dimensions (size, domain, origin, open or closed-source) so some degree of representativeness could be claimed. I also added multiple versions of a number of the projects in the corpus for the purposes of performing longitudinal studies of how the structure of those projects had evolved over time.

I made the decision to distinguish between classes appearing in a project's actual source code versus being depended on as third party libraries to avoid "double counting" of classes between applications. The way I achieved that was to manually inspect each project and record with it the list of Java package prefixes that contained its source (vs. those packages that contained external code). Initially I had only downloaded each project's source code but later on it became clear to me that I should download both source code and binaries, because (1) some forms of analysis were easier to perform on Java byte code than on source and (2) because I wanted to verify the correctness of my Jepends source code-analyzing tool by comparing its output to another such tool I built to read such dependencies out of the byte code using Apache BCEL and (3) because it was too time consuming to figure out the build process specific to each project which oftentimes involved editing an Ant script, downloading the right versions of external jars and so on.

Some other interesting design decisions are also worth mentioning. One was my decision to include more than one project from the same domain—for instance Netbeans and Eclipse. Since both those applications are Integrated Development Environments, what that sometimes allowed was comparisons to be made about whether structural phenomena detected might be inherent to the domain, or not.

Judging from its 170+ citations, the contribution of this paper was largely as intended—it is mostly cited by other researchers using the corpus to do their own empirical studies of structural attributes. What is very interesting though, is that there is also at least one work where the contribution of it is to corpus itself—modifying the artifacts in the corpus so they could be successfully compiled from their source code [TMVB13].This paper was awarded best paper at the conference in which it appeared.

If one were to retrospectively ascribe a research question (RQ) that this paper sought to answer that question might reasonably be stated as follows:
RQ10: What were the specific considerations, issues and limitations encountered when designing the Qualitas Corpus and what is the case for other researchers making future use of it in their empirical studies?

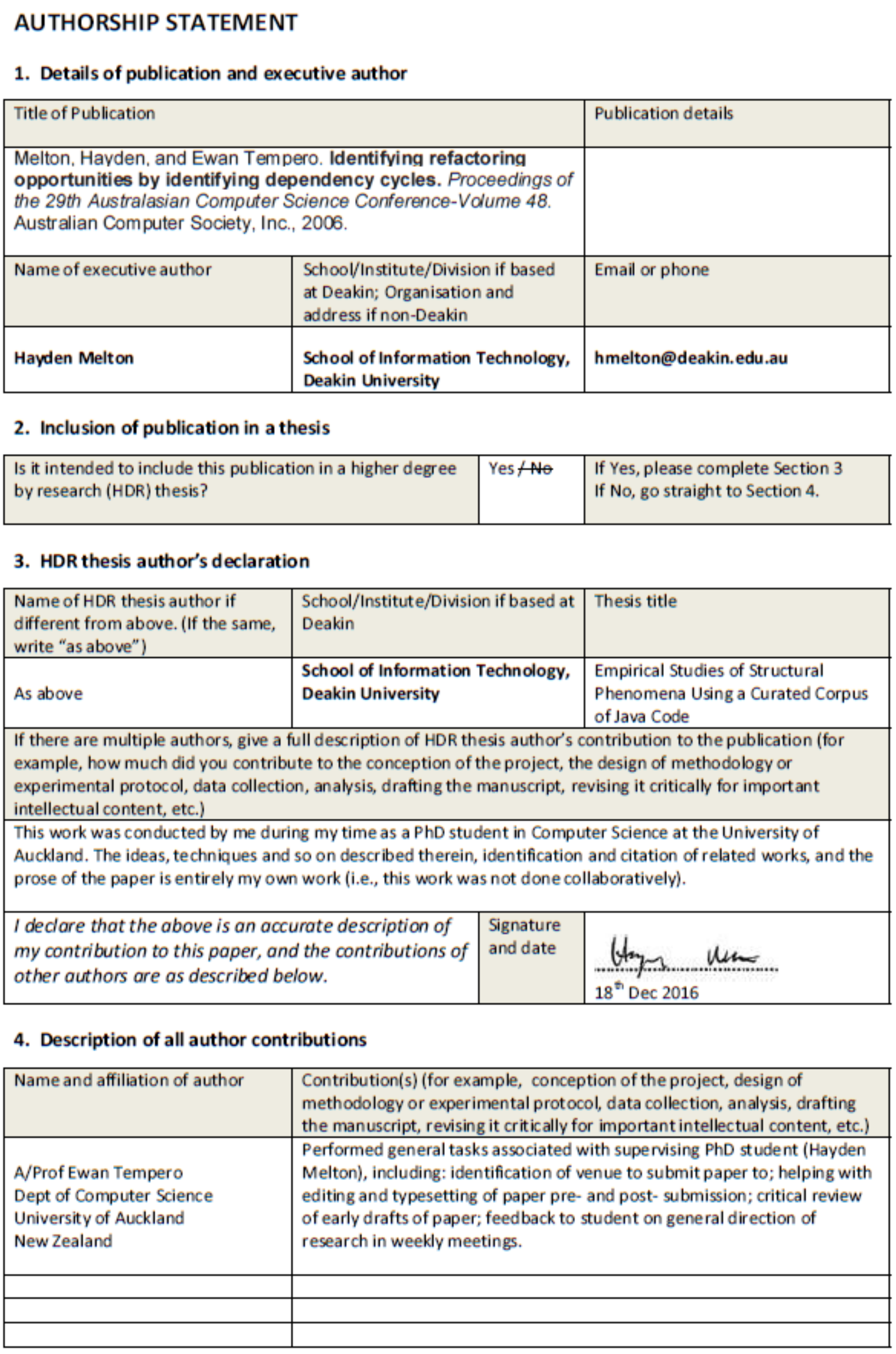
1.3.11 On the Connections between the Publications

To restate the connections of the papers to one another: the initial "cycles" paper [MT06] led to the insight that some metric distributions were seemingly the same between Java projects and others were different. These insights led to three papers: the "shape" paper [BFN+06] which examines distributions that are similar between projects and the CRSS paper [MT07a] and the in-depth "cycles" paper [MT07b] which both examine metric distributions that are different among projects. Also as a result of the initial “cycles” paper the benefits of performing studies of just internal attributes became clear to me and those were espoused in my doctoral symposium paper [Mel06]. Following on from the in-depth “cycles” paper [MT07b], I wondered about the causes of cycles which led to the tool paper for avoiding cycles [MT07c] and the “statics” paper investigating their relationship with non-private static members [MT07d]. In describing all the arguments for why cycles were bad in the in depth “cycles” paper it occurred to me that there was an intermediate step involving an *activity* between cycles and external quality attributes, and that is described in this introductory chapter (in the context of coupling) and also in the “modularity” paper [MT07e]. Further, in wondering on the cause of and large transitive dependencies [MT07a] cycles I wondered if default implementations in dependency injection were responsible for them and for large transitive dependencies, and that led to the dependency injection paper [YTM08]. Finally, after gradually evolving the corpus, and repeatedly using it in many of my studies, it had become a thing in its own right worthy of discussion, and that led to the “corpus” paper [TAD+10].

1.4 Organization of this Dissertation

In this introductory chapter I have motivated this research and given an overview of it. As noted earlier, the body (i.e., Chapters 2-11) of this dissertation comprise the published papers that have resulted from this research. The papers are presented in chronological order, by date of publication, verbatim as they were accepted for publication by the referees (except for changes to formatting and bibliographic references, which have been consolidated to all use the same style). In the final chapter of this dissertation—Chapter 12 *Conclusions and Future Work*—I review the contributions of this work, its significance and relevance, I evaluate its outcomes in terms of my stated goals for it, I self-identify some possible criticisms of it, and I discuss future directions of this work, which despite the many works it has led to, there are still many.

Coauthor Declaration for Chapter 2 [MT06]



AUTHORSHIP STATEMENT

1. Details of publication and executive author

Title of Publication		Publication details
Melton, Hayden, and Ewan Tempero. **Identifying refactoring opportunities by identifying dependency cycles.** *Proceedings of the 29th Australasian Computer Science Conference-Volume 48.* Australian Computer Society, Inc., 2006.		
Name of executive author	School/Institute/Division if based at Deakin; Organisation and address if non-Deakin	Email or phone
Hayden Melton	**School of Information Technology, Deakin University**	**hmelton@deakin.edu.au**

2. Inclusion of publication in a thesis

Is it intended to include this publication in a higher degree by research (HDR) thesis?	Yes / ~~No~~	If Yes, please complete Section 3 If No, go straight to Section 4.

3. HDR thesis author's declaration

Name of HDR thesis author if different from above. (If the same, write "as above")	School/Institute/Division if based at Deakin	Thesis title
As above	**School of Information Technology, Deakin University**	Empirical Studies of Structural Phenomena Using a Curated Corpus of Java Code
If there are multiple authors, give a full description of HDR thesis author's contribution to the publication (for example, how much did you contribute to the conception of the project, the design of methodology or experimental protocol, data collection, analysis, drafting the manuscript, revising it critically for important intellectual content, etc.)		
This work was conducted by me during my time as a PhD student in Computer Science at the University of Auckland. The ideas, techniques and so on described therein, identification and citation of related works, and the prose of the paper is entirely my own work (i.e., this work was not done collaboratively).		
I declare that the above is an accurate description of my contribution to this paper, and the contributions of other authors are as described below.	Signature and date	[illegible] 18th Dec 2016

4. Description of all author contributions

Name and affiliation of author	Contribution(s) (for example, conception of the project, design of methodology or experimental protocol, data collection, analysis, drafting the manuscript, revising it critically for important intellectual content, etc.)
A/Prof Ewan Tempero Dept of Computer Science University of Auckland New Zealand	Performed general tasks associated with supervising PhD student (Hayden Melton), including: identification of venue to submit paper to; helping with editing and typesetting of paper pre- and post- submission; critical review of early drafts of paper; feedback to student on general direction of research in weekly meetings.

5. Author Declarations

I agree to be named as one of the authors of this work, and confirm:

i. that I have met the authorship criteria set out in the Deakin University Research Conduct Policy,

ii. that there are no other authors according to these criteria,

iii. that the description in Section 4 of my contribution(s) to this publication is accurate,

iv. that the data on which these findings are based are stored as set out in Section 7 below.

If this work is to form part of an HDR thesis as described in Sections 2 and 3, I further

v. consent to the incorporation of the publication into the candidate's HDR thesis submitted to Deakin University and, if the higher degree is awarded, the subsequent publication of the thesis by the university (subject to relevant Copyright provisions).

Name of author	Signature*	Date
Ewan Tempero	[illegible]	16/12/2016

6. Other contributor declarations

I agree to be named as a non-author contributor to this work.

Name and affiliation of contributor	Contribution	Signature* and date

* If an author or contributor is unavailable or otherwise unable to sign the statement of authorship, the Head of Academic Unit may sign on their behalf, noting the reason for their unavailability, provided there is no evidence to suggest that the person would object to being named as author

7. Data storage

The original data for this project are stored in the following locations. (The locations must be within an appropriate institutional setting. If the executive author is a Deakin staff member and data are stored outside Deakin University, permission for this must be given by the Head of Academic Unit within which the executive author is based.)

Data format	Storage Location	Date lodged	Name of custodian if other than the executive author
Qualitas Corpus of Java code, both binaries and source code with metadata (see *http://qualitascorpus.com/*)	University of Auckland, New Zealand	Starting 2006, onward	Ewan Tempero

This form must be retained by the executive author, within the school or institute in which they are based.

If the publication is to be included as part of an HDR thesis, a copy of this form must be included in the thesis with the publication.

Chapter 2 Identifying Refactoring Opportunities by Identifying Dependency Cycles

The purpose of refactoring is to improve the quality of a software system by changing its internal design so that it is easier to understand or modify, or less prone to errors and so on. One challenge in performing a refactoring is quickly determining where to apply it. We present a tool (Jepends) that analyses the source code of a system in order to identify classes as possible refactoring candidates. Our tool identifies dependency cycles among classes because long cycles are detrimental to understanding, testing and reuse. We demonstrate our tool on a widely-downloaded, open-source, medium-sized Java program and show how cycles can be eliminated through a simple refactoring.

2.1 Introduction

Refactoring is defined as "the process of changing a software system in such a way that does not alter the external behaviour of the code yet improves its internal structure" [FBB99]. Refactoring is most appropriate for software systems whose existing (internal) design is hard to understand, hard to modify and prone to errors and so on. By refactoring such a software system we alter its design to make it easier to understand, modify and less prone to errors. As such, refactoring is regarded as an important technique for improving software quality during a system's maintenance phase.

There are several challenges in performing a refactoring. One is to identify characteristics of a design that make it hard to understand, modify or test etc. Fowler produces a list of these characteristics which he refers to as 'bad smells in code' or simply smells. Examples of smells include large classes, long parameter lists, feature envy and data classes. Many of these smells have a large degree of subjectivity in their interpretation. For instance, how large is too large for a class? How do we justify (in the case of the feature envy smell) if one method is 'more interested' in another class than in that which it is defined? This leads us to the second challenge in performing a refactoring—identifying where to perform it.

Since many smells have a large degree of subjectivity or variety in their interpretation it is difficult to (reliably) automatically detect where to apply a refactoring. Much refactoring therefore relies upon the slow and tedious task of manually inspecting code. It would be beneficial to be able to reliably automatically detect where refactorings could be applied. To this effect we have identified a particular structure in a system's source code that can be automatically detected, and has a detrimental effect on the system's understandability, testability and reusability. The structure we have identified is long dependency cycles between classes in the system.

Long cycles among classes in Java programs create problems for developers because it is difficult to isolate any class in the cycle. Anyone wanting to understand any class in the cycle effectively has to understand every class in the cycle. This has implications for the cost of maintenance. Anyone wanting to test any class, effectively has to test every class. And anyone wanting to lift a class for reuse in another system, ends up having to lift every class in the cycle. This suggests software with cycles in the compilation dependency graph may be more costly to maintain than those without, which gives motivation for detecting and removing cycles.

Of course detecting and removing cycles would not be so interesting if they did not exist in "real software", or they were "mostly harmless". This leads into the contributions of this paper. One contribution is to show that cycles do exist in real software. We have done this by examining several widely-downloaded, open-source Java applications. In order to determine the prevalence of cycles we have built a tool to detect them — this is another contribution. Since we detect cycles from source code and not from byte code we have had to develop an algorithm for computing name bindings that is of little burden to implement, unlike a fully-fledged Java compiler that by its very nature has to compute name-bindings and requires significant effort to implement—another contribution. The final contribution is showing how dependency cycles detected by our tool can be used as the starting point for refactoring.

The paper is organised as follows. In section 2 we motivate our work by discussing in more detail why cycles can create problems for software developers. We then discuss the literature related to our work in section 3. Section 4 presents the algorithm we use to create the compilation dependency graph. Section 5 discusses Jepends and section 6 shows the results of applying Jepends to a medium sized open source Java application. Section 7 discusses how the results of the analysis can be used to identify opportunities for refactoring, and finally section 8 presents our conclusions.

2.2 Motivation

Cycles in compilation dependency graphs (CDGs) have implications in understanding, testing, and reusing classes in the cycle. But are they really so bad? The simplest cycle is one involving two classes that depend on each other. It is very easy to find examples of such cycles – consider `java.lang.Class` and `java.lang.reflect.Method`, from the Java API for example. It is hard to argue this cycle is 'bad' because of the natural parent-child type relationship between a class and its methods. This relationship is represented at the source code level by `Class` providing a `Method[] getDeclaredMethods()` method and `Method` providing a `Class getDeclaringClass()` method. Breaking this cycle would involve terminating the parent's reference to its children or the children's reference to its parent, both of which are necessary relationships in order to provide usable `Method` and `Class` objects.

It would be tempting to simply declare 2-class cycles "good" and everything else bad, but we suspect "good" 3- class cycles can also be found, and so the question would then be at what size do cycles become "bad"? The 'necessary relationship' argument stated above is an appealing criteria, and may be a correct one, however it has the problem, from our point of view, that it is difficult to detect violations of it through mechanical analysis. While it may be difficult to state categorically that a cycle of a certain size is bad, we would argue that it would be hard to argue that a large cycle, of size 50 for example, is something to be entirely happy with. We feel certain that it would be useful to know that cycles of that size (or larger) exist in our software, since that would provide a candidate for refactoring.

Large cycles in the CDG may indicate another problem. As we discuss in the next section, a number of authors have suggested that cycles of *subsystems* (groups of classes with coherent functionality) are bad. If we have a group of closely related classes (and so coherent functionality) then we would tend to want to understand, test, and reuse them as a unit. As we argued above, cycles within such classes may not be such a problem. However cycles between subsystems suggests that the subsystems are in fact not so coherent, and so again may indicate candidates for refactoring. The larger the cycle in a CDG, the larger the likelihood that the cycles cross subsystem boundaries. For example, if there is cycle of size 50, but all subsystems have fewer than 50 classes, then it must be that there is a cycle between subsystems.

Our goal then is to construct and analyse CDGs, and identify cycles, in particular large cycles.

2.3 Background

There has been a considerable amount of work done in analysing dependencies of different kinds. We mention only the most directly relevant here.

Graphs are a natural representation of computer programs well-suited for program analysis and transformation. Existing work in graph representations of programs is diverse. One dimension of this diversity is the context in which program entities are considered. Program entities may be considered dynamically—from the runtime state of the executing program, or statically—from the source code or an intermediate representation of it. Another dimension of work in graph representation of programs is the purpose for which the graph is used. Purposes include, but are not limited to, identifying violations of design heuristics, change propagation analysis, reverse engineering, reducing compilation time, and runtime performance optimisation. The work most relevant to this paper relates to identifying violations of design heuristics.

The earliest work in the area of runtime performance optimisation using graphs is by Kuck. Kuck introduces a *program dependency graph* in order to determine statements that can be executed in parallel in a (Fortran-like) program [KMC72].

Program dependency graphs have also been used in order to analyse change propagation. The term 'ripple effect' is often used describe how a change can propagate[Bla01]. In essence, a change to the code of one module can have an effect on the data that is passed into other modules. This is of concern during software maintenance because a change to one module that may naively seem isolated could cause a regression fault in another.

In terms of reducing compilation time the graph representation typically comprises source files as vertices and compilation dependencies as directed edges. Yu et al. identify false dependencies as a cause of long compilation times and use a 'partitioning' operation on the graph in order to determine redundant `#include` statements [YDFM03]. Cockerham uses a graph of dependencies amongst Ada source files in order to infer those files that can be compiled in parallel [Coc89]. Assuming multiple processors are available for the compilation, its time is reduced. Lague et al. generate a graph of dependencies between C/C++ source files through processing their *#include* statements [LLLB+98]. This graph is used for reverse engineering in the sense that Lague et al. want to recover the layered architecture of the telecommunications system under study from its implementation (source files).

Several recent studies have profiled the overall characteristics of dependencies among classes in object oriented systems. Wheeldon et al. profiled the distributions of 5 different types of dependencies (e.g. inheritance, aggregation) in several large Java applications [WC03]. Marchesi et al. profiled the distributions of in-degrees and out-degrees for nodes in the class relationship graphs of 4 Smalltalk applications where the relationship took into account potential method invocations and superclasses [MPST04]. The authors of both studies found power laws in these distributions. Furthermore they speculated that these distributions are common across all large object oriented systems and that such distributions may be useful for predicting design complexity as a system grows and measuring the effects of refactorings on software quality. We also consider relationship graphs, however we concentrate on distributions related to the transitive closure of the relationships.

Work with compilation dependencies is usually associated with incremental compilation. Determining what needs to be recompiled when one source file is changed is non-trivial in Java. Lagorio has developed an algorithm for sound cascading recompilation in Java [Lag04] that deals with these issues. Lagorio's algorithm is sound in that its output is guaranteed to have the same effect as recompilation of the whole program. We have adapted Lagorio's algorithm to identify the relationships we are interested in.

Discussion in the literature of the consequences of dependency cycles is limited. Booch makes the observation that a CDG should be a directed acyclic graph as early as 1984, but provides no justification for it [Boo87, p.567]. Szyperski also observes "…can introduce cyclic dependencies and threaten organizational structure" [SGM02].

In terms of dependency cycles between subsystems Riel [Rie96] provides a heuristic that states the model of the application should never be dependent on the user interface of that application. Presumably this heuristic aims to eliminate a dependency cycle between the model and view of the application.

Martin gives the Acyclic Dependency Principle (ADP), namely "the dependency structure between *packages* must be a directed acyclic graph" (our emphasis) where packages are defined similarly to subsystems but with an emphasis on reusability [Mar96b]. As we argued in the previous section, long cycles in the CDG may indicate that the ADP has been broken.

The most comprehensive discussion we found of dependency cycles among subsystems in object oriented software is given by Lakos. Lakos argues for the acyclic property on the basis that cyclic dependencies inhibit understanding, testing and reuse: "once two components are mutually dependent, it is necessary to understand both in order to fully understand either" [Lak96,p.185].

Hautus has developed a tool to detect cycles between packages in Java applications and support removing them [Hau02]. His tool differs from ours in that it assumes classes are correctly organized into subsystems by the use of Java packages. The metrics his tool computes are far less comprehensive than ours and as far as we can tell his tool does not prioritize classes based on some notion of their need for refactoring.

2.4 Algorithm

We have developed an algorithm for inferring compilation dependencies between Java source files in an application. While this may seem at first thought trivial, it is not. As noted by Lagorio the rules for name binding (i.e. binding identifiers in Java source code to their corresponding program entities such as classes, methods, variables) are complicated. This is "because the dot notation is used to name many different kinds of things (types, packages, fields and so on), its semantics is context dependent and tricky" [Lag04].

Suppose we are presented with the dotted name (e.g., `a.b.C` in a Java source file. As stated in section 6.5 of the Java Language Specification the following happens to the name: "First, context causes a name syntactically to fall into one of six categories: PackageName, Type- Name, ExpressionName, MethodName, PackageOrType- Name, or AmbiguousName. Second, a name that is initially classified by its context as an AmbiguousName or as a PackageOrTypeName is then reclassified to be a PackageName, TypeName, or ExpressionName. Third, the resulting category then dictates the final determination of the meaning of the name (or a compilation error if the name has no meaning)". There is a long set of rules for determining the name binding in each of the syntactic classifications. One option would be to implement all these rules in a program to infer dependencies. The other option is to find a heuristic based algorithm that is simpler to implement.

Fortunately there is a relatively simple (heuristic) algorithm for inferring dependencies between Java source files—it is described in Lagorio's work in sound, cascading recompilation in Java[Lag04]. Lagorio's algorithm actually detects a superset of the actual dependencies of a source file. We have adapted Lagorio's algorithm so that it minimises the number of spurious dependencies detected, and ignores some compilation dependencies that are of little consequence to the developer's view of the system's class. The final output of our algorithm is a CDG whose vertices are source files and whose (directed) edges are compilation dependencies. The CDG is built up by processing the names, import statements and package declaration in each source file in order to determine a set of fully qualified type names to which that source file *may* refer. This set is subsequently used to infer dependencies between source files by comparing the type names in it to those declared by other source files in the application.

A simplified version of our algorithm can be expressed as follows: Let the source files in the application be denoted S_1, S_2, S_3, …, S_n. The output of the algorithm is an adjacency list representation of the program's compilation dependency graph of the form $S_i \rightarrow R_i'$ where R_i' is set of source files that S_i directly "refers-to", that is, those source files contain the declarations of types used in S_i.

Firstly consider names in Java that are used to refer to program entities such as methods, types, variables etc. A name can be simple, that is consist of a single identifier, or qualified, that is, consists of a sequence of 2 or more identifiers delimited by "." characters. We will express a name in the form e_1. e_2. e_3. e_4... e_k where e_j represents an identifier.

In order to construct R_i' we first compute R_i by combining, in a particular way, the names in the body of S_i that might refer to types with those appearing in the S_i's package declaration and import statements. R_i is the set of fully qualified class names to which S_i *may* refer. In Java fully qualified type names uniquely identify types within a program.

Let *onDemands*(S_i) be the set of names used in import-on-demand statements in S_i, as well as the package name that S_i belongs to. Import-on-demand statements are imports ending with a '.*'. Let *singleType*(S_i) be the set of names used in single-type-import statements in S_i. Single-type-import statements are imports that do not end with a '.*'. Let body(S_i) be the set of names that could refer to types in the body of S_i. Then:"

$$\begin{aligned}
\mathcal{R}_i^{\text{body}} &= \{e_1 \dots e_j | e_1 \dots e_j \dots e_k \in body(S_i), \\
&\quad 1 \leq j \leq k\} \\
\mathcal{R}_i^{\text{ondemand}} &= \{e_1 \dots e_j \dots e_k | e_j \dots e_k \in \mathcal{R}_i^{\text{body}}, \\
&\quad e_1 \dots e_{j-1} \in onDemands(S_i)\} \\
\mathcal{R}_i^{\text{single}} &= \{e_1 \dots e_j \dots e_k | e_1 \dots e_j \in \\
&\quad singleType(S_i), e_j \dots e_k \in \mathcal{R}_i^{\text{body}} \vee \\
&\quad e_1 \dots e_k \in singleType(S_i)\}
\end{aligned}$$

And so:

$$\mathcal{R}_i = \mathcal{R}_i^{\text{single}} \cup \mathcal{R}_i^{\text{body}} \cup \mathcal{R}_i^{ondemand}.$$

Let T be the set of all types declared in $S_1,\ldots,S_n$, then $R_i' = \mathit{declaringSources}(R_i \cap T)$ where *declaringSources* takes a set of type names and returns a set containing the source files in which the types are declared.

This presentation of the algorithm has been simplified by not taking into account all of the issues due to Java's rules for shadowed names, obscured names, and nested types. Lagorio discusses these issues in full detail[Lag04].

We illustrate the algorithm using the following source file.

```
//file S1
package a.b;
import x.*;
import y.Z;
class MyClass {
 private A a = new A();
 public void doStuff() {
   B b = new C();
   a.exec();
   System.out.println();
 }
}
```

The different sets in the algorithm are:

$$\begin{aligned}
\mathit{body}(S_1) &= \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{System.out}\} \\
\mathcal{R}_1^{\text{body}} &= \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{System.out}, \mathtt{System}\} \\
\mathit{onDemands}(S_1) &= \{\mathtt{a.b}, \mathtt{x}\} \\
\mathcal{R}_1^{\text{ondemand}} &= \{\mathtt{a.b.A}, \mathtt{a.b.B}, \mathtt{a.b.C}, \\
&\quad \mathtt{a.b.System.out}, \\
&\quad \mathtt{a.b.System}, \mathtt{x.A}, \mathtt{x.B}, \mathtt{x.C}, \\
&\quad \mathtt{x.System.out}, \mathtt{x.System}\} \\
\mathit{singleType}(S_1) &= \{\mathtt{y.Z}\} \\
\mathcal{R}_1^{single} &= \{\mathtt{y.Z}\} \\
\mathcal{R} &= \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{System.out}, \mathtt{a.b.A}, \\
&\quad \mathtt{a.b.B}, \mathtt{a.b.C}, \mathtt{a.b.System.out}, \\
&\quad \mathtt{a.b.System}, \mathtt{x.A}, \mathtt{x.B}, \mathtt{x.C}, \\
&\quad \mathtt{x.System.out}, \mathtt{x.System}, \mathtt{y.Z}\}
\end{aligned}$$

It is worth noting that there were names in the body of the source that did not appear in *body*(S_1). Particularly `a` on line 6 does not appear because its context makes it a variable name, thus its name cannot refer to a type. Method declarations/calls such as `.exec()` (9), `doStuff()` (7) and `.println()` (10) do not appear because their context identifies them as methods. The `a` on line 9 does not appear because we can infer from the source file that it cannot refer to a type: it is in the scope of a declared field.

It is also worth noting that many of the names in each source file's *R* will identify types that are not declared in the application's other source files. Lagorio refers to these names as *ghost dependencies*. Since we are not interested in ghost dependencies we cull them from each source file's *R* in order to get a new set *R'*. To know which names to cull we build up a map from type to source file of all the types declared across all the source files in the application. This allows *declaringSources* to be computed.

The key difference between our algorithm and Lagorio's is in the construction of the refers to set, *R*. We minimise the number of entries in *R* by resolving names to variables and types inside a source file where allowed by the Java Language Specification (JLS)[Gos00, Chapter 6]. We remove ghost dependencies from *R*. We do not add single-type-import statements to *R* whose types are not used in the body of the source file (contrary to the example above). While ignoring redundant single-type-imports is not sound in cascading recompilation, it is a minor concern in program analysis where we found it was causing many superfluous dependencies between source files.

2.4.1 Benefits

It is in many ways beneficial to infer dependencies from a system's source files and not its compiled code (i.e. byte code). While inferring a class's dependencies from its byte code is trivial (one can simply look at the fully qualified class names appearing in the class file's constant pool) the process of compiling source files to byte code is seldom straight-forward for a newly downloaded application. It can involve having to track down external libraries, modify build scripts for the local environment and so on. Furthermore if something is preventing the system from compiling (e.g. an unresolved reference or syntax error) then *no* dependencies can be computed. Downloading the application in its compiled form doesn't help much either because it then becomes difficult to determine which classes correspond to sources and which classes have originated from external libraries.

A major benefit of our algorithm is that it is specifies a simpler means of inferring dependencies between source files than the way in which a compiler goes about inferring these dependencies. For instance our algorithm is unconcerned with statement reachability checking, type checking and static context checking, whereas a compiler must perform these steps. As a consequence of the omission of such steps our algorithm should be faster at inferring dependencies between Java source files than a compiler. Even compared to the subsystem of a compiler whose purpose is to compute name bindings our algorithm is superior in that the compiler's subsystem is complicated to implement because it must implement the pages upon pages of rules discussed in section 6.5 of the Java Language Specification. Furthermore, again unlike a compiler, our algorithm does not require references to any external jar files used by an application in order to infer dependencies between sources.

Another benefit our algorithm is that it could be easily adapted to infer compilation dependencies between source files in other Java-like languages such as C#. The simplicity of the algorithm is such that it can be implemented in a few hundred lines of code assuming one starts with an off the shelf parser for the target language.

2.4.2 Limitations

While the algorithm we have described avoids much of the work performed by a compiler, which by its very nature has to infer dependencies, there are situations where it could detect spurious dependencies. Consider the following example in illustration of this.

```
package pack;
import x.*;
class Example {
 A a = new A();
5: }
```

Computing *R* for this source file yields {`pack.A`, `x.A`}. Assume that in the application's source files both types are declared. The JLS states that the types are resolved using the implicit package import in preference to import-on-demand statements (section 6.5.5) so in reality `Example` only depends on `pack.A`. Our algorithm (incorrectly) infers that `Example` depends on both `pack.A` and `x.A`.We expected this type of situation would be very rare. For a medium-sized Java application called Azureus we detected this situation, where two classes had the same simple name, and manually inspected all incidences of it in offending source files' texts. Of the 30 occurrences of conflicting names none caused erroneous references. In each case both classes were actually referenced in the source file's text: one using its fully qualified name and the other using its simple name in conjunction with a single-type- import.

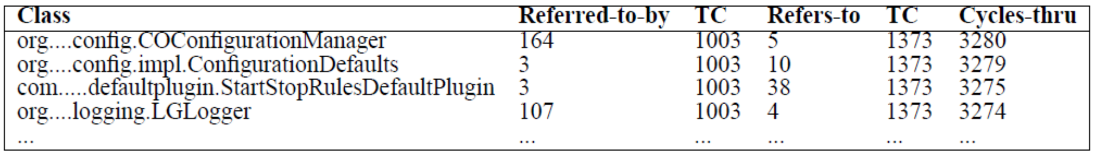
Another way our algorithm could infer an erroneous reference is if a variable name was interpreted as a class name. This is analog to a potential problem stated in the JLS where a variable name could *obscure* a simple type name. Fortunately the convention of naming classes with an initial uppercase letter and naming variables with an initial lowercase letter minimizes this type of conflict (see JLS section 6.8). In all the systems we ran our tool on during its development we casually observed source files had obeyed this coding standard, almost certainly eliminating all erroneous references that could be generated in this way.

One final point to note is that in the general case our algorithm does not infer a direct dependency between a class that uses an inherited field or method, and the class that defines that field/method. Consider a class `A` using a field defined in its superclass's superclass `C`. Our algorithm detects an indirect dependency between `A` and `C` through `A`'s superclass. In this particular example a Java compiler would infer a direct dependency from `A` on `C`, and this would be written to `A`'s binary class file (see JLS 13.4.7). Briand et al's framework for measuring coupling more thoroughly addresses this issue [BDW99].

2.5 Jepends

An implementation of the algorithm described in section 4 has a number of practical benefits in terms of the kinds of analysis we are interested in. In particular, it does not require that the source code be in a deployable (or even buildable) state. This avoids problems with source files not being available or organised incorrectly, dealing with external jar files or other subsystems, or configuration issues.

We have implemented the algorithm as part of our tool Jepends. Jepends uses the results of the algorithm to build up the compilation dependency graph, and then analyses the graph in various ways. Jepends can compute a suite of sets for each of the application's source files: **Refers-to** — the *R'* set i.e., the other sources referred to directly by the names in the given source file; **Refers-to-tc** — the transitive closure of refers-to; **Referred-to-by** — the inverse of refers-to; **Referred-to-by-tc** — the transitive closure of referredto- by; **Cycles-thru** — a subset of all simple cycles (no repeated vertices) that a given source file participates in. The size of the refers-to and referred-to-by sets give the out-degrees and in-degrees of the corresponding vertex in the compilation dependency graph. The transitive closure relations determine what source files either require or are required by a given file during the compilation process. Currently Jepends outputs dependency profiles as text files that can be imported into tools such as Excel for sorting, graphing and further analysis. Table 1 shows part of the output, in this case the top four classes when sorted by Cycles-thru. The **TC** columns are the transitive-closure version of the column to the left. The fact that the numbers are the same for all classes in these columns is discussed in the next section.



Class	Referred-to-by	TC	Refers-to	TC	Cycles-thru
org....config.COConfigurationManager	164	1003	5	1373	3280
org....config.impl.ConfigurationDefaults	3	1003	10	1373	3279
com.....defaultplugin.StartStopRulesDefaultPlugin	3	1003	38	1373	3275
org....logging.LGLogger	107	1003	4	1373	3274
...	...	...	...	...	...

Table 1: Part of the output by Jepends. Class names have been elided.

The fact that Cycles-thru is a *subset* of all the simple cycles a given source file participates in requires further explanation. Efficiently finding all the simple cycles a given node in a directed graph participates in is a difficult problem[AYZ94]. One approach to finding all simple cycles (that is easily implemented in Java) is to find all

simple paths between each pair of nodes the graph and determine which of these paths also correspond to a simple cycle. A simple path corresponds to a simple cycle if there exists an edge in the graph from the
terminal node in the path to the initial node in the path. Several different paths can correspond to the same simple cycle and this is easily detected by checking that the paths contain the same nodes, and that these nodes occur in the same order (when they are arranged into a cycle).

Unfortunately finding all simple paths between all pairs of nodes is infeasible with respect to time for a graph of any decent size. Our approach is to keep track of all the simple cycles source files participate in that are encountered during the course of the depth first searches to construct the Refers-to-tc set of each node. In this regard Cycles-thru is a sample of the total cycles that pass through a node. More importantly it shows that a given node participates in *at least* this many simple cycles.

2.6 Results

In this section we demonstrate Jepends by using it on Azureus, an open-source application that provides peerto- peer file sharing[Azu05]. Azureus is written in Java 1.4 and release 2.3.0.0 comprises 1913 Java source files with approximately 114000 lines of non-comment source statements. Azureus are uses the Standard Widget Toolkit for its user interface (like Eclipse), and has no automated unit test suite.

We came across Azureus because it frequently appears on Sourceforge's top 10 lists for number of downloads and development activity. Our end-user experience of Azureus is that it is easy to use, stable and feature-rich. This is atypical of our end-user experience with other peer-to-peer file-sharing applications. It also raises the question 'Is Azureus's internal design indicative of its positive end-user experience?'.

Figures 1 and 2 show the distribution of set sizes in the referred-to-by and refers-to relations. In the figures, the x-axis is the size of the sets and the y-axis is the number of classes that have a given sized set. So figure 1 says that about 1800 classes have refers-to-by sets of size between 0 and 19. Both distributions show that small values are extremely common whereas large values are very rare. This is reminiscent of the power law relationships found by Marchesi et al[MPST04].

Figures 3 and 4 respectively show the distributions of the set sizes for refers-to-tc and referred-to-by-tc. The distributions in figures 3 and 4 are of particular interest. Both distributions show two distinct clusters: from 0-99 and 1000-1199 for referred-to-by-tc distribution, and from 0-99 and 1300-1499 in the refers-to-tc distribution. These seem to be very odd distributions—in the case of referred-to-by-tc, this says that between 1000 and 1199 classes depend (transitively) on nearly 1400 other classes.Furthermore, the distributions indicate no classes depend on (for example) 500 other classes. It is very much that classes depend on only a few classes (fewer than 100) or most of the classes.

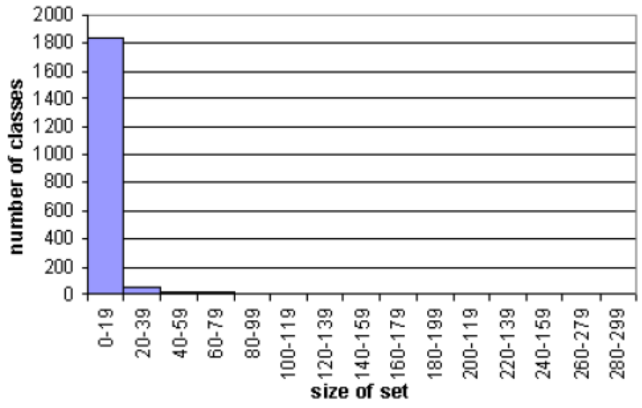


Figure 1: Azureus' referred-to-by distribution

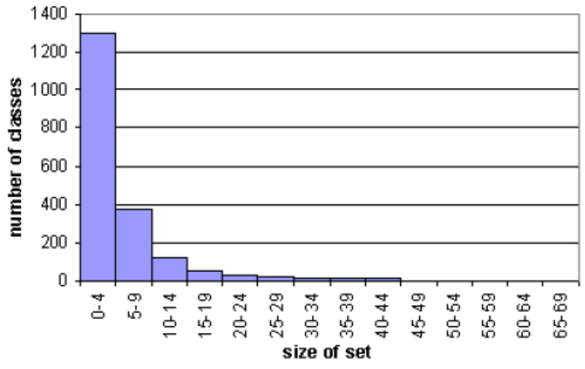


Figure 2: Azureus' refers-to distribution

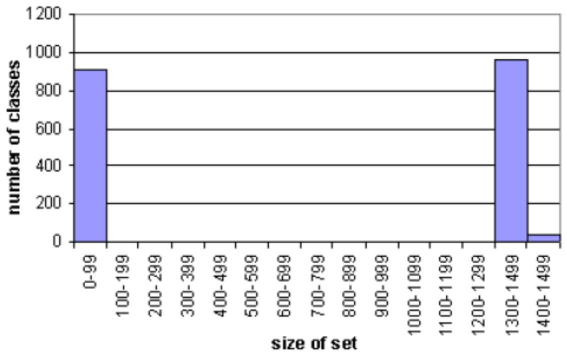


Figure 3: Azureus' refers-to-tc distribution

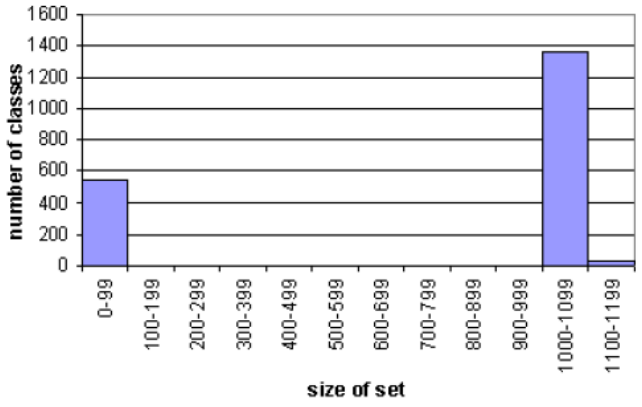


Figure 4: Azureus' referred-to-by-tc distribution

The question then is, is this distribution somehow characteristic of all applications, or somehow peculiar to Azureus. If it is peculiar to Azureus, then the presence of such distributions may tell us something about the nature of Azureus' design. We used our tool to examine the distribution of these relations in other systems such as Tomcat 5.5.9, Eclipse 3.0 and Netbeans 3.6 and found some clustering, but overall large valued clusters were less common than small valued clusters as exemplified by Tomcat's refers-to-tc distribution in Figure 5.

Now the question is, why does Azureus have such odd distributions? Is it just some particular characteristic of the application that is not related to the design, or is it indicative of some, possibly bad, design characteristic?

In fact, such distributions indicate the possible presence of long cycles. To see this, consider the distribution in Figure 3. The right-hand cluster indicates that of the approximately 1900 source files in Azureus, about 1000 of them depend (either directly or transitively) on 1300 or more other source files. The left-hand cluster indicates that the remaining 900 or so source files in the application depend on between 0 and 99 other source files. In fact the 900 source files in the left-hand cluster cannot depend on any in the right-hand cluster because of the transitivity. If a source file in the left-hand cluster depended on one in the right-hand cluster, it would depend on all the source files the latter depended on, which we know is 1300 or more, and so that source file should have been in the right-hand cluster.

Files in the right-hand cluster can refer to files in the left-hand cluster, but since there are at most 900 in the left hand cluster that means *every* file in the right-hand cluster must refer to at least *one other* file in the right-hand cluster, meaning there must be cycles within the right-hand cluster. The length of the cycles depends on the internal structure of the CDG, however we get hints by looking at the raw output of Jepends as shown in Table 1. As noted earlier, the values of the **TC** columns for the classes shown are all the same. This means that with transitive closure they all have the same set of classes that they depend on or are depended on, which could be explained by all of the classes belonging to a cycle.

It was the appearance of the odd distributions for Azureus compilation dependencies and other applications that led to our interest in cycles, and the introduction of cycle profiling to Jepends. If we use Jepends to profile the distribution of lengths of unique simple cycles we get the graph as shown in Figure 6. Note that because vertices in the graph can participate in more than one unique cycle, the sum of the frequencies is greater than the number of source files. The graph shows that there are a large number of long cycles in Azureus. Indeed 75% of the cycles in involve more than 50 nodes. Now the question is how we can use this information to identify possibilities for refactoring, which we discuss in the next section.

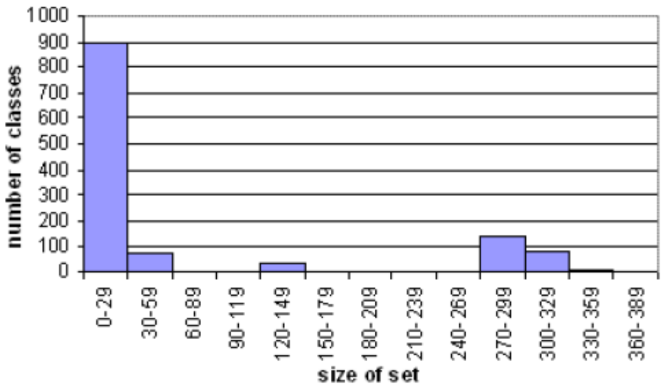


Figure 5: Tomcat’s refers-to-tc distribution

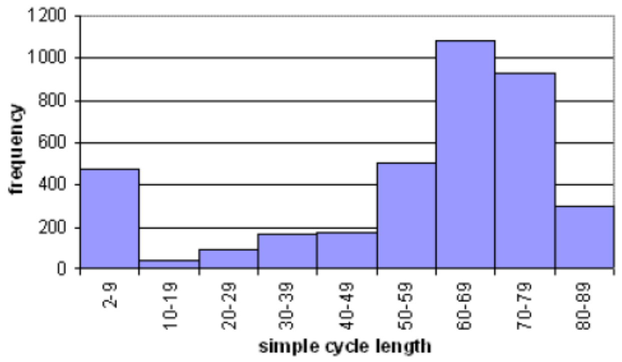


Figure 6: Azureus’ simple cycle length distribution

2.7 Refactoring

In this section we will explain how the analysis by Jepends can be used to indicate starting points for refactoring and measure the effect a refactoring on dependencies. The data in table 1 comes from Azureus and, as mentioned earlier, shows the top 4 classes when files are sorted by the number of cycles in which they participate.

Based on this data, we surmise that breaking the cycles through `COConfigurationManager` may greatly reduce the total number of (long) cycles in the system. A technique for breaking all cycles through `COConfigurationManager` would be to extract an interface from it and replace all existing references to its implementation with the extracted interface. In order to avoid a dependency on the interface’s implementation, we would have to further refactor the classes referencing `COConfigurationManager` not to create a new instance of, or statically depend on, its implementation.

While the ‘extract interface’ refactoring would definitely reduce the number of cycles in a system the overall effect on design quality by repeatedly performing this refactoring is dubious. The repeated use of the refactoring would dramatically increase the total number of source files in the system and the existence of the interfaces defined in these files would be justified on the basis of reducing cycles alone.

A refactoring whose justification can be more strongly argued is more subtly indicated by the data in table 1. The name `COConfigurationManager` suggests that its class is involved in something to do with configuration, potentially belonging to a configuration subsystem. Upon inspection of this class’s source we find that it is the Façade into the configuration subsystem. The configuration subsystem is responsible for loading and saving user configurable parameters used throughoutAzureus’s code (e.g. the directory to which files download, and the maximum download and upload rates). These parameters are saved to flat text files so they can remain persistent between executions of Azureus.

It is hard to believe that functionality as primitive as saving and reading properties from disk should transitively depend on 1373 other classes. We think that in a better design for the configuration subsystem would depend only on the threading subsystem and the logging subsystem. These two subsystems are themselves primitive and probably should not depend on any other source files in Azureus. By a brief code inspection we identified 5 classes relating to threading: `AEMonitor`, `AEMonSem`, `AERunnable`, `AESemaphore`, `AEThread`. These classes were mixed up with other utility-type classes in the `org.gudy.azureus2.core3.util` package. In the logging subsystem (comprising its own package) we found 4 source files: `ILoggerListener`, `LGAlertListener`, `LGLogger`, `LGLoggerImpl`. Since the configuration subsystem (again in its own package) contains 13 files we would expect `COConfigurationManager` to transitively refer-to no more than 22 other files (=5+4+13). In any case this is an order of magnitude less than its current 1373.

The point of this discussion is to support our claim that the analysis provided by Jepends provided very valuable insight into the current design of Azureus, and so provided a useful starting point for the refactoring process.

2.8 Conclusions

In this paper we have discussed how data from the automated analysis of source code can be used to identify opportunities for refactoring. We have developed an algorithm based on work by Lagorio on incremental compilation, that allows compilation dependency graphs to be created for an application. We have implemented this algorithm in Jepends, which also analyses the resulting graph. We have provided canonical examples of refactorings indicated by running Jepends over the open-source Java application Azureus.

Many characteristics of the distributions of dependencies we found in Azureus' source are not unique to Azureus. We have seen similar distributions in a number of other applications that we have analysed. However we have also seen different distributions (such as Tomcat's). The fact that different distributions are possible suggest that it may be possible to get a sense of the quality of the design by profiling these distributions. We are completing the analysis of these other applications to better understand the relationship between different profiles and design quality.

Jepends and the algorithm it is based on are Java specific. However the principles behind their development are not language specific. We intend to widen the scope of Jepends in order to carry out large-scale studies on commercial software.

Coauthor Declaration for Chapter 3 [BFN+06]

AUTHORSHIP STATEMENT

1. Details of publication and executive author

Title of Publication		Publication details
Baxter, Gareth, Marcus Frean, James Noble, Mark Rickerby, Hayden Smith, Matt Visser, Hayden Melton, and Ewan Tempero. **Understanding the shape of Java software.** In *ACM Sigplan Notices*, vol. 41, no. 10, pp. 397-412. ACM, 2006.		
Name of executive author	School/Institute/Division if based at Deakin; Organisation and address if non-Deakin	Email or phone
Ewan Tempero	**Dept of Computer Science, University of Auckland, NZ**	**ewan@cs.auckland.ac.nz**

2. Inclusion of publication in a thesis

Is it intended to include this publication in a higher degree by research (HDR) thesis?	Yes / ~~No~~	If Yes, please complete Section 3 If No, go straight to Section 4.

3. HDR thesis author's declaration

Name of HDR thesis author if different from above. (If the same, write "as above")	School/Institute/Division if based at Deakin	Thesis title
As above	**School of Information Technology, Deakin University**	Empirical Studies of Structural Phenomena Using a Curated Corpus of Java Code
If there are multiple authors, give a full description of HDR thesis author's contribution to the publication (for example, how much did you contribute to the conception of the project, the design of methodology or experimental protocol, data collection, analysis, drafting the manuscript, revising it critically for important intellectual content, etc.)		
For this paper I collected the metrics studied using both the Jepends tool I had developed and curated corpus I had also developed (that were both as part of my own, independent PhD research), I defined many of the metrics that were collected, performed the initial visualization of the distributions of those metrics using Excel, I identified many of the related works that were cited in the paper, and helped with writing and editing of various parts of the paper. What I was not involved in was the statistical analysis of the metrics distributions and the writing associated with these sections. Relating to the conception of the paper, I was interested in powerlaws and aware of the 2003 Counsell and Wheeldon on Powerlaws (indeed it is cited earlier my ASCS2006 "refactoring" paper which pre-dates this one) but it was James Noble who had the idea for us all to collaborate on this and submit this paper to OOPSLA.		
I declare that the above is an accurate description of my contribution to this paper, and the contributions of other authors are as described below.	Signature and date	[illegible] 18th Dec 2016

4. Description of all author contributions

Name and affiliation of author	Contribution(s) (for example, conception of the project, design of methodology or experimental protocol, data collection, analysis, drafting the manuscript, revising it critically for important intellectual content, etc.)
A/Prof Ewan Tempero Dept of Computer Science University of Auckland New Zealand	Primary author of paper; PhD supervisor of Hayden Melton at University of Auckland.
Dr Gareth Baxter Department of Physics, I3N, University of Aveiro	I carried out statistical analysis of the data, created some of the figures and contributed to writing and revision of some sections of the text for this paper. I did not contribute to the conception, design and data collection.

A/Prof Marcus Frean School of Engineering and Computer Science Victoria University of Wellington NZ	Some role in conception, analysis (of powerlaw metric), and some minor input into the drafting of manuscript.
Prof James Noble School of Engineering and Computer Science Victoria University of Wellington NZ	Proposed the idea of the research, collected and coordinated the research team, oversaw the writing and preparation of the paper.
Mark Rickerby	*(Unknown—Did not respond to my email sent via LinkedIn)*
Hayden Smith	Hayden [Smith] assisted with the tool that took compiled Java class files and produced various metrics, as well as running the tool against programs in the corpus.*
Prof Matt Visser Dept of Mathematics Victoria University of Wellington NZ	Prof Visser is a mathematician/theoretical physicist, not a computer scientist. His contribution was to the mathematical background and general framework of the paper --- power laws, their infra-red behaviour and representation in terms of harmonic numbers and Riemann zeta functions.

***Note from Hayden Melton regarding this contribution since I am required to sign to verify correction of stated contributions—the tool to which Smith refers is not the Jepends tool to which I refer and am the sole creator, but an entirely different one. I also believe the corpus referred to is not the Qualitas Corpus but a smaller corpus that VUW had been using and was superceded by the former.**

5. Author Declarations

I agree to be named as one of the authors of this work, and confirm:

i. that I have met the authorship criteria set out in the Deakin University Research Conduct Policy,

ii. that there are no other authors according to these criteria,

iii. that the description in Section 4 of my contribution(s) to this publication is accurate,

iv. that the data on which these findings are based are stored as set out in Section 7 below.

If this work is to form part of an HDR thesis as described in Sections 2 and 3, I further

v. consent to the incorporation of the publication into the candidate's HDR thesis submitted to Deakin University and, if the higher degree is awarded, the subsequent publication of the thesis by the university (subject to relevant Copyright provisions).

Name of author	Signature*	Date
Ewan Tempero	[illegible]	12/12/2016
James Noble	[illegible]	2/12/2016
Matt Visser	Matt Visser	12/12/2016
Marcus Frean	[illegible]	14 Dec 2016
Gareth Baxter	[illegible]	2/12/2016
Hayden Smith	[illegible]	15/12/2016
Mark Rickerby	*(Did not respond to my email sent via LinkedIn)*	

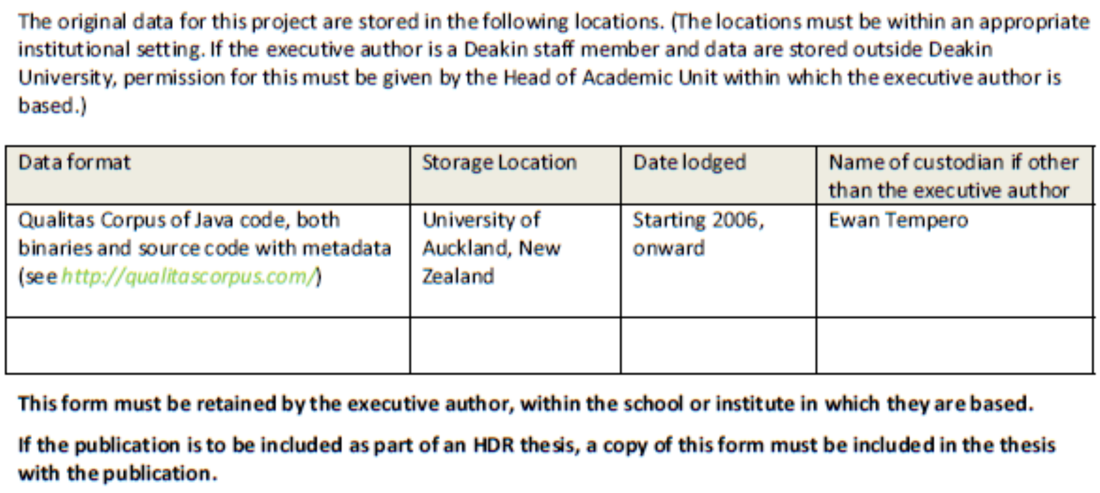
6. Other contributor declarations

I agree to be named as a non-author contributor to this work.

Name and affiliation of contributor	Contribution	Signature* and date

* If an author or contributor is unavailable or otherwise unable to sign the statement of authorship, the Head of Academic Unit may sign on their behalf, noting the reason for their unavailability, provided there is no evidence to suggest that the person would object to being named as author

7. Data storage



The original data for this project are stored in the following locations. (The locations must be within an appropriate institutional setting. If the executive author is a Deakin staff member and data are stored outside Deakin University, permission for this must be given by the Head of Academic Unit within which the executive author is based.)

Data format	Storage Location	Date lodged	Name of custodian if other than the executive author
Qualitas Corpus of Java code, both binaries and source code with metadata (see *http://qualitascorpus.com/*)	University of Auckland, New Zealand	Starting 2006, onward	Ewan Tempero

This form must be retained by the executive author, within the school or institute in which they are based.

If the publication is to be included as part of an HDR thesis, a copy of this form must be included in the thesis with the publication.

Chapter 3 Understanding the Shape of Java Software

Large amounts of Java software have been written since the language's escape into unsuspecting software ecology more than ten years ago. Surprisingly little is known about the structure of Java programs in the wild: about the way methods are grouped into classes and then into packages, the way packages relate to each other, or the way inheritance and composition are used to put these programs together. We present the results of the first in-depth study of the structure of Java programs. We have collected a number of Java programs and measured their key structural attributes. We have found evidence that some relationships follow *power-laws*, while others do not. We have also observed variations that seem related to some characteristic of the application itself. This study provides important information for researchers who can investigate how and why the structural relationships we find may have originated, what they portend, and how they can be managed.

3.1 Introduction

Much of software engineering has focused on how software could or should be written, but there is little understanding of what actual software really looks like. We have development methodologies, design principles and heuristics, but even for a well-defined subset of software, such as that written in the Java programming language, we cannot answer simple questions such as "How many methods does the typical class have?" or even "Is there such a thing as a 'typical class'?"

What we would really like to know about software is "Is it good?" that is, does it have quality attributes such as high modifiability, high reusability, high testability, or low maintenance costs. We believe current methodologies lead to good software, but without knowing what good software looks like, we cannot know that the methodologies are actually working. We are left with circular arguments of the form "The methodologies are good because the software is good, and the software is good because the methodologies are good." Understanding the shape of existing software is a crucial first step to understanding what good software looks like.

Just as biologists classify species in terms of shape and structure and ecologists study the links and interactions between them, we have been collecting a body of software and analysing its abstract form. We remove semantics and focus on the network of connections where information flows between components. Just as biologists (and other scientists) seek to understand the characteristics of the population under study, so too would we like to know such basic features as the distributions of the software structures we find.

Of specific interest are recent claims that many important relationships between software artifacts follow a 'power-law' distribution (e.g. [WC03]). If this were true, it would have important implications on the kinds of empirical studies that are possible. One issue is the fact that a power-law distribution may not have a finite mean and variance. If this is the case, the central limit theorem does not apply, and so the sample mean and variance (which will always be finite, because the sample size is finite) cannot be used as estimators of the population mean and variance. This would mean that basing any conclusions on sample means and variances without fully understanding the distribution would be questionable at best.

In this paper, we extend past similar studies in two ways. First, we examine a much larger sample than previous studies. We have analysed a corpus of Java software consisting of 56 applications of varying sizes, and measured a number of different attributes of these applications. Second, we consider distributions other than those following a power-law. We find evidence that supports claims by others of the existence of power-law relationships, however we also find evidence that some distributions do *not* appear to obey a power-law. Furthermore, whether or not a relationship follows a power-law appears to depend on an identifiable characteristic of the relationship, namely, whether or not the programmer is inherently aware of the size of the relationship at the time the software is being written. We also see variations between applications. We speculate that this may be due to some characteristic in the application's *design*, that is, some property of the design is reflected in the distribution of some measurements.

The rest of the paper is organised as follows. In Section 2, we discuss the motivation for our study. Section 3 describes in detail the salient features of our study, namely the corpus we use and the metrics we gather. In Section 4 we give the analysis of our results, and in Section 5 we give our interpretation of this analysis. Section 6 discusses the most relevant related work, and we give our conclusions in Section 7.

3.2 Motivation and Background

Software systems are now large, complex, and ubiquitous, however surprisingly little is known about the internal structures of practical software systems. A large amount of research has studied how software 'ought' to be written, how it 'should' be structured. Many rules, methodologies, notations, patterns and standards for designing and programming such large systems [GHJV95] [Kru00] [Obj04] have been produced. Psychological models have been constructed of the programming process [ES84][Wei85]. Quantitative models of software have been designed to predict the effort required to produce a system, measure the development rates of software over time (process metrics) or measure the volume of software in a system and its quality (product metrics)—see e.g. [FP96][Jon86][PV03]. But we know very little about the large-scale structures of software that exists in the real world.

With the methodologies, notations, and other advice that has been developed, we should be able to say something about the software that results *if such advice is followed*. However the conditional is key—until recently there was very little work done in determining even if the advice that has been offered is actually been taken. There is some evidence that common advice is not being followed. For example, a number of people have advised against creating cycles of dependencies in software, but recent evidence suggests that not only do programmers regularly introduce cycles, but they are often very large [MT06].

One consequence of much of the advice offered with respect to object-oriented design is what we call the *Lego Hypothesis*, which says that software can be put together like Lego, out of lots of small interchangeable components [PNFB05][SGM02]. Software constructed according to this theory should show certain kinds of structure: components should be small and should only refer to a small number of closely related components.

In fact, we don't know whether or not this is true, because we lack models describing the kinds of large structures that exist in real programs. There are no quantitative, testable, predictive theories about the internal structures of large scale systems, or how those structures evolve as programs are constructed [BJ03][Hee03].While design patterns, rules, metrics and so on, can give guidance regarding developing program structure, they cannot predict the answers to questions about the large-scale structure that will result, such as: in a program of a given size, how many classes or methods will exist? How large will they be? How many instances of a each class will be created? How many other objects will refer to any given object? We need answers to these kinds of questions in order to be able understand how large scale software is actually organised, built, and maintained in practise.

Recently there has been an interest in looking for *power-law* relationships in software. A distribution of the number of occurrences N_k of an event of size k is a power-law if it is proportional to k raised to some power s. A common method used to detect possible power-laws is to rank the event sizes by how often they occur, and then plot N vs. the rank on logarithmic scales. A distribution following a power-law will appear as a line with slope s.

Studies of computer programs have considered both static [VCS02][VS03][WC03] and dynamic [NB03][PNB04][PNFB05] relationships, in different forms of software as diverse as LISP, visual languages, the Linux kernel, and Java applets[CG77] [NB01] [NB03] [PNB04] [PNFB05] [SFCMV02] [VS03], and the design of Java programs[DH99][VCS02][VS03]. The conclusions from these studies is that power-laws appear to be quite common.

Our work follows from Wheeldon and Counsell, who examined a number of inter-class relationships in Java source code, namely Inheritance, Interface, Aggregation, Parameter Type, and Return Type in three Java systems: the core Java class libraries, Apache Ant, and Tomcat [WC03]. We attempted to reproduce the Wheeldon and Counsell study, and found examples of their metrics that, for some applications, did not appear to obey a power-law. One example is shown in Figure 1 (which appears again, with full explanation, as Figure 3). This figure shows a plot organised as described above —it is a log-log plot of frequency of occurrence of different values of a particular metric. The data in this figure seems to have a distinct curve to it. Had we plotted this on a normal scale, we would see something like a power-law curve, except 'truncated' at the high end. This figure casts some doubt as to whether the distribution shown is a power-law.

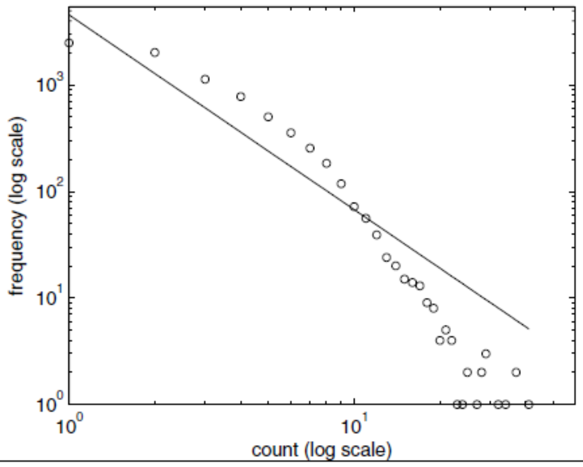


Figure 1 A distribution that does not appear to obey a power-law. Open circles are data, solid line is best fit power law distribution.

Our experience raised two questions. The first is, do the relationships others have studied really obey a power-law? While the evidence provided is compelling to the naked eye, there is little analytical support. In this paper, we will provide such an analysis to support our claims. The second question is, are the studies representative of software in general. This is not a question that can be answered easily due to the scale involved, however, our study involves a much larger corpus than other studies, and so provides better support for our claims.

3.3 Method

3.3.1 Gathering the Corpus

The corpus consists of 56 applications whose source code is available from the web. Many of the applications were chosen because they have been used in other studies (e.g., [GM05][GPV01][PNFB05]), although comparison to these other studies isn't possible as version numbers were not always provided. Also, we weren't always able to acquire all applications used in those other studies. Further applications were then added to the corpus based on software that we were familiar with (e.g. Azureus, ArgoUML, Eclipse, NetBeans). Finally we identified popular (widely down-loaded) and actively developed open-source Java applications from various web-sites, including: developerWorks[4], SourceForge[5], Freshmeat[6], Java.net[7], Open Source Software In Java[8] and The Apache Software Foundation[9]. Figure 2 gives an indication of the distribution of the size of the applications, measured in terms of the number of top-level classes. Appendix B gives more details of contents of the corpus we used.

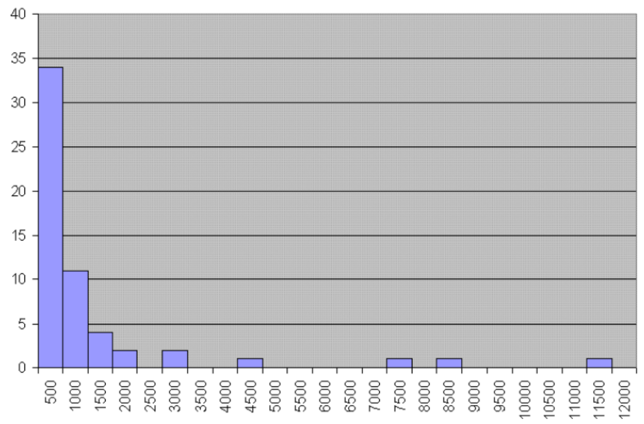


Figure 2 Distribution of application size in Corpus.

3.3.2 3.2 Metrics

There are a number of variables that must be taken into account when carrying out this kind of research. In the interests of allowing others to reproduce and extend our results, we discuss our choices in detail.

Any Java program makes some use of the Standard API, and so there is a question of how much the Standard API is counted when doing the analysis. For example, when counting the number of methods per class, should the number of methods in the java.lang.String class be counted, or should the number of methods that use String as a parameter or return type be counted? This type is so heavily used that measuring its use seems likely to distort the results, and so it would seem reasonable to not consider it. However there are also less frequently used types, such as

[4] http://www-128.ibm.com/developerworks/views/java/downloads.jsp
[5] http://sourceforge.net/
[6] http://freshmeat.net/
[7] http://community.java.net/projects/
[8] http://java-source.net/
[9] http://apache.org/

java.util.jar.Pack200, that seem less likely to distort the results and so maybe should be counted. It is not clear where to draw the line.

In this analysis we have chosen to consider only the human editable aspect of an application's construction, that is, the source code that is under the control of the application developers. For this reason, when metrics have been computed, we have considered only those classes declared in the source files of the application. Uses of the Standard API (and indeed any other API used but not constructed for the application) are not considered. In the descriptions below, the phrase "in the source" will reinforce this choice.

Note that in the case where the application is the JDK/JRE, it is the Standard API being analysed. All the metrics have been computed from the byte code representation of 'top-level' classes, that is, classes that are not contained within the body of another class or interface [Gos00, chapter 8]. Relationships relating to inner classes are merged with their containing class. To restrict the analysis to only those classes in the application's source code, names discovered in the byte code were filtered according to package names of packages in the source code. Note that this means our analysis is limited to those applications that use a package structure.

We used two methods to carry out the analysis. One method applied to the byte code directly, using the Byte Code Engineering Library (BCEL)[10]. The other applied javap, a Java byte code disassembler that outputs representations of classes in a plain text format. From this, we were able to extract information about the structure of fields, methods, and opcode instructions, which we used to build a meta model of each application as a nested collection of the basic types 'package', 'class', 'method', and 'field'. These collections gave us a simple source for calculating metrics we were interested in. When byte code is generated, some information (particularly type information) is thrown away. This means some of our results will not match a similar analysis done directly on the source code. We discuss this point in more detail when we present the metrics.

[10] http://jakarta.apache.org/bcel

Many of the metrics we use come from Wheeldon and Counsell, as indicated in the list below, and we use their naming scheme where possible [WC03, Figures 8-10]. Due to the difficulty in interpreting their descriptions [WC03, Figure 1] we give more detailed definitions here, with a more formal treatment in Appendix A. We will use the abbreviations given below. Where the abbreviation does not match the Wheeldon and Counsell names, we indicate the phrase on which they are based.

Our definitions assume that there is only one *top-level* [Gos00] type declaration per source file (.java file). That is, we explicitly rule out the following situation, where two classes are declared in the same file (or *compilation unit*).

```
// A.java containing two class declarations
public class A { ... }
class B { ... }
```

The main reason for making this assumption is that it simplifies the definitions. However, compiling the file A.java above will yield two files, A.class and B.class. Since there is no requirement that a class be declared to be public, even when it is the only class in a compilation unit, there is no way to tell from looking at B.class that it was generated from the same source file as A.class.

In the following description, we occasionally need to distinguish between when a name refers to a **class** and when it refers to an **interface**. When no distinction is necessary, we will say the name refers to a **type**.

Number of Methods nM (WC) For a given **type**, the number of all methods of all access types (that is, public, protected, private, package private) declared (that is, not inherited) in the type.

Number of Fields nF (WC) For a given **type**, the number of fields of all access types declared in the type.

Number of Constructors nC (WC) For a given **class**, the number of constructors of all access types declared in the class. Note that since the measurements are taken from the byte code, this is guaranteed to be at least 1. If no constructor is specified, the Java compiler automatically generates a default public nullary constructor that is included in the byte code.

Subclasses SP — Subclass as Provider (WC) For a given **class**, the number of top-level classes that specify that class in their extends clause.

Implemented Interfaces IC —Interface as Client (WC) For a given **class**, the number of top-level interfaces in the source that are specified in its implements clause. For a given **interface**, the number of top-level interfaces in the source that are specified in its extends clause.

Interface Implementations IP —Interface as Provider (WC) For a given **interface**, the number of top-level classes in the source for which that interface appears in their implements clause. Note that when an inner class implements a given interface, it is the top-level class that contains it that is counted.

References to class as a member AP — Aggregate as Provider (WC) For a given **type**, the number of top-level types (including itself) in the source that have a field of that type.

Members of class type AC —Aggregate as Client (WC) For a given **type**, the size of the set of types of fields for that type.

References to class as a parameter PP — Parameter as Provider (WC) For a given **type**, the number of top-level types in the source that declare a method with a parameter of that type.

Parameter-type class references PC—Parameter as Client (WC) For a given **type**, the size of the set of types used as parameters in methods for that type.

References to class as return type RP—Return as Provider (WC) For a given **type**, the number of top-level classes in the source that declare a method with that type as the return type.

Methods returning classes RC —Return as Client (WC) For a given **type**, the size of the set of types used as return types for methods in that type.

Depends on DO For a given **type**, the number of top-level types in the source that it needs in order to compile. The intent is to count all top-level types from the source whose names appear in the source for the type. There are some rare situations (when only methods from parent classes are called on the object) where the types of local variables are not recorded in the byte code. Our experience is that this happens sufficiently rarely to have no effect on the results.

Depends On inverse DOinv For a given **type**, the number of type implementations in which it appears in their source.

Public Method Count PubMC The number of methods in a **type** with public access type.

Package Size PkgSize The number of **types** contained direction in a package (and not contained in sub-packages).

Method size MS The number of byte code instructions for a method. Note that this is not the number of bytes needed to represent the method.

3.4 Results

We have applied the 17 metrics described in the previous section to 56 applications from our corpus. This has yielded more data than can be conveniently shown here, so instead we have done some preliminary analysis based on various assumptions as to what the distribution of the data is, and present the results of analysis.

3.4.1 Analysis

The raw data consists of a number for each ‘element’ (method, top-level class, package) in each application. The first step was to group all values by application, count the number of occurrences of each value and record that in order of value. The primary goal of our analysis was then to determine whether the resulting distribution obeyed a power-law.

Some of the distributions derived from our analysis of software structure look like straight lines when plotted with logarithmic scales on both axes. This is the hallmark of a power-law distribution, which is interesting because of its ‘scale-free’ properties, which we will describe below. Any other distribution will not be exactly a straight line in such a plot.

Not all the plots look exactly straight. Some have a sort of curve to them. We can respond by either saying that we do not care, as they are *nearly* straight, at least for part of the range, or we can say that they really are not power-laws at all, and are characterised by some other distribution. Secondly, even if it ‘really’ is a power-law, because the data is noisy and because there is a finite sample size and a finite range of ‘sizes’, a power-law curve won’t exactly fit the data, especially at large values of the metric. This also means that some alternative distributions might be made to fit the data just as well—we might not be able to discriminate, even for the plots that look pretty straight.

Our approach is to take the data, and do rigorous best-fits to several different distributions, and see first whether it is reasonable to fit a power-law, second whether a power-law is more reasonable than the others, third whether the data can be divided into two or more groups according to which distribution fits ‘best’.

3.4.1.1 Power-Law

In general a power-law distribution has the form [21]:

$$p_{powlaw}(x) \propto x^{-\alpha}\,, \tag{1}$$

where α is a positive constant and we assume x to be non-negative. In our case, x is the value of the metric as defined in the previous section. If $\alpha < 1$ there must be a finite maximum value of x, in order for the distribution to be normalisable. If $\alpha > 1$, normalisability requires that the minimum value of x not be equal to zero. For $\alpha \leq 2$ the mean of the distribution is infinite (assuming there is no upper cutoff in x). When

$\alpha > 2$ the mean is proportional to the small-x cutoff. For $\alpha \leq 3$ the variance is also infinite. One consequence of this fact is that the central limit theorem doesn't hold for such distributions, so the mean and variance of a sample (which will always be finite) cannot be used as estimators for the population mean and variance.

A distribution is said to be scale free if [21]:

$$p(bx) = g(b)p(x)\,, \tag{2}$$

where g does not depend on x. This means the relative probability of occurrence of 'events' of two different sizes (bx and x) depends only on the ratio b, and not on the 'scale' x. One of the reasons for the interest in power-laws is that they possess this scale-free property. If we can show that the distributions we see in our analysis of software obey a power-law, we can say that there is no characteristic size (where 'size' might mean in-degree, for example) to the components. A scale-free distribution such as a power-law would contradict the Lego Hypothesis.

While an idealised power-law distribution might be strictly scale-free, for the distributions we encounter in real systems this can only be approximately true. The data in our studies only occurs at discrete, integer values of x. This imposes a small-size cutoff on x — the smallest value of x we measure is 1. There is also a large-size cutoff of x, as the programs in the corpus are of finite size. Nevertheless, we are still interested in power-laws. The scalefree property (2) may still hold over a limited range. We can never say for certain that a distribution *is* a power-law – because we are always dealing with measured data that involve some noise, and also finite size effects — but we might be able to say that it is approximately a power-law, well characterised by a power-law over a large range, or more likely to be a power-law than something else.

3.4.1.2 Other Candidates

Given our experience with plots such as that shown in Figure 1, we are interested in distributions that are close to power-laws, but resemble the curves we have seen. Two other distributions which have some credibility as 'natural' distributions are:

Log-normal distribution. Power-laws and log-normals look the same at low values of 'x' (i.e., at the high frequency end), but the tail is 'fatter' for a power-law. For continuous x a log-normal probability density function is defined as:

$$p_{lognorm}(x) = \frac{1}{x\sigma\sqrt{2\pi}} \exp\left\{ \frac{-(\ln x - \mu)^2}{2\sigma^2} \right\}, \qquad (3)$$

while for discrete values of x, the normalisation will be more complicated, and the distribution is of absolute probability, not probability density. Note that our data is not ranked, so it is usually, but not necessarily monotonically decreasing with x: sometimes the smallest value of x does not have the highest frequency. Log-normal distributions can reproduce this pattern, but to fit a power-law we must treat this 'turnover' as a statistical anomaly.

Stretched exponential. This is known to occur in natural distributions [LS98] (it is the same as the two-parameter Weibull distribution [Wei51] which is used to model electrical component failure probabilities):

$$p_{strexp}(x) = \frac{c}{x_0} \left(\frac{x}{x_0} \right)^{c-1} \exp\left\{ - \left(\frac{x}{x_0} \right)^{c} \right\}. \qquad (4)$$

Again, this is the continuous x version of the distribution. The form is the same in the discrete case, but the normalisation is different. A stretched exponential looks just like a power-law for small values of x, but has a sort of exponential behaviour for large x.

Both of these (depending on the choice of parameters) are slightly curved on a log-log plot, so they are likely to be good fits to the data we have that is not exactly straight. Neither has the long tail characteristic of a power-law, so the curves drop off sharply at the right hand side of a log-log plot.

The distinguishing features of power-laws are therefore 'straightness' in the log-log domain, and not dropping off as fast as the others for large values of x. This is sometimes called a 'fat tail' or 'long tail', in contrast with the 'truncated tail' evident in Figure 1. One potential problem is that the data is poorest in this tail region—our best statistics will be at the non-tail end.

3.4.1.3 Weighted Least Squares Fits

Fitting a distribution to data means choosing the parameters of the distribution so that it is 'closest' to the data. One way to do this is to minimise the sums of the squares of the differences between the data values and the distribution values.
Suppose the data takes value h_i at x_i, where i runs from 1 to k, the number of data points. If the value of the distribution at x_i is given by $f(\alpha, \beta, x_i)$, where α and β are the parameters of the distribution, we want to choose α and β so that the *residual*:

$$Q = \sum_{i=1}^{k} [h_i - f(\alpha, \beta, x_i)]^2 \tag{5}$$

is as small as possible.

Weighted least squares fitting is where we use this method but allow for different uncertainties in different data points by introducing a weight to each square in the sum:

$$Q = \sum_{i=1}^{k} w_i [h_i - f(\alpha, \beta, x_i)]^2 \ . \tag{6}$$

w_i should reflect how much uncertainty there is in the value of a data point. We set $w_i = 1/h_i$. Thus

$$Q = \sum_{i=1}^{k} \frac{1}{h_i} [h_i - f(\alpha, \beta, x_i)]^2 \ . \tag{7}$$

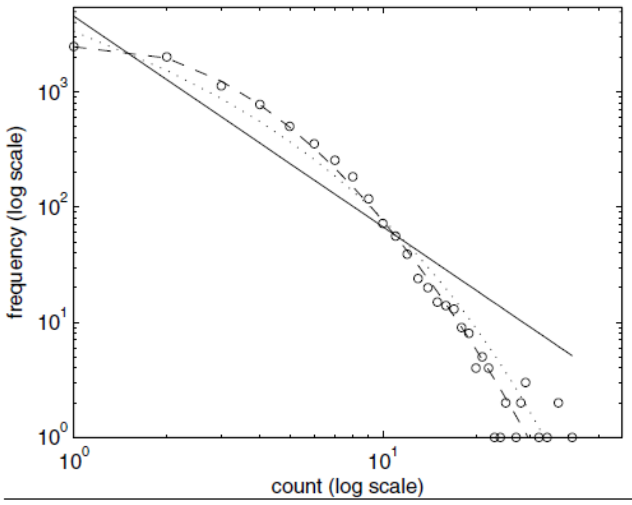


Figure 3 AC distribution and fitted curves for Eclipse. Open circles are data, solid

line is best-fit power-law, dashed line is best-fit log-normal and dotted line is best-fit stretched exponential.

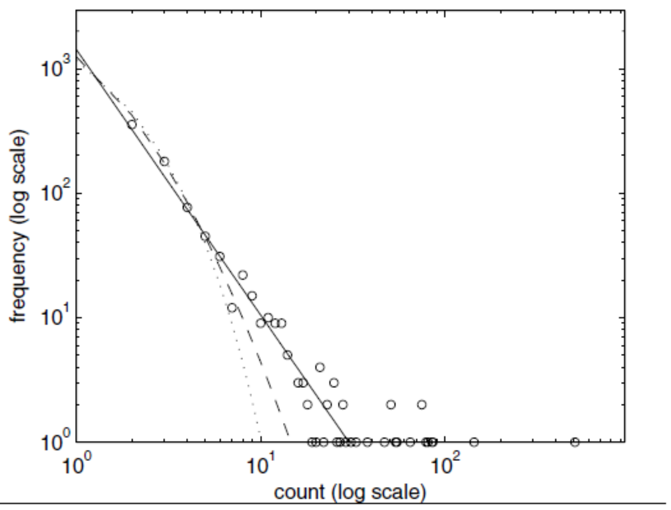


Figure 4. AP distribution and fitted curves for NetBeans.

3.4.1.4 Uncertainty and Confidence Intervals

If f is the 'true' distribution, we would have $E[h_i] = f(\alpha, \beta, x_i)$ where $E[z]$ denotes the expected value of z. Expanding each term in (7) and neglecting higher terms we find

$$E[Q] \sim k - 1 \tag{8}$$

And

$$\text{Var}[Q] = E[(Q - E[Q])^2] \sim k - 2\,. \tag{9}$$

We have assumed h_i is binomially sampled from a distribution with mean f/N, where N is the sample size, $N = \Sigma_i h_i$.

This gives us a way to estimate how good our fit is. We have effectively a distribution for Q, based on our assumption that the data follows the candidate distribution f. We can then choose a *Confidence Interval* (CI) for Q, and if the value for Q that we actually find from our fitting procedure actually falls within this range, we can take this as evidence for our assumption about f.

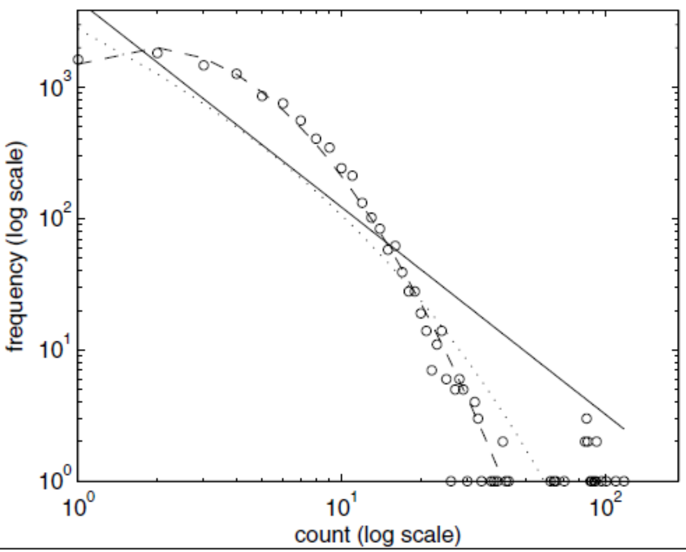


Figure 5. PC distribution and fitted curves for Eclipse.

For example, if the distribution is 'really' the one we have fitted, we would expect Q to be within 1.64σ of $E[Q]$, where $\sigma = \sqrt{(\mathrm{Var}[Q])}$, 90% of the time. $E[Q] \pm 1.64\sigma$ is called a 90% confidence interval (CI), and if the minimum value of the residual Q that we *do* get falls within this range, we say that the distribution fits the data at the 90% CI. (This is *not* the same as saying that "we are 90% sure the distribution is right.")

3.4.1.5 4.1.5 Fitting the data

In the current study, the minimisation of equation (7) was done numerically, with $f(\alpha, \beta, x_i)$ replaced by each of the three distributions (1), (3) and (4) in turn. The raw data is in the form of frequencies occurring at integral values of x. Note that the normalisation of these distributions at discrete values differs from the normalization of a continuous distribution, and it is important to take this into account. This normalisation depends of course on the parameter values. The log normal and stretched exponential distributions each have two parameters, while the power-law distribution is defined by a single parameter. A second parameter could be introduced by allowing the constant of normalisation to vary (in a log-log plot, a power-law appears as a straight line, with slope given by the single parameter, α, also known as the 'exponent'. The 'offset' of the line is given by the normalisation constant, so fitting an offset parameter is equivalent to fitting the normalisation constant). We found that the fit was very similar when the fit was done with only a single parameter (calculating the normalisation explicitly), returning very similar exponent values and residuals.

The aim of this exercise is mainly to establish the plausibility of the different distributions fitting the data, therefore we do not give uncertainties in the fitted parameters, or speculate on the interpretation of, for example, different fitted power-law exponents.

Table 1 shows a small excerpt from the results of the fit process. This shows the estimated parameters for each of the three distributions using the full datasets: a_pow is for power-law, m_log and s_log are for log-normal, and a_str and b_str are for the stretched exponential. The next three columns show the residuals for each of the fitted curves, tot_cnt is the sum of the frequencies, and the last column is the number of data points.

Recall that the expected value for the residuals is $k-1$ and the variance is $k-2$. This means, for the first row of Table 1 (the AC metric) the 90% confidence interval would be 25 $\pm$ 8.03 ($1.64 \times \sqrt{24}$), and so we can conclude that the log-normal distribution fits the data at the 90% CI, but the other two distributions do not.

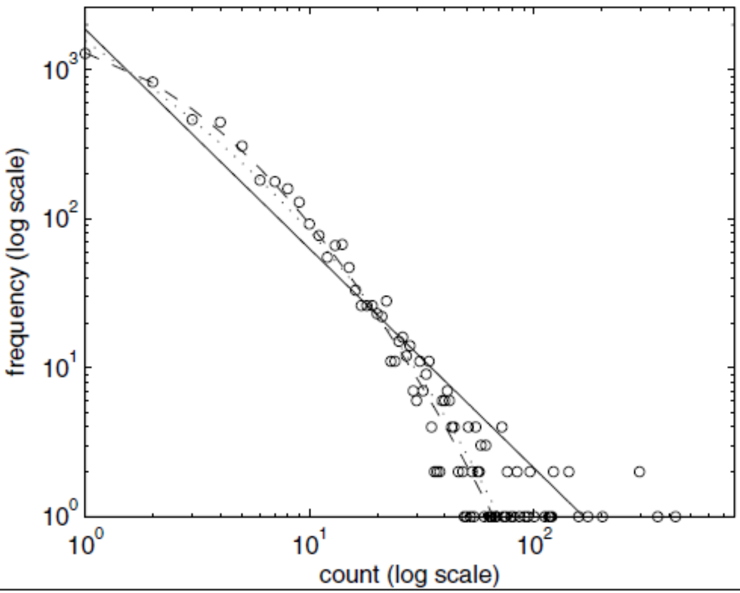


Figure 6. nF distribution and fitted curves for JRE.

Figure 3 shows an example of a plotted dataset with fitted curves (and is the same as Figure 1). This figure is a log-log plot of the number of types (*y*-axis) having a given number of fields (*x*-axis), that is, the AC metric, for Eclipse. The best-fit for a power-law is shown as a solid line, the best fit for the log-normal is shown as a dashed curve, and the best fit for the stretched exponential is a dotted curve. In this case, there is a pronounced curve in the data, and in fact the log-normal has a much better fit than the power-law. Figures 4-15 show a representative sample of fitted curves for different metrics and different applications. The parameters and residuals for these curves are shown in Table 2.

3.4.1.6 Summarising the results

For each metric of each program in the corpus, the fits were done first to the whole set of available data, then the number of points was reduced by removing 5, 10, 15, or 20 percent of the data points (or 'cuts') from both ends—that is, using only the 'middle' 90, 80, 70, or 60 percent of the non-zero data points. The residuals for each fit were then compared for the three distributions. We checked whether each fit was consistent with the data at 95%, 90%, 80% and 60% confidence intervals, and then the power-law fit was compared to the best (residual closest to the expected value) of the other two fits. Each metric for each program could then be classified at each CI with 'flags' as follows:

a Power-law residual is within the CI and both other residuals outside CI.
b Power-law residual within CI and one or both of the other residuals within CI.
c Log normal and/or stretched exponential residual within CI, but power-law residual outside CI.
d None of the residuals within CI.
x No data.

Roughly speaking, this order (ignoring *x*) represents *decreasing* support for the distribution of the data being a power-law. While *b* does not rule out a power-law, the fact that it fits one of the other candidate distributions indicates more doubt than *a* indicates. Since we chose our other candidate distributions to be close to power-law, a *d* suggests that not only do we not have a power-law, but we do not even have something close.

Metric	**a_pow**	**m_log**	**s_log**	**a_str**	**b_str**	Q_{pow}	Q_{log}	Q_{str}	**tot_cnt**	k
AC	1.72	0.62	0.83	0.64	1.03	163.54	32.63	52.79	668	26
AP	3.03	0.52	8.40	1.94	0.95	12.06	410.05	19.61	326	23
DO	1.11	1.56	0.78	0.28	0.63	1004.79	187.74	575.22	1251	97
DOinv	1.12	-3.20	3.80	0.33	0.54	1045.67	684.07	650.27	1251	634
IC	2.12	-0.23	0.85	1.09	0.90	3.91	8.92	12.72	89	14
IP	3.29	0.72	7.79	2.04	0.97	8.18	240.15	2.88	157	9
MS	0.91	2.38	1.20	0.08	0.57	7304.28	1354.52	5545.82	9859	1854
PC	1.65	0.68	0.85	0.61	1.05	254.13	22.11	61.78	1105	18
PP	1.83	-0.30	1.20	0.69	0.89	8.27	14.55	19.22	127	113
PubMC	1.36	0.95	1.14	0.42	0.98	199.13	70.02	110.66	1005	306
RC	1.55	-1.07	1.90	0.52	0.76	994.56	510.30	426.12	1240	38
RP	2.44	-0.30	0.75	1.51	0.86	25.34	41.39	50.76	263	31
SP	1.41	-2.95	8.80	0.57	0.80	37.45	47.79	36.81	133	94
nC	3.07	-0.06	0.50	1.72	0.99	37.32	13.01	32.34	1153	10
nF	1.40	0.72	1.24	0.45	1.03	80.27	36.37	43.36	668	146
nM	1.21	1.22	1.15	0.33	0.93	292.00	113.29	205.25	1170	320
pkgSize	0.92	2.80	2.85	0.00	1.19	13.73	12.53	13.51	72	128

Table 1 The estimated parameters for the three distributions for arguuml-0.18.1 for the full dataset.

Application	**Metric**	**a_pow**	**m_log**	**s_log**	**a_str**	**b_str**	Q_{pow}	Q_{log}	Q_{str}	k
eclipse	AC	1.82	0.80	0.82	0.58	1.03	3685.96	44.52	701.82	41
eclipse	PC	1.57	1.24	0.79	0.44	0.76	10613.55	214.25	3470.69	118
eclipse	IC	1.91	-0.06	1.12	0.70	1.01	65.96	33.83	64.41	117
eclipse	MS	1.11	2.53	1.22	0.18	0.38	286047.39	13143.66	91823.29	4172
5jre	nF	1.47	0.90	1.29	0.42	1.03	933.12	113.72	229.11	427
jre	nM	1.26	1.54	1.25	0.31	0.92	2374.36	218.22	1084.17	257
jre	nC	2.74	-0.06	0.70	1.16	1.00	340.55	68.30	243.58	14
jre	SP	1.84	-0.03	1.10	0.72	1.01	91.50	46.63	61.28	353
jre	IC	1.86	0.10	0.99	0.73	1.02	74.64	37.64	40.27	451
netbeans	AP	2.13	-0.15	0.95	0.87	0.96	44.61	105.89	184.63	508
netbeans	IP	3.14	-0.02	0.50	1.63	1.00	109.44	11.08	45.33	7
netbeans	PP	1.85	-0.25	1.35	0.58	0.93	93.60	204.40	308.58	618
tomcat	MS	0.89	2.45	1.25	0.00	0.51	10318.07	4334.30	7020.94	1634
tomcat (5% cut)	MS	1.68	1.65	1.75	0.29	0.96	320.63	569.26	285.78	562
openoffice	IP	3.74	0.43	9.15	2.26	0.95	133.66	19713.79	21.95	8
compiere	RC	1.20	1.20	0.62	0.30	0.37	3113.16	457.41	923.42	18

Table 2 Fitted parameters for applications and metrics shown in plots.

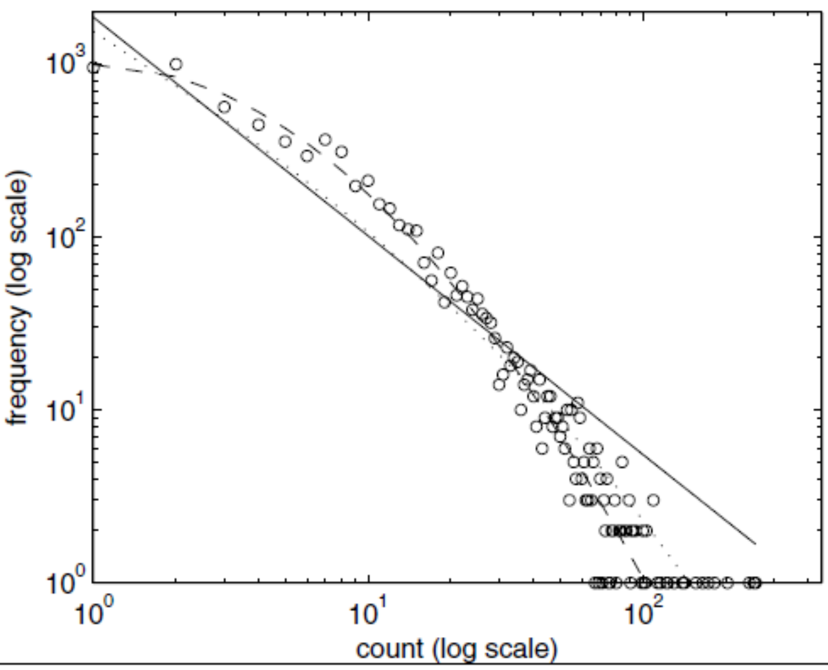


Figure 7 nM distribution and fitted curves for JRE.

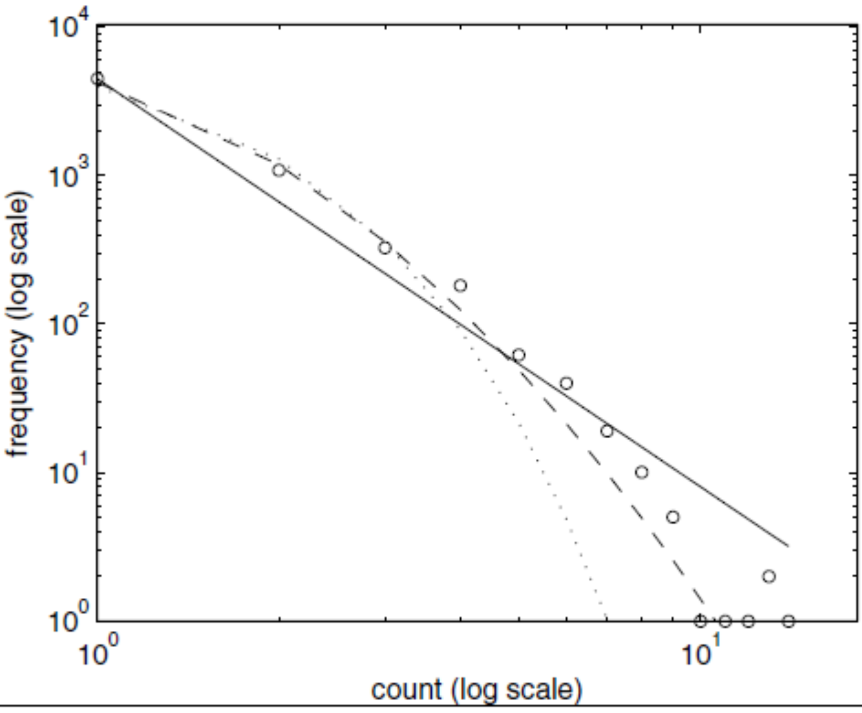


Figure 8 nC distribution and fitted curves for JRE.

	AC	AP	DO	DOinv	IC	IP	MS	PC	PP	PubMC	RC	RP	SP	nC	nF	nM	pkgSize
jeppers	*b*	x	*b*	d	x	x	*b*	*b*	d	*b*	*b*	d	x	*b*	*b*	*b*	*b*
fitjava	*b*	*b*	c	*b*	x	x	c	*b*	*b*	*b*	*b*	*b*	*b*	d	*b*	*b*	d
junit	*b*	c	*b*	*b*	c	x	d	*b*	*b*	*b*	*b*	*b*	c	*b*	*b*	*b*	*b*
jgraph	*b*	*b*	*b*	*b*	*b*	d	d	*b*	*b*	*b*	*b*	*b*	d	*b*	*b*	*b*	*b*
jparse	*b*	*b*	c	d	*b*	d	d	d	*b*	*b*	d	*b*	*b*	*b*	*b*	c	*b*
jaga	*b*	*b*	d	*b*	*b*	x	d	*b*	*b*	c	c	*b*	*b*	c	*b*	c	*b*
joggplayer	*b*	*b*	*b*	*b*	x	x	a	c	*b*	*b*	c	d	d	*b*	*b*	*b*	*b*
fitlibraryforfitnesse	c	c	c	*b*	d	d	d	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*
javacc	*b*	*b*	*b*	d	*b*	d	d	d	*b*	*b*	*b*	*b*	*b*	c	*b*	*b*	*b*
lucene	d	*b*	c	d	d	x	d	d	*b*	*b*	d	*b*	*b*	*b*	*b*	*b*	*b*
rssowl	*b*	*b*	c	*b*	d	d	d	c	*b*	*b*	c	*b*	*b*	c	*b*	*b*	*b*
sablecc	*b*	*b*	d	d	*b*	x	d	d	d	d	d	*b*	*b*	d	*b*	c	*b*
jag	c	*b*	*b*	c	*b*	x	d	c	*b*	*b*	c	*b*	*b*	c	*b*	*b*	*b*
antlr	c	*b*	c	*b*	*b*	*b*	d	*b*	*b*	*b*	d	*b*	*b*	*b*	*b*	*b*	*b*
jrefactory	*b*	*b*	d	d	c	d	d	d	d	*b*	c	*b*	*b*	d	*b*	*b*	*b*
hsqldb	*b*	*b*	*b*	d	*b*	x	d	d	*b*	*b*	c	*b*	*b*	a	*b*	*b*	*b*
jedit	c	*b*	c	c	*b*	d	d	c	*b*	*b*	c	*b*	*b*	a	*b*	*b*	*b*
axion	c	*b*	c	c	*b*	d	d	d	*b*	*b*	d	*b*	*b*	a	*b*	*b*	*b*
galleon	d	*b*	*b*	*b*	*b*	c	d	c	*b*	*b*	c	*b*	*b*	d	c	*b*	*b*
james	*b*	*b*	*b*	*b*	*b*	d	d	c	*b*	*b*	d	*b*	*b*	d	*b*	*b*	*b*
colt	c	*b*	d	*b*	*b*	d	d	c	*b*	*b*	d	*b*	*b*	d	*b*	*b*	*b*
aglets	d	*b*	c	d	d	*b*	d	c	*b*	*b*	c	*b*	*b*	c	*b*	*b*	*b*
jhotdraw	c	*b*	c	c	*b*	*b*	d	c	*b*	*b*	c	*b*	*b*	c	c	*b*	*b*
ganttproject	c	*b*	c	d	*b*	*b*	d	c	*b*	*b*	c	*b*	*b*	c	*b*	c	*b*
jetty	c	*b*	d	c	*b*	*b*	d	c	*b*	*b*	d	*b*	*b*	c	*b*	*b*	*b*
ireport	*b*	*b*	c	d	*b*	d	d	c	*b*	*b*	d	*b*	*b*	c	*b*	*b*	*b*
jext	*b*	*b*	d	c	*b*	x	d	*b*	*b*	*b*	*b*	*b*	*b*	d	*b*	*b*	*b*
pmd	*b*	*b*	d	c	*b*	d	d	d	d	*b*	c	*b*	*b*	c	*b*	*b*	*b*
aoi	c	*b*	c	d	*b*	d	d	c	*b*	d	d	*b*	*b*	c	c	c	*b*
jung	c	*b*	c	c	*b*	c	d	d	*b*	c	c	*b*	*b*	c	*b*	*b*	*b*
megamek	c	*b*	c	c	*b*	d	d	d	*b*	*b*	d	*b*	*b*	*b*	*b*	*b*	*b*
jfreechart	c	*b*	c	c	*b*	*b*	d	c	*b*	*b*	c	*b*	*b*	c	c	*b*	*b*
poi	c	*b*	d	d	*b*	*b*	d	d	*b*	c	d	c	*b*	d	c	d	*b*
jmeter	d	*b*	c	c	*b*	*b*	d	d	*b*	c	c	*b*	*b*	c	c	c	*b*
glassfish	a	*b*	c	d	*b*	*b*	d	c	*b*	*b*	c	*b*	*b*	*b*	*b*	c	*b*
jchempaint	c	*b*	c	c	*b*	d	d	c	*b*	d	d	*b*	*b*	d	d	d	*b*
jasperreports	*b*	*b*	d	d	c	c	d	c	*b*	*b*	d	*b*	*b*	c	*b*	*b*	*b*
scala	*b*	*b*	c	c	*b*	d	d	c	*b*	d	d	*b*	*b*	c	*b*	d	*b*
drjava	*b*	*b*	d	d	c	c	d	d	d	d	d	*b*	d	d	*b*	d	*b*
ant	c	*b*	c	c	*b*	c	d	c	*b*	c	d	*b*	*b*	d	c	c	*b*
sandmark	c	*b*	c	d	*b*	*b*	d	d	*b*	c	d	*b*	*b*	d	c	c	*b*
tomcat	c	*b*	c	c	*b*	*b*	d	d	*b*	*b*	d	*b*	*b*	d	c	*b*	*b*
hibernate	c	*b*	c	d	*b*	c	d	d	*b*	c	d	*b*	c	d	c	c	*b*
sequoiaerp	c	*b*	d	c	*b*	d	d	c	*b*	c	d	*b*	*b*	d	c	c	*b*
columba	c	*b*	c	c	*b*	c	d	d	*b*	c	c	*b*	*b*	d	c	c	*b*
argouml	c	a	d	d	*b*	c	d	c	*b*	c	d	d	*b*	c	c	c	*b*
compiere	c	*b*	d	d	*b*	d	d	d	*b*	d	d	*b*	c	c	*b*	d	*b*
derby	c	*b*	c	d	*b*	c	d	c	*b*	c	d	*b*	*b*	d	c	c	*b*
azureus	c	*b*	c	d	*b*	d	d	d	*b*	c	c	c	*b*	c	c	c	c
geronimo	d	*b*	d	c	a	c	d	d	d	d	d	d	*b*	d	c	d	*b*
jtopen	d	*b*	d	d	*b*	d	d	d	*b*	c	d	*b*	*b*	d	c	c	*b*
openoffice	c	*b*	d	c	*b*	d	d	c	*b*	d	c	*b*	*b*	d	d	d	*b*
jboss	c	*b*	d	d	*b*	c	d	d	c	c	d	c	*b*	d	c	c	d
jre	c	*b*	d	d	c	d	d	d	*b*	d	d	*b*	c	d	c	c	*b*
netbeans	d	a	d	d	*b*	c	d	d	a	c	d	c	*b*	d	c	c	c
eclipse	c	a	d	d	*b*	d	d	d	a	c	d	*b*	c	d	c	c	c

Table 3 Quality of fit at Confidence Interval 80% for full dataset: a–good fit only to power-law, b–good fits to more than one curve, c–good fit only to other curves, d–no good fits. Applications are ordered by increasing size (number of classes).

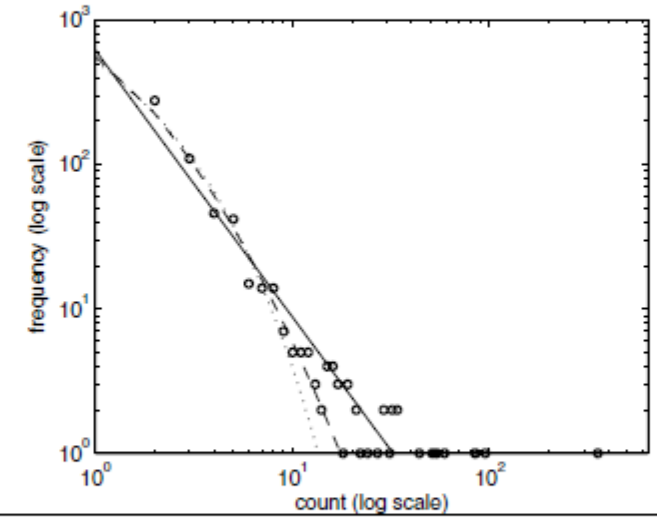


Figure 9 SP distribution and fitted curves for JRE.

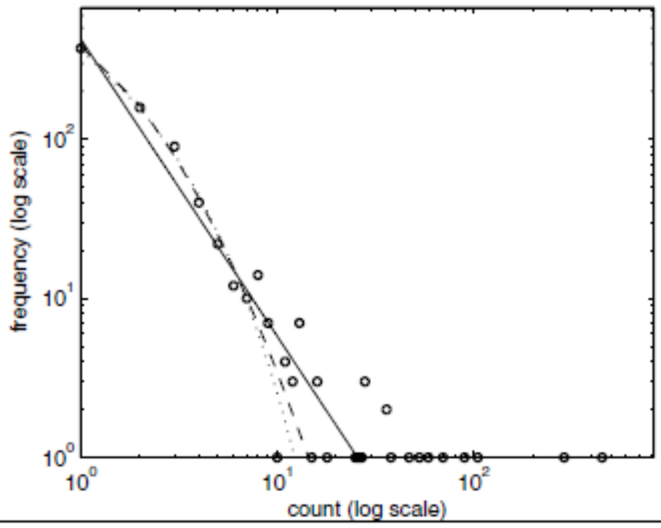


Figure 10 IC distribution and fitted curves for JRE.

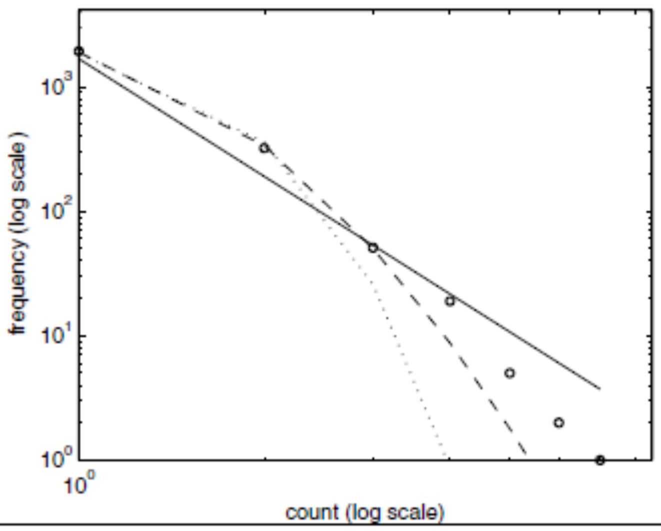


Figure 11 IP distribution and fitted curves for NetBeans.

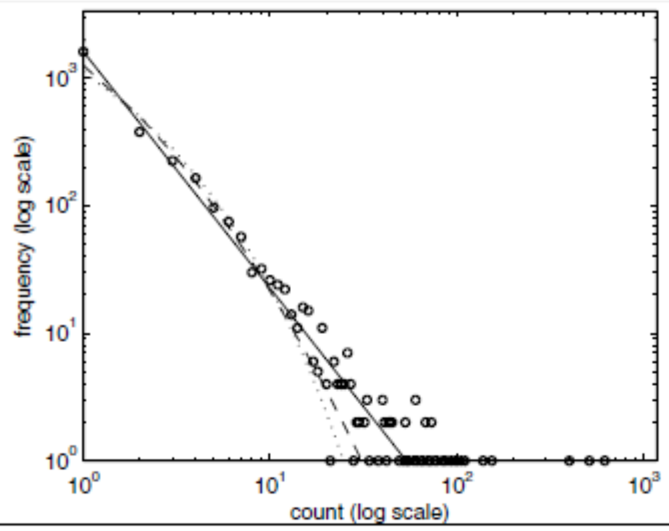


Figure 12 PP distribution and fitted curves for NetBeans.

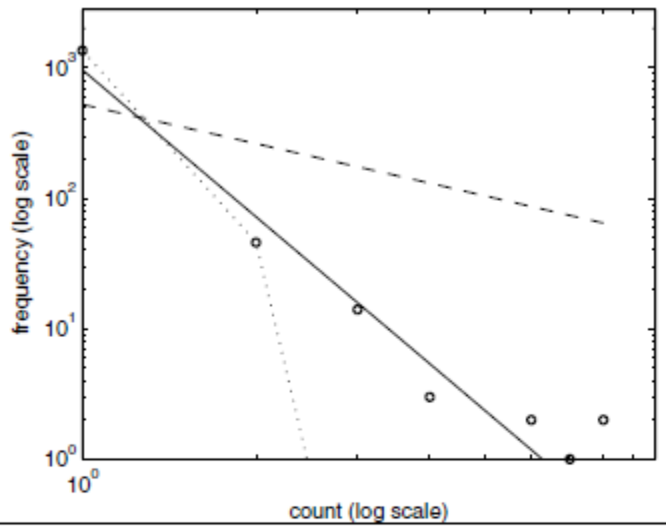


Figure 13 IP distribution and fitted curves for Openoffice.

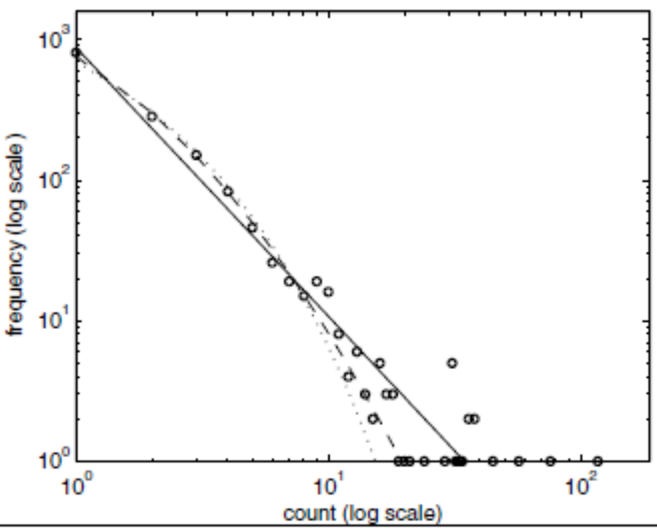


Figure 14 IC distribution and fitted curves for Eclipse.

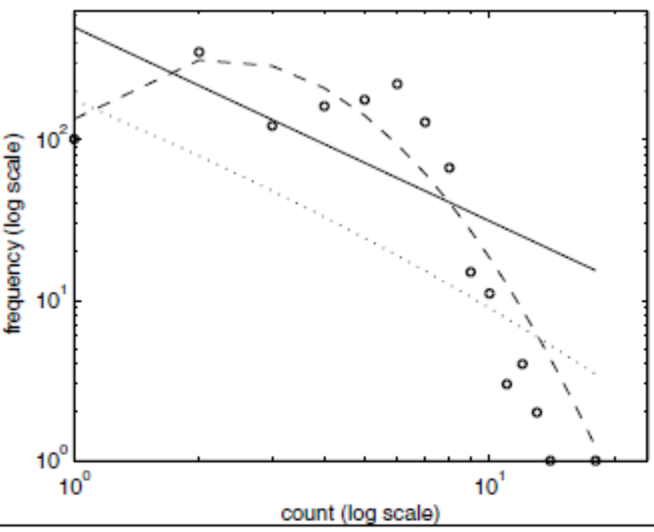


Figure 15 RC distribution and fitted curves for Compiere.

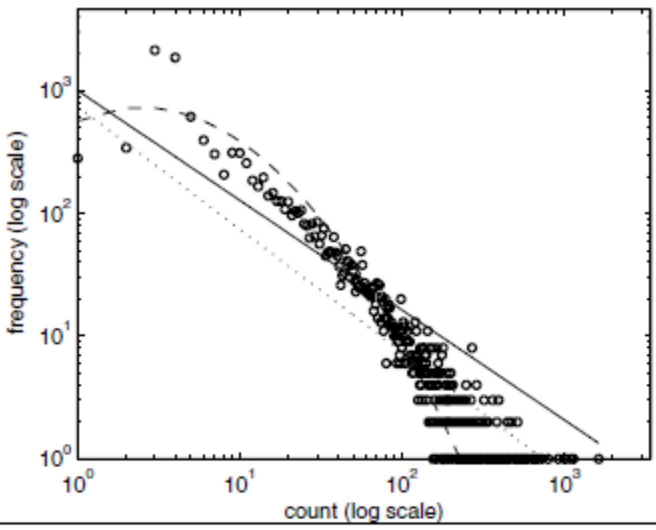


Figure 16 MS distribution and fitted curves for Tomcat.

Table 3 shows these results for the 80% CI and using complete datasets (0% cuts). In this table, the applications are ordered in increasing size, as measured by number of classes. The four groups are: applications with fewer than 200 classes, applications with fewer than 500 classes, applications with fewer than 1000 classes, and those with more than 1000 classes. To aid comprehension, we use different typefaces for the entries.

For the moment, we will just note patterns and trends, and leave interpretation and discussion to the next section. The first thing to note (other than the sheer size), is that, while all values are represented, *b* (multiple distributions have good fits) is quite prominent. The next point is that *a* (good fit only to power-law) is relatively rare.

Looking at individual metrics for the larger applications (last category), we note that AC, PC, and RC tend to have *c* and *d*, indicating lack of support for them having a power-law distribution, whereas their opposites, AP, PP, and RP, as well as SP, tend to have *a* and *b*. In almost all cases, however, there are exceptions for individual applications. IC and IP show the opposite trend, with IC having mainly *a* and *b* and IP having mainly *c* and *d*.

It must be kept in mind that Table 3 represents only 5% of the results of the curve fitting (which itself represents a summarization of the original data)—there are the other CIs and cuts. What the results show for the other cuts and CIs is what one would expect. As the cut size increases, meaning the highest and lowest frequency data (where most of the variation occurs) is removed, we get better fits for all three distributions (that is, tending toward *b*). Similarly, as the CI is increased, it also becomes easier to get a good fit.

We chose to show the 80% CI as it seemed the most representative. The 60% CI is not that different from what is shown in Table 3, and all of the differences are what one would expect —more *d*'s (no good fits) at 60% than at 80% or tending toward *b* when going from 60% to 80%.

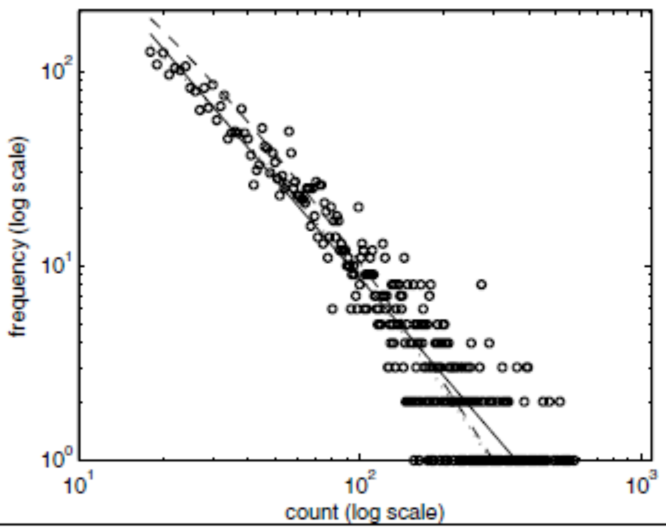


Figure 17 MS distribution and fitted curves for Tomcat after a 5% cut.

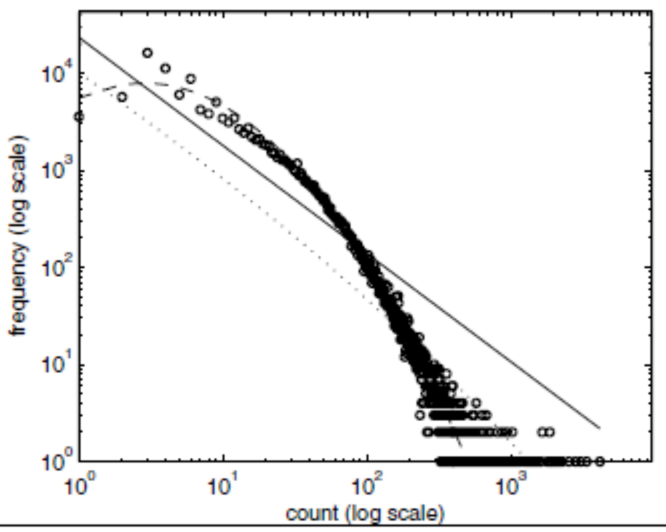


Figure 18 MS distribution and fitted curves for Eclipse.

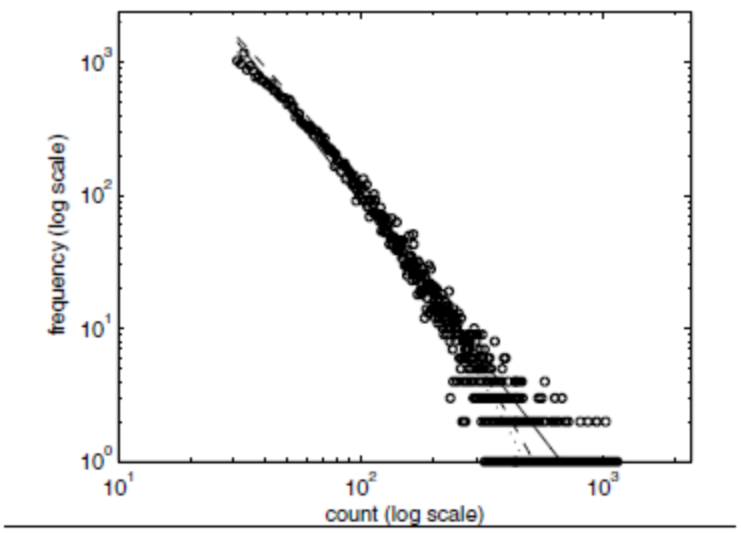


Figure 19 MS distribution and fitted curves for Eclipse after a 5% cut.

To finish this section, we show a few more fitted curves. In this case, Figures 16-19, we show various MS distributions. These are interesting as they have many more data points than the others, being based on methods not types. We also show the effect of applying a 5% cut.

3.5 Discussion

3.5.1 Interpretation

Recall that several of our metrics measure 5 inter-type relationships—Inheritance (SP), Aggregation (AC and AP), Parameter (PC and PP), Return (RC and RP), and Interface (IC and IP). The 'C' variant of the metric for a relationship measures the 'client' end and 'P' the 'provider' end. Or, if the code were represented as a directed graph with types as vertices and the different relationships as edges, then 'C' would be the out-degree and 'P' the in-degree for each relationship of each vertex. We note that out-degree is impacted by decisions made with respect to the type represented by the vertex, whereas in-degree is the result of decisions made with respect to other types.

In the previous section, we noted that AC, PC, and RC distributions tended not to have good fits to a power-law, but AP, PP, RP, and SP did. From the comments above, this suggests out-degree distributions are not power-laws but in-degree are.

The distributions we are seeing for the ‘C’ metrics tend to be truncated at the high-value (low-frequency) end. A person changing the code for a class is inherently aware of its outward dependencies (e.g. the number of types it uses or the number of interfaces it implements), but they are not inherently aware of the number of classes that subtype it or call methods on it. They therefore have less control over the latter than they do over the former. Furthermore, we believe there is a tendency is to avoid (consciously or subconsciously) ‘big things’, whether due to difficulty of management (e.g., methods with many parameters) or simply through training (“Don’t write big classes!”). This suggests that ‘C’ relationships are more likely than ‘P’ relationships to have ‘truncated’ curves. We can generalize this to hypothesise that any metric that measures something that the programmer is inherently aware of will tend to have a ‘truncated’ curve, that is, not be a power-law.

The nF, nM, and PubMC, distributions are explained by our hypothesis. They are all aspects of a type description that the developer is inherently aware of, and all tend not to have support for power-laws.

Unfortunately our hypothesis does not explain the IC and IP distributions. We believe that the main cause of the poor fits for the IP distributions is the small datasets (no more than 11 data points, and see for example Figure 13). This, however, does not explain IC (e.g., Figures 10 and 14). nC also suffers from having small datasets, which might explain the results we see. DO and DOinv are related—DO is the ‘client’ end, and DOinv the ‘provider’. However in this case there is not a strong distinction between the two, both being *c* and *d*. The DO relationship is effectively including all of AC, PC, RC, and IC, as well as types used for local variables. This would mean that the behaviour of IC noted above would oppose the behaviour of the others, which may explain the results. We do know that types used for local variables (or rather, not used in the published interface) do account for significant dependency structures [MT07b].

MS, with few exceptions (all small applications), does not fit any distribution at the 80 CI. However, at 90 CI and above, there are good fits to all of them. Our hypothesis would suggest this should be a truncated curve (the size of the method being a decision made as it is written) but it would seem that there is too much noise to be sure.

There is another important point to make. There is quite noticeable variation on the degree of fit between different applications. This raises an interesting question: if a given relationship (metric) does follow a particular distribution, why do we not see this distribution for all applications, how is it that this variation exists?

Two answers spring to mind. The first is that different applications come from different domains, and it is possible that different domains have different distributions. For example, NetBeans and OpenOffice often have different values (usually *c* vs *d* or *a* vs *d*). NetBeans is an IDE, whereas OpenOffice is an office suite, and in fact is really several applications wrapped as one. We picked these two because they were both originally Sun products. That said, Compiere is ERP and seems somewhat different in nature than, for example, Openoffice, and yet the distributions seem mainly similar.

Another answer is that there is another thing that is potentially quite different (and much harder to see) between the applications—their design. If we are seeing different distributions due to different designs, if we could understand how aspects of the design related to the kind of distribution exhibited, there is the potential for developing a *quantitative measure* for design quality. Having such a measure could have tremendous impact on how software is developed in the future.

Of course before this can happen, we must understand (presuming such a relationship exists) *which* distribution corresponds to a good design and which does not. It is not obvious that, for example, the power-law distribution is found in 'good' designs—it could just as easily be the opposite! Our results do not provide much advice either way. This does, however, suggest an extremely interesting avenue for future research.

3.5.2 Threats to Validity

The most likely threat to the validity of our conclusions is the corpus we used. It consists entirely of open-source applications of small to medium size. Some applications originated from commercial organisations, but it is not obvious that the

IBM and Sun-donated code is typical of closed-source code. Other studies have suggested there is little difference between open-source and closed-source software [MT07b], but we cannot say whether or not this is true here. While we cannot claim that our corpus represents a random sample of Java software, our situation is no different than corpora used in applied linguistics. Hunston describes a number of ways corpora may be reasonably used [Hun02]. Our corpus is what she describes as a reference corpus, which are often used as base-line for further studies. Thus, a random sample is not necessary in order to produce an valid result. Our results hold for what is in our corpus: whether or not they hold for other collections will in itself be of interest.

So we cannot say for sure how representative our corpus is of Java software in general, or even open-source software in particular. Nevertheless, the commonality we have seen across all of the applications we analyse gives us confidence that our conclusions will hold generally.

A similar issue is that our corpus consists only of Java applications. It is possible we may see different distributions when looking at other languages such as C# or C++. While there appears nothing obviously different between Java and languages such as C# or C++ with respect to our study, they do share the property of having static type checking, so while we may see no differences for such languages, we may see differences in languages, such as Smalltalk, that do not have static type checking.

A property of the software we have studied that we have not addressed in our study is the manner in which the software was created. Our hypothesis is based on the lack of global view a developer has of the application being developed. Recently, there has been a significant increase in the use of sophisticated Integrated Development Environments (IDE) such as Eclipse, and one characteristic of these IDEs is that they provide a better view of the source code than has been available in the past. The use of such IDEs may affect the shape of the distributions we have been investigating. We believe most of the code in our corpus was written before the advent of such IDEs, but some of the variation we see may be due to how the code was written. Again Smalltalk may show differences as it has always had an IDE.

As noted earlier, because we measure from byte code, there is some information from the source code not available to us. The circumstances for which this is the case seem to be such that this will be rare.

3.6 Related Work

As with many other things, Knuth was one of the first to carry out empirical studies to understand what code that is actually written looks like [Knu71]. He presented a static analysis of over 400 FORTRAN programmes and dynamic analysis of about 25 programs. His main motivation was compiler design, with the concern that compilers may not optimise for the typical case as no-one knew what the typical case was. His analysis was at the statement level, counting such things as the number of occurrences of an IF statement, or the number of executions of a given statement.

Collberg et al. have carried out a study of 1132 Java programs[CMS04]. These were gathered by searching for jar files with Google and removing any that were invalid. Their main goal was the development of tools for protection of software from piracy, tampering, and reverse engineering. Like Knuth, they argued that their tools could benefit by knowing the typical and extreme values of various aspects of software. Consequently, their interest is in the low-level details of the code with a view toward future tool support or language design.

Although their interest is in low-level details, Collberg et al. do gather a number of similar statistics to ours, such as number of classes per package, number of fields per class, number of methods per class, size of the constant pool, and so on. However comparison with their results is problematic, as they appear to include all classes referred to in an application, whereas we only consider classes that appear in the application source.

Gil and Maman analysed a corpus of 14 Java applications for the presence of *micro patterns*, patterns at the code level that represent low-level design choices [GM05]. They found that 3 out of 4 classes matched one of the 27 micro patterns in their catalogue, and just over half of the classes are catalogued by just 5 patterns. This is a form of structural analysis, however it focuses on individual classes, rather than at the application level as we have done.

As already mentioned, Wheeldon and Counsell have performed a similar analysis to ours. They looked at JDK 1.4.2, Ant 1.5.3, and Tomcat 4.0. They computed the 12 metrics as noted in section 3 and concluded that what they were seeing were power-laws. There are some differences between their work and ours. Most notably is how the metrics were computed. Wheeldon and Counsell used a custom doclet to extract the relevant information, which limited them to just the information available from the Javadoc comments. Also, they were not specific as to what choices they made for the variables discussed in section 3.

We believe the inconsistency between Wheeldon and Counsell's conclusions and ours is due to our more extensive corpus. Our original intention was to reproduce their study and, we thought, results. The 'truncated-curve' distribution only really became apparent in the repetition across multiple applications. In fact, their figure 2(b) appears to have something of a curve to it. Our work does, however, add significant evidence to support their hypothesis that there are regularities that are common across all non-trivial Java programs.

3.7 Conclusion

We have studied the hypothesis that the distribution of a number of metrics on object-oriented software obey a power-law. We did so over a larger sample size than has been considered by past similar studies, and applied analysis techniques to characterise how closely each distribution obeyed a power-law. We have presented our method and analysis in what we hope is sufficient detail to allow our studies to be reproduced with confidence.

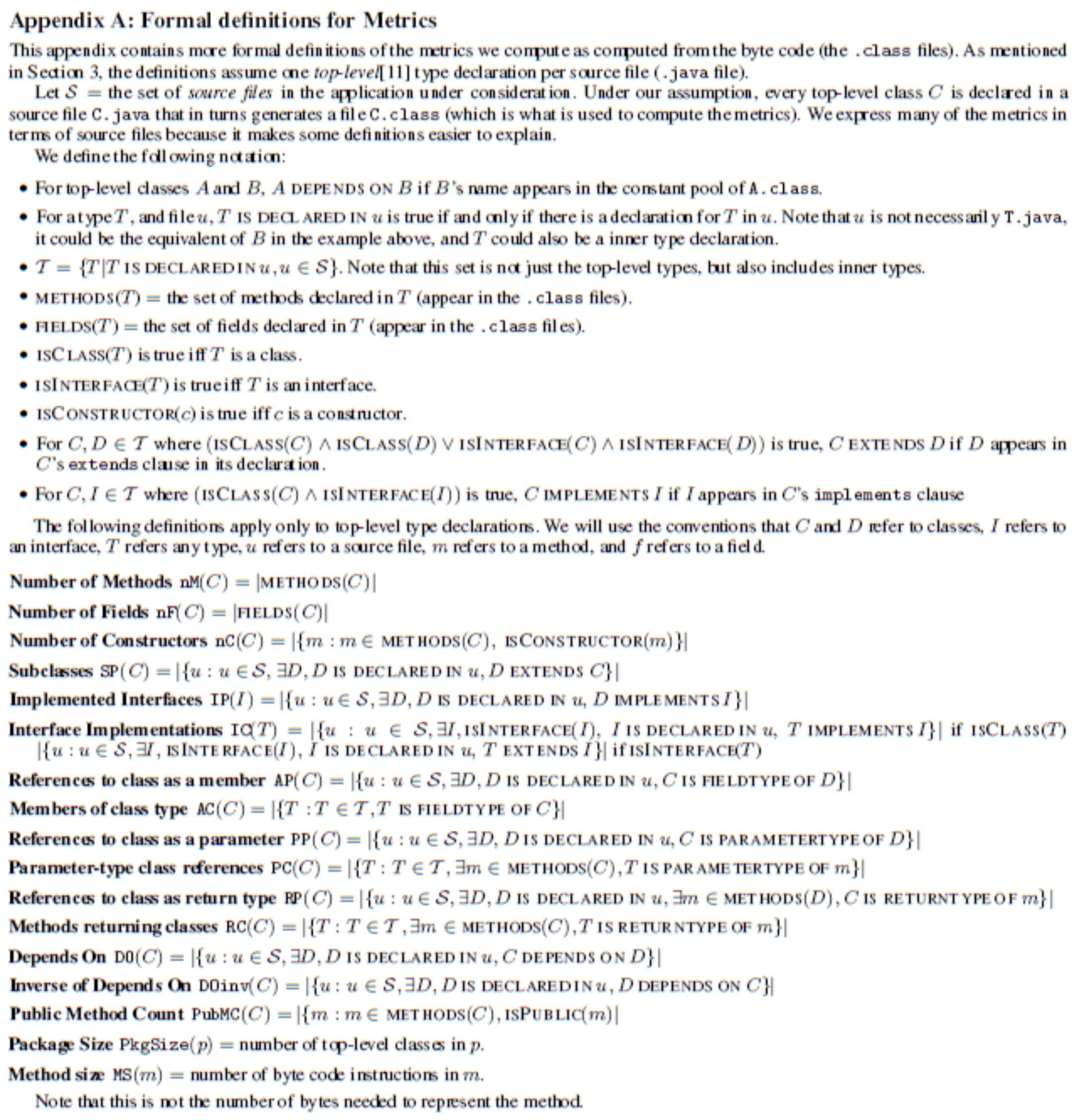
What we found was that while there were distributions for which there was good evidence for a power-law, there are a number for which there was little evidence that a power-law exists. This is in contrast with what earlier studies have suggested. We hypothesise that any metric that measures a relationship that the programmer is inherently aware of will tend to have a 'truncated' curve, that is, not be a power-law.

Of particular interest is the fact that some applications frequently differed for some metrics from the other applications, indicating that *some attribute of the application's code can affect the resulting distribution*. This finding has potentially tremendous implications. If the distribution does depend on either design quality or domain, then knowing the distribution of a 'good' design would provide a much sounder foundation for developing software than currently exists. As open-source applications make extensive use of version control and bug-tracking systems, we believe the data necessary for such studies as correlations between distribution and prevalence of defects will be possible.

There remains much work to be done. Further studies are needed to determine how representative our findings are. This means expanding the studies to other (especially larger) applications, to applications developed in other environments, such as closed-source, to other domains (for example, real-time software is not represented in our corpus at the moment), and to other languages.

We need to be able to explain why we see some distributions in some applications for some metrics and not others. For example, we need models that explain how these distributions arise. In the case of power-law distributions, there is no theory to explain why we should see such scale-free structures in software. Two main hypothetical mechanisms have been put forward [Bar02] to account for the origin of scale-free network structure in other domains: growth with preferential attachment[BA99], in which existing nodes link to new nodes with probability proportional to the number of links they already have, and hierarchical growth [Wei85] in which networks grow in an explicitly self-similar fashion. Additionally arguments from optimal design have been proposed[VCS02][SFCMV02]. It is still far from clear, however, what (if any) fundamental theory might account for the ubiquity of the phenomenon in software.

Ultimately, we need to understand the relationship between large-scale structures found in software, and quality attributes such as understandability, modifiability, testability, and reusability. We believe this study is an important step toward that goal.



Appendix A: Formal definitions for Metrics

This appendix contains more formal definitions of the metrics we compute as computed from the byte code (the `.class` files). As mentioned in Section 3, the definitions assume one *top-level*[11] type declaration per source file (`.java` file).

Let $\mathcal{S}$ = the set of *source files* in the application under consideration. Under our assumption, every top-level class C is declared in a source file `C.java` that in turns generates a file `C.class` (which is what is used to compute the metrics). We express many of the metrics in terms of source files because it makes some definitions easier to explain.

We define the following notation:

- For top-level classes A and B, A $\textsc{depends on}$ B if B's name appears in the constant pool of `A.class`.
- For a type T, and file u, T $\textsc{is declared in}$ u is true if and only if there is a declaration for T in u. Note that u is not necessarily `T.java`, it could be the equivalent of B in the example above, and T could also be a inner type declaration.
- $\mathcal{T} = \{T | T \textsc{ is declared in } u, u \in \mathcal{S}\}$. Note that this set is not just the top-level types, but also includes inner types.
- $\textsc{methods}(T)$ = the set of methods declared in T (appear in the `.class` files).
- $\textsc{fields}(T)$ = the set of fields declared in T (appear in the `.class` files).
- $\textsc{isClass}(T)$ is true iff T is a class.
- $\textsc{isInterface}(T)$ is true iff T is an interface.
- $\textsc{isConstructor}(c)$ is true iff c is a constructor.
- For $C, D \in \mathcal{T}$ where $(\textsc{isClass}(C) \wedge \textsc{isClass}(D) \vee \textsc{isInterface}(C) \wedge \textsc{isInterface}(D))$ is true, C $\textsc{extends}$ D if D appears in C's `extends` clause in its declaration.
- For $C, I \in \mathcal{T}$ where $(\textsc{isClass}(C) \wedge \textsc{isInterface}(I))$ is true, C $\textsc{implements}$ I if I appears in C's `implements` clause

The following definitions apply only to top-level type declarations. We will use the conventions that C and D refer to classes, I refers to an interface, T refers any type, u refers to a source file, m refers to a method, and f refers to a field.

Number of Methods $\texttt{nM}(C) = |\textsc{methods}(C)|$

Number of Fields $\texttt{nF}(C) = |\textsc{fields}(C)|$

Number of Constructors $\texttt{nC}(C) = |\{m : m \in \textsc{methods}(C),\ \textsc{isConstructor}(m)\}|$

Subclasses $\texttt{SP}(C) = |\{u : u \in \mathcal{S}, \exists D, D \textsc{ is declared in } u, D \textsc{ extends } C\}|$

Implemented Interfaces $\texttt{IP}(I) = |\{u : u \in \mathcal{S}, \exists D, D \textsc{ is declared in } u, D \textsc{ implements } I\}|$

Interface Implementations $\texttt{IC}(T) = |\{u : u \in \mathcal{S}, \exists I, \textsc{isInterface}(I), I \textsc{ is declared in } u, T \textsc{ implements } I\}|$ if $\textsc{isClass}(T)$ $|\{u : u \in \mathcal{S}, \exists I, \textsc{isInterface}(I), I \textsc{ is declared in } u, T \textsc{ extends } I\}|$ if $\textsc{isInterface}(T)$

References to class as a member $\texttt{AP}(C) = |\{u : u \in \mathcal{S}, \exists D, D \textsc{ is declared in } u, C \textsc{ is fieldtype of } D\}|$

Members of class type $\texttt{AC}(C) = |\{T : T \in \mathcal{T}, T \textsc{ is fieldtype of } C\}|$

References to class as a parameter $\texttt{PP}(C) = |\{u : u \in \mathcal{S}, \exists D, D \textsc{ is declared in } u, C \textsc{ is parametertype of } D\}|$

Parameter-type class references $\texttt{PC}(C) = |\{T : T \in \mathcal{T}, \exists m \in \textsc{methods}(C), T \textsc{ is parametertype of } m\}|$

References to class as return type $\texttt{RP}(C) = |\{u : u \in \mathcal{S}, \exists D, D \textsc{ is declared in } u, \exists m \in \textsc{methods}(D), C \textsc{ is returntype of } m\}|$

Methods returning classes $\texttt{RC}(C) = |\{T : T \in \mathcal{T}, \exists m \in \textsc{methods}(C), T \textsc{ is returntype of } m\}|$

Depends On $\texttt{DO}(C) = |\{u : u \in \mathcal{S}, \exists D, D \textsc{ is declared in } u, C \textsc{ depends on } D\}|$

Inverse of Depends On $\texttt{DOinv}(C) = |\{u : u \in \mathcal{S}, \exists D, D \textsc{ is declared in } u, D \textsc{ depends on } C\}|$

Public Method Count $\texttt{PubMC}(C) = |\{m : m \in \textsc{methods}(C), \textsc{isPublic}(m)|$

Package Size $\texttt{PkgSize}(p)$ = number of top-level classes in p.

Method size $\texttt{MS}(m)$ = number of byte code instructions in m.

Note that this is not the number of bytes needed to represent the method.

Appendix B: Corpus details

This appendix provides the details of the part of the corpus used in this study.We use the standard naming scheme for each application, which typically includes some kind of version identification. The domain comes from our assessment based on the application documentation. We identify where we acquired the source code. The column "O/C" refers to whether the application can be considered open or closed source (all applications used here are open source). The column "V" identifies where we have multiple versions (we only used the latest version in this study). Finally, any notes that seem relevant are provided.

Application	*#*	*Domain*	*Origin*	*O/C*	*V*	*Notes*
aglets-2.0.2	280	Framework for developing mobile agents	Sourceforge	O	N	Donated by IBM
ant-1.6.5	700	Java build tool	Apache	O	Y	
antlr-2.7.5	209	Parser generator	`antlr.org`	O	N	
aoi-2.2	415	3D modelling and rendering	Sourceforge	O	N	
argouml-0.18.1	1251	UML drawing/critic	`tigris.org`	O	Y	
axion-1.0-M2	237	SQL database	`tigris.org`	O	N	
azureus-2.3.0.4	1650	P2P filesharing	Sourceforge	O	Y	
colt-1.2.0	269	High performance collections library	`hoschek.home.cern.ch`	O	Y	
columba-1.0	1180	Email client	Sourceforge	O	N	
compiere-251e	1372	ERP and CRM	Sourceforge	O	N	
derby-10.1.1.0	1386	SQL database	Apache Jakarta	O	N	Donated by IBM
drjava-20050814	668	IDE	Sourceforge	O	N	
eclipse-SDK-3.1-win32	11413	IDE	`www.eclipse.org`	O	Y	Donated by IBM
fitjava-1.1	37	Automated testing	`fit.c2.com`	O	N	
fitlibraryforfitnesse-20050923	124	Automated testing	Sourceforge	O	N	
galleon-1.8.0	243	TiVo media server	Sourceforge	O	N	
ganttproject-1.11.1	310	Gantt chart drawing	Sourceforge	O	N	
geronimo-1.0-M5	1719	J2EE server	Apache	O	N	
glassfish-9.0-b15	582	J2EE server	`dev.java.net`	O	N	
hibernate-3.1-rc2	902	Persistence object mapper	Sourceforge	O	N	
hsqldb-1.8.0.2	217	SQL database	Sourceforge	O	N	
ireport-0.5.2	347	Visual report design for JasperReports	Sourceforge	O	N	
jag-5.0.1	208	J2EE application generator	Sourceforge	O	N	
jaga-1.0.b	100	API for genetic algorithms	`jaga.org`	O	N	
james-2.2.0	259	Enterprise mail server	Apache	O	N	
jasperreports-1.1.0	633	Reporting tool	Sourceforge	O	N	
javacc-3.2	125	Parser generator	`dev.java.net`	O	N	
jboss-4.0.3-SP1	4143	J2EE server	Sourceforge	O	N	

Continued on next page

Application	*#Classes*	*Domain*	*Origin*	*O/C*	*V*	*Notes*
jchempaint-2.0.12	612	Editor for 2D molecular structures	Sourceforge	O	N	
jedit-4.2	234	Text editor	Sourceforge	O	N	
jeppers-20050607	20	Spreadsheet editor	Sourceforge	O	N	
jetty-5.1.8	327	HTTP Server/servlet container	Sourceforge	O	N	
jext-5.0	353	IDE	Sourceforge	O	N	
jfreechart-1.0.0-rc1	469	Chart drawing	Sourceforge	O	N	
jgraph-5.7.4.3	50	Graph drawing	Sourceforge	O	Y	
jhotdraw-6.0.1	300	Graph drawing	Sourceforge	O	Y	
jmeter-2.1.1	560	Java performance testing	Apache	O	Y	
joggplayer-1.1.4s	114	MP3 player	`joggplayer.webarts.bc.ca`	O	N	
jparse-0.96	69	Java compiler front-end	`www.ittc.ku.edu/JParse`	O	N	
jre-1.4.2.04	7257	JRE	sun.com	O	N	
jrefactory-2.9.19	211	Refactoring tool for Java	Sourceforge	O	N	
jtopen-4.9	2857	Java toolbox for iSeries and AS/400 servers	Sourceforge	O	N	Donated by IBM
jung-1.7.1	454	Graph drawing	Sourceforge	O	Y	
junit-3.8.1	48	Unit testing framework	Sourceforge	O	N	
lucene-1.4.3	170	Text indexing	Apache	O	Y	
megamek-2005.10.11	455	Game	Sourceforge	O	N	
netbeans-4.1	8406	IDE	netbeans.org	O	Y	Donated By Sun
openoffice-2.0.0	2925	Office suite	openoffice.org	O	N	Donated By Sun
pmd-3.3	375	Java code analyser	Sourceforge	O	N	
poi-2.5.1	480	API to access Microsoft format files	Apache	O	N	
rssowl-1.2	189	RSS Reader	Sourceforge	O	N	
sablecc-3.1	199	Compiler/ Interpreter generating framework	Sourceforge	O	N	
sandmark-3.4	837	Software watermarking/security	`www.cs.arizona.edu/sandmark`	O	N	
scala-1.4.0.3	654	Multi-paradigm programming language	`scala.epfl.ch`	O	N	
sequoiaerp-0.8.2-RC1-all-platforms	936	ERP and CRM	Sourceforge	O	N	
tomcat-5.0.28	892	Servlet container	Apache	O	N	

Chapter 4 On the Usage and Usefulness of OO Design Principles

4.1 Introduction

There is a plethora of instructional literature on object-oriented (OO) design [Lak96][Ber93]. This literature describes how OO systems *should* be structured, yet we have very little knowledge of how they are *actually* structured. In other words, we have very little idea of the extent to which software developers in industry follow the "design principles" proposed in this literature. Casual observation would suggest that many design principles are *not* widely followed in the construction of "real" software systems [FY97]. In this paper I explain why we would like to know with greater certainty the extent to which developers of OO software follow design principles, and how we might go about determining this.

We would like to know the extent to which software developers follow specific design principles so we can:

Better align research in software engineering with problems actually faced by practitioners. *Engineering* is about applying scientific and mathematical principles to practical ends. It follows that research in software engineering ought to focus on solving problems actually faced by practitioners. Unfortunately the perception of many practitioners is that research is not relevant to them [Par94]. One way we can improve the perception of software engineering research is to study the artefacts produced by practitioners. More specifically, by performing empirical studies of real software systems we can determine design principles that are not widely followed by practitioners. Such studies can then be used to convince practitioners of the relevance of tools, techniques, and educational material purporting to improve software structure.

Better study the effect of design principles on software quality. While it is widely accepted that the design principles presented in the literature lead to systems that are better (e.g., cheaper to maintain, less prone to error, easier to understand and so on) the reality is that we have little idea about the efficacy of these principles. In other words, seldom has there been *empirically* established a relationship between a design principle and a specific attribute of software quality [FP96, p.80]. A reason we lack knowledge of this nature lies in the (typically high) cost and difficulty associated with performing convincing empirical studies to expose such relationships. To get the best "bang for our buck" in performing such studies we ought to concentrate on design principles that are not widely followed. It is knowledge of these design principles that would be most useful to practitioners because, in a sense, practitioners have already accepted the overhead (and benefits) of applying design principles that are already in wide use.

Be more scientific in our research. There is a lack of "science" in the field of software engineering—many decisions made in industry are made solely on the basis of fashion, folklore or hype [FP96]. Research in software engineering also lacks "science". Tichy [Tic98] cites two studies that compare publications in computer science to publications in other science disciplines. Both studies found that a substantially higher proportion of publications in computer science lacked empirical (scientific) data to support the claims they made than in the other disciplines. By studying the extent to which design principles are evident in real software systems, we can be more scientific. We can begin to characterise a population that is of interest to us (i.e., the world's software systems). Other science disciplines have gained useful insights from characterising populations (e.g., in medicine obesity has been correlated with diabetes).

4.2 Approach

The approach I have taken in determining the extent to which developers follow certain design principles is to study their output—source code. In this respect I have attempted to build up a large, representative sample of software systems which I refer to as a *software corpus*. At the time of writing the corpus I have built up comprises 78 different Java systems, though it is growing as others in the research group to which I belong find applications to add to it. The systems in the corpus have been deliberately chosen to vary greatly in size, maturity and problem domain. Additionally the systems vary in where they have been sourced (e.g. Sourceforge, the

Apache Software Foundation, various universities, companies) and whether they are open- or closed-source.

So far the corpus comprises only software written in Java because (1) Java is widely used and taught, (2) it is (relatively) easy to analyse because of its bytecode representation, (3) there is a large amount of accessible Java software available (more than C# because Java has been around longer). I have concentrated *only* on Java (1) because of the overhead for me, personally, to build tools to analyse code spanning multiple programming languages; and (2) to leave open the opportunity for others to build up corpora of other languages, and perform parallel studies of design principles in these languages.

I have built tools to infer different forms of static dependencies (cf. dynamic or runtime dependencies) among the classes of Java applications. The specifics of these tools, the metrics they collect and some results from running them over the corpus are described in other works[MT06][MT07b]. To summarise the relevant findings, many of the applications in the corpus have many source files that *transitively* depend on many other source files. If we plot a histogram of these transitive dependencies then many applications in the corpus have a distribution reminiscent of that shown in Figure 1(a) [6]. This is almost invariably due to a dependency structure among all an application's source files resembling that shown in Figure 1(b).

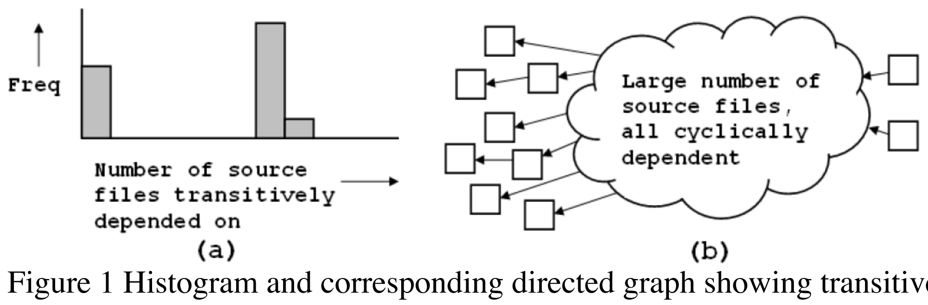

Freq
Number of source files transitively depended on
(a)
Large number of source files, all cyclically dependent
(b)

Figure 1 Histogram and corresponding directed graph showing transitive dependencies among an application's source files.

Design Principles. The design principles to which the transitive dependencies pertain are described in the context of the OO paradigm in Lakos' book [Lak96]. These principles are couched in terms of directed graphs, the nodes of which represent source files and edges of which represent compilation dependencies. In order to illustrate these principles consider the graphs of three different systems shown in Figure 2. Assuming that the designs of these systems are comparable which has the best structure?

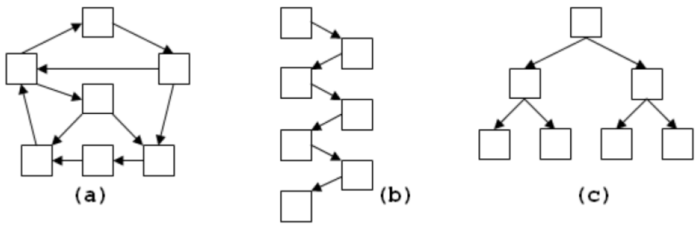


Figure 2 Source file dependency graphs of three comparable software systems.

Lakos [Lak96, ch.4] argues that the system represented by Figure 2(c) has the best structure because it has 4 source files that are "totally decoupled" (i.e., depend on no other source types) [Ber93]. This means these source files can (1) be thoroughly tested in isolation from the rest of the system, (2) each be reused verbatim in another system independently from any other source files in the original system, (3) each be understood in isolation, entirely independently from any other source files in the system, and (4) can each be developed by separate people, concurrently and entirely independently of each other. The other systems, 2(a) and 2(b), have fewer source files that are totally decoupled (zero and one, respectively). Additionally there are more topological orderings of 2(c) than 2(b) (2(a) has no topological ordering) which means there are more orders to proceed in building, (integration) testing and incrementally understanding it.

If we try to characterise the shapes of the three systems in Figure 2 we might characterise 2(a) as *cyclic*, since all the source files transitively depend on one-another so are cyclically dependent; 2(b) as *tall*, since when it has a greater height than 2(a) and 2(c) when its nodes are arranged on top of one another; and 2(c) as *flat* since it has a flatter structure than 2(b). Lakos's design principles stated in terms of "shape" are *avoid dependency cycles among source files* [Lak96, p.185], and *favour a flatter rather than taller graph* [Lak96, p.196].

4.3 Goals

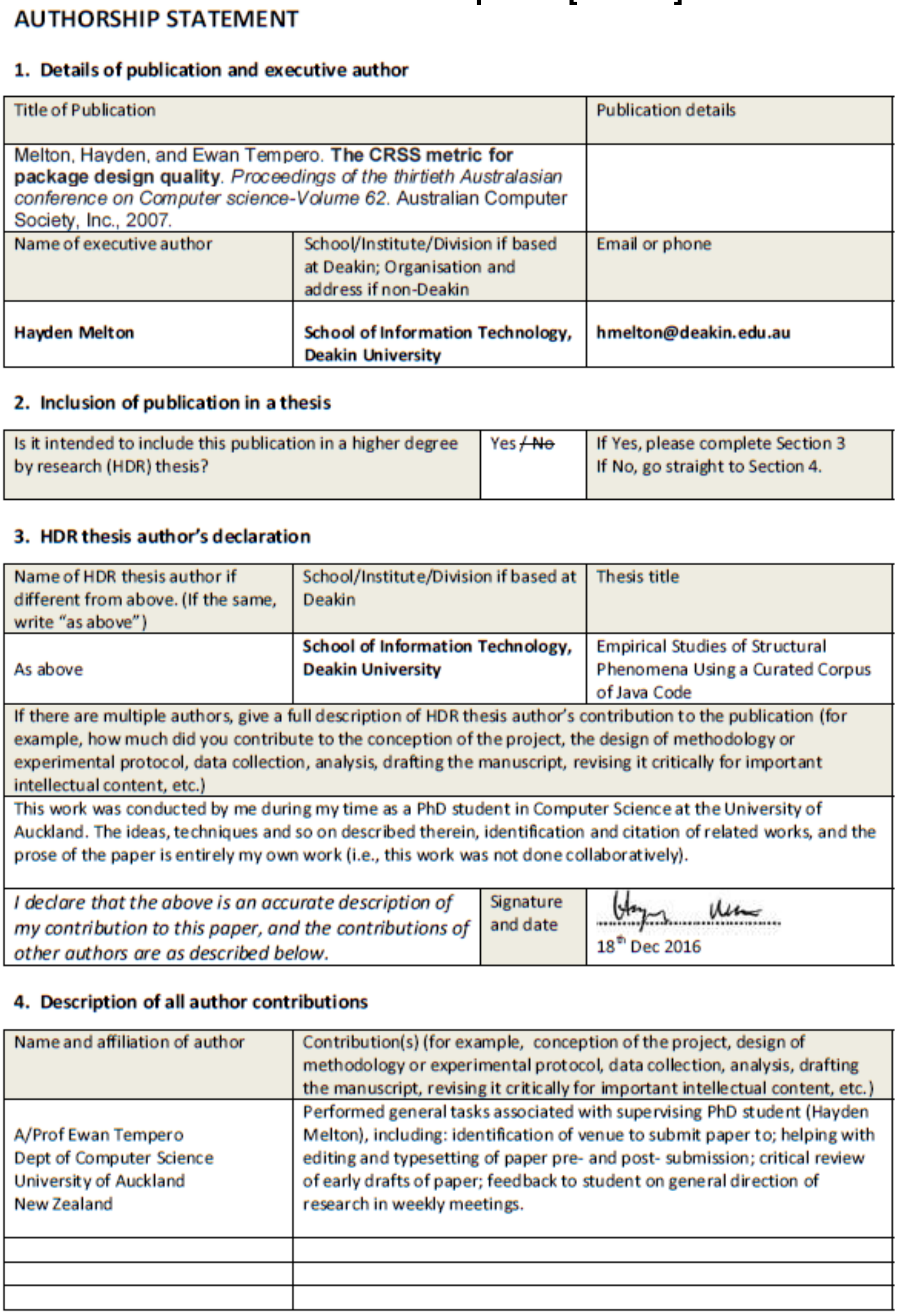
I have already collected data that shows the design principles proposed by Lakos are not widely followed by Java developers[MT06][MT07b]. This was one of my goals, my remaining goals are described here.

Empirically establish a relationship between these principles and understandability. I want to concentrate on linking these design principles to understanding because (1) I believe the rational arguments linking them to reuse, testing and buildability (see [Lak96] [Ber93]) are relatively strong in comparison and (2) understandability is a more fundamental attribute of software quality than others—in order to do almost anything to a software system (e.g., reuse, test, modify, etc) a developer must possess some level of understanding of it. I intend to go about establishing such a link by performing a controlled experiment or correlation using fault data from a release history. For the controlled experiment I will ask subjects to make modifications to a system with a tall, cyclically dependent structure and compare the effort of doing so to modifying a system with the same functionality but with a flatter, acyclic structure. This experimental setup derives from that used by Arisholm et al. [AS04]. The rationale for the fault correlation is that classes involved in cycles or with large transitive dependencies are more difficult to understand so are more susceptible to faults when they are changed.

Evaluate ways of disseminating my results to *practitioners*. Much research in software engineering is not presented in a way that is accessible to practitioners [Par94]. In order to address this I have begun to disseminate my results back to developers of specific applications in the corpus via mailing lists. The response so far has been mixed, but in the case of Azureus (a Sourceforge-hosted project) the data I collected led to some useful discussion and immediate refactoring. There was also useful discussion on the ArgoUML mailing list and an intent to improve the structure over time.

Make the Java corpus widely-accessible. A list of all the open-source applications in the corpus is available at http://www.cs.auckland.ac.nz/~hayden/corpus.htm. While this list enables others to locally replicate a significant proportion of our corpus, it is of limited use because (1) it does not contain data we used to mark-up the corpus (e.g. distinguish classes defined in source files from externally-defined classes) and (2) it doesn’t reduce the considerable amount of work involved in the downloading each of the application and getting it into a uniform structure suitable for automated analysis. What we really need is a large, widely-accessible, documented corpus so researchers can share data and replicate each other studies.

Coauthor Declaration for Chapter 5 [MT07a]



AUTHORSHIP STATEMENT

1. Details of publication and executive author

Title of Publication		Publication details
Melton, Hayden, and Ewan Tempero. **The CRSS metric for package design quality**. *Proceedings of the thirtieth Australasian conference on Computer science-Volume 62*. Australian Computer Society, Inc., 2007.		
Name of executive author	School/Institute/Division if based at Deakin; Organisation and address if non-Deakin	Email or phone
Hayden Melton	**School of Information Technology, Deakin University**	**hmelton@deakin.edu.au**

2. Inclusion of publication in a thesis

Is it intended to include this publication in a higher degree by research (HDR) thesis?	Yes ~~/ No~~	If Yes, please complete Section 3 If No, go straight to Section 4.

3. HDR thesis author's declaration

Name of HDR thesis author if different from above. (If the same, write "as above")	School/Institute/Division if based at Deakin	Thesis title
As above	**School of Information Technology, Deakin University**	Empirical Studies of Structural Phenomena Using a Curated Corpus of Java Code
If there are multiple authors, give a full description of HDR thesis author's contribution to the publication (for example, how much did you contribute to the conception of the project, the design of methodology or experimental protocol, data collection, analysis, drafting the manuscript, revising it critically for important intellectual content, etc.)		
This work was conducted by me during my time as a PhD student in Computer Science at the University of Auckland. The ideas, techniques and so on described therein, identification and citation of related works, and the prose of the paper is entirely my own work (i.e., this work was not done collaboratively).		
I declare that the above is an accurate description of my contribution to this paper, and the contributions of other authors are as described below.	Signature and date	[illegible] 18th Dec 2016

4. Description of all author contributions

Name and affiliation of author	Contribution(s) (for example, conception of the project, design of methodology or experimental protocol, data collection, analysis, drafting the manuscript, revising it critically for important intellectual content, etc.)
A/Prof Ewan Tempero Dept of Computer Science University of Auckland New Zealand	Performed general tasks associated with supervising PhD student (Hayden Melton), including: identification of venue to submit paper to; helping with editing and typesetting of paper pre- and post- submission; critical review of early drafts of paper; feedback to student on general direction of research in weekly meetings.

5. Author Declarations

I agree to be named as one of the authors of this work, and confirm:

i. that I have met the authorship criteria set out in the Deakin University Research Conduct Policy,

ii. that there are no other authors according to these criteria,

iii. that the description in Section 4 of my contribution(s) to this publication is accurate,

iv. that the data on which these findings are based are stored as set out in Section 7 below.

If this work is to form part of an HDR thesis as described in Sections 2 and 3, I further

v. consent to the incorporation of the publication into the candidate's HDR thesis submitted to Deakin University and, if the higher degree is awarded, the subsequent publication of the thesis by the university (subject to relevant Copyright provisions).

Name of author	Signature*	Date
Ewan Tempero	[illegible]	16/12/2016

6. Other contributor declarations

I agree to be named as a non-author contributor to this work.

Name and affiliation of contributor	Contribution	Signature* and date

* If an author or contributor is unavailable or otherwise unable to sign the statement of authorship, the Head of Academic Unit may sign on their behalf, noting the reason for their unavailability, provided there is no evidence to suggest that the person would object to being named as author

7. Data storage

The original data for this project are stored in the following locations. (The locations must be within an appropriate institutional setting. If the executive author is a Deakin staff member and data are stored outside Deakin University, permission for this must be given by the Head of Academic Unit within which the executive author is based.)

Data format	Storage Location	Date lodged	Name of custodian if other than the executive author
Qualitas Corpus of Java code, both binaries and source code with metadata (see *http://qualitascorpus.com/*)	University of Auckland, New Zealand	Starting 2006, onward	Ewan Tempero

This form must be retained by the executive author, within the school or institute in which they are based.

If the publication is to be included as part of an HDR thesis, a copy of this form must be included in the thesis with the publication.

Chapter 5 The CRSS Metric for Package Design Quality

Package design is concerned with the determining the best way to partition the classes in a system into subsystems. A poor package design can adversely affect the quality of a software system. In this paper we present a new metric, *Class Reachability Set Size (CRSS)*, the distribution of which can be used to determine if the relationships between the classes in a system preclude it from a good package design. We compute CRSS distributions for programs in a software corpus in order to show that some real programs are precluded in this way. Also we show how the CRSS metric can be used to identify candidates for refactoring so that the potential package structure of a system can be improved.

5.1 Introduction

The classes in an object-oriented software system can be partitioned into groups, or *subsystems* [WBWW90, p.135]. These subsystems serve to provide a higher-level view of the key abstractions in the system than that which is represented by individual classes [Boo91]. In large-scale software systems, comprising thousands of classes, subsystems are absolutely essential [Lak96][Boo91][CY91][Mar96b]. They help us to avoid information overload — a result of the limits on the human mind's information-processing capacity [CY91]. They facilitate a vocabulary that developers of a system can use in communication [CY91][Boo91]. They allow managers to determine a *partial ordering* of development activities with respect to time, that allows for parallelism in development effort [Mey95][Lak96][Mar96b]. Finally, they have a significant impact on the system's quality particularly with respect to reusability and testability [Lak96][Mar96b][SGM02].

One of the challenges in partitioning classes into subsystems is that for any given set of classes there are many possible ways to partition them [Mar96b]. The choice of partitionings is strongly influenced by the classes that make up the system because it is the relationships between these classes that cause relationships between the partitions in a given partitioning. It follows that relationships between partitions can be altered by moving classes between partitions (repartitioning) or by altering the source code of these classes to break their relationships with other classes.

Package design is the area concerned with determining the 'best' way to partition classes into subsystems[Mar96b]. The question addressed in this paper is "do the relationships between the classes in a system preclude them from a 'good' partitioning?". In order to answer this question we must determine what constitutes a 'good' partitioning. One contribution of this paper is a careful discussion of how partitioning (or package design) purportedly affects the external quality attributes of *reusability* and *testability*. In the case where the relationships between classes in a system do preclude them from a good partitioning we also provide advice on how to improve this situation through a refactoring strategy.

We define a metric, Class Reachability Set Size (CRSS), which we use in order to determine if the relationships between the classes in a system preclude them from a good package design. This metric counts, for a given class, all the other classes in the system's source code that it transitively depends-on for its compilation. In this way the metric takes into account the *whole system*, not just individual classes from selected subsystems. We show how the *distribution* of the CRSS values for all of the classes defined in system is useful for answering our question. We present empirical evidence to support this claim.

Our use of the CRSS metric is to determine whether design principles proposed by others can be met by an existing class structure, that is, we provide an *operational* means to check conformance to these design principles. However, the CRSS metric says nothing about whether the design principles themselves are correct, that is, following the principles lead to a higher quality design than not following them. In fact, we have found very little empirical evidence to support *any* design principles. We consider this a serious lack in software engineering research. We believe this is due to the difficulty in operationally checking conformance, and so believe that developing metrics such as CRSS to be a promising approach to understanding the structure of software.

The remainder of this paper is organised as follows. In Section 2, we survey the package design principles proposed in the literature. Section 3 discusses in detail how these package design principles impact testability and reusability. We then present the CRSS metric in Section 4. In Section 5 we present an empirical study on the use of CRSS on a corpus of Java software systems. In Section 6, we demonstrate a new refactoring strategy that uses CRSS and one other metric in order to identify classes for refactoring so the package design quality can be improved. We discuss related work in Section 7 and conclude with Section 8.

5.2 Background

5.2.1 Package Design

Programming languages such as Java, C++ and Ada support a higher-level of organisation through the `package` construct. Packages allow classes to be organised into named abstractions more generally referred to as *subsystems*. Within a system there may be subsystems at several levels of abstraction [Lak96] [WBWW90] [CY91] [Boo91]. In this respect the subsystems at a given level of abstraction can themselves be partitioned into new subsystems representing a higher-level abstraction.

Package design is ultimately about organising classes into subsystems. In this respect programming languages without the package construct can still allow the level of abstraction provided by subsystems. Subsystems can be realised without the use of packages by arranging source files into separate (file system) directories, the names of which identify these subsystems. Alternatively, or additionally, subsystems can be realised through multiple class declarations in a single source file[Lak96].

Many authors have identified principles for package design but have referred to it using different terms. Lakos, for instance, uses the term *physical design* to collectively refer to his principles for package design [Lak96, p.97]. Martin uses the term *package design* [Mar96b]. Earlier work in package design often does not give it an explicit name but uses terms such as *class category* [Boo91], *clusters* [Mey95], *subject areas* [CY91], *domains* [SM92] and *subsystems* [Boo87] to refer to its fundamental units. Our review of the package design literature has identified two flavours of design principles. First are those that relate to the formation of classes into individual packages. Second are those that relate to properties of the directed graph formed by the dependencies between the packages at a given level of abstraction.

5.2.1.1 Package Formation

The `package` construct, at least in Java, is a recursive structure in that it contains classes and/or other packages. It is the recursive nature of a package that allows it to represent subsystems at different levels of abstraction. A given package can represent a subsystem at a higher level of abstraction than the subsystems represented by the packages it contains. The principles of **manageable size**, **stand-alone**, **cohesive**, and **encapsulation** have been proposed to guide package design. We address the first two in this paper.

Manageable Size. The number of items (packages or classes) contained by a package should not exceed a given limit. Coad et al. identify Miller's paper on 'the magic number seven, plus or minus two' [Mil56] as the basis for this principle [CY91, p.107]. Essentially Miller's paper states that the short-term memory of a human can hold 5-9 things at a time. Based on Miller's work it could be argued that for package to be quickly understood (using our short-term memory) it should contain 5-9 other packages or classes. Other authors differ on this limit, but nevertheless identify the need for a limit to a package's size. Lakos identifies 500 to 1000 lines of code (LOC) for a *component* (low-level subsystem), and 5000-50000 LOC or a few dozen components per *package* (higher-level subsystem) [Lak96, p.481]. Meyer states a *cluster* should contain 5-40 classes and be able to be developed by 1-4 people and entirely understood by a single person [Mey95, p.51].

For the purposes of this paper we will not specify a particular limit for package size other than to say that such a limit should exist, and that the limit can be stated in terms of number of classes directly or indirectly contained by a package. The limit may be dictated by company policy, personal preference, or some other mechanism. All of our arguments apply to any limit on package size, so long as the limit exists.

Stand-alone. A package should be stand-alone in that it should have minimal dependency on other packages [Lak96, p.147]. A given package depends on another if its classes cannot be compiled without some of the latter's classes. The notion of compilation dependency among packages is important because we want to be able to lift packages from one program for deployment in another. In this way we can reuse code without having to modify its textual content to remove dependencies. The stand-alone property of a package is also important for understandability, testability, and the extent to which parallel development effort can occur across packages in a system [Lak96, p.149-202].

5.2.1.2 Graph Properties

The principle that a package should be stand-alone leads to more package design principles when it is applied to *all* the packages in a system. These principles are auxiliary in that certain characteristics of what we refer to as *Package Dependency Graphs (PDGs)* imply that a system's packages are not stand-alone. If a system's PDG contains cycles, or is 'taller' rather than 'flatter', then the packages that comprise the system cannot be as stand-alone as their flat, acyclic analogs.

A PDG is a directed graph representing all the packages in a system's source code at a given level of abstraction as nodes, and compilation dependencies between these packages as directed edges. Packages have compilation dependencies on each other due to the underlying dependencies between the classes that they contain (both directly and indirectly through their subpackages). We say package `A` *depends on* package `B` if any class directly or indirectly contained by `A` depends on any class directly or indirectly contained by `B`. We will present a formal definition of what it means for a class to depend on another in Section 4. Also, since packages may exist at different levels of abstraction in a system, a system may have several PDGs.

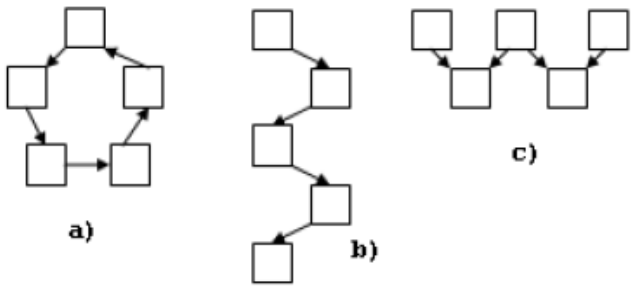


Figure 1: Cyclic, tall and flat PDGs

The graphs in Figure 1 are PDGs. We can reasonably compare them to one another because they comprise the same number of subsystems (vertices) and the same number of dependencies (edges) (except (a), which has an extra edge). The purpose of these PDGs is to illustrate that tall and cyclic graphs cannot comprise packages that are stand-alone, so PDGs should be flat and acyclic. Consider firstly any package from the cyclic PDG 1(a). In order to deploy this package in another program we also have to copy with it all the other packages in the graph. Even though the package itself directly depends on only one other package, this other package also depends on another to compile. The process goes on for the transitive closure of the dependency so at least in terms of deployment the stand-alone property of a package can be considered on the basis of *transitive* dependencies.

The argument is similar for the tall graph of Figure 1(b) the top-most package requires all other packages in order for it to be deployed in another system. The tall graph of (b) is better than in the cyclic graph of (a) because packages towards the bottom of the graph transitively depend on fewer and fewer other packages. The flat graph of Figure 1(c) is better than the tall and cyclic graphs because it has the most packages that can be deployed with the minimal number of other packages, so each package is more 'stand-alone'.

One problem with the PDGs of Figure 1 is that they are not indicative of real designs because real designs tend to have more direct dependencies between packages and tend to have more 'layers'. Lakos claims that a PDG that forms a balanced binary tree (see Figure 2) is a good reference point with which compare real designs[Lak96, p.187], although he notes that real designs are not nearly so regular. In terms of deployment, leaves of the tree are the most stand-alone since they depend on no other packages. More than half of the packages in a balanced binary tree depend on no other packages. One quarter of the packages in such a tree can be deployed with just two other packages, and so on.

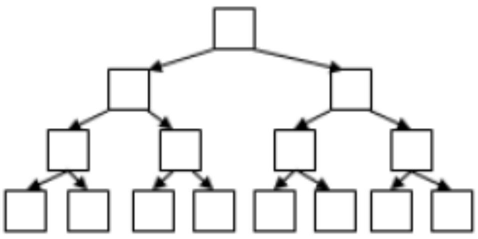


Figure 2: Balanced Binary Tree PDG

This discussion of design principles for PDGs has justified these principles in terms of the packages in a graph being stand-alone so that they can be deployed in other systems. There are more reasons other than deployment for ensuring that PDGs are flat, acyclic graphs. These are more complicated and are discussed in the following section.

5.3 Effects on Quality

The motivation for any design principle, whether it relates to classes (e.g. group related logic and data together), object interactions (e.g. design patterns) or packages is that the application of the principle will improve the quality of software system in some way. With regard to package design the claim is that allocating classes to packages according to the package design principles described above will result in a system of higher quality than if classes were allocated to packages in a more ad-hoc fashion. In this section we present a discussion of how the manageable size and stand-alone package design principles clearly relate to reusability and testability.

5.3.1 Reusability

Reusability is defined as “the degree to which a software module or other work product can be used in more than one computer program or software system” (IEE 1990). One can reuse things of a conceptual nature such as software architecture descriptions and design patterns, or things of a more ‘binary’ nature such as procedures, classes and modules [SGM02]. The literature on package design claims improved reusability on the basis of reusing the functionality implemented in the source code of one program in another. This relates to quality because reusing code from one program in another can lead to reduced development effort and fewer defects since the reused code has been ‘proven’ in the context of its original program [GJC92].

Code reuse involves copying source files (and the libraries that they depend on) from one program to another, without having to modify the textual content of these source files. Compare this to *code copying* where text is copied from one program to another, usually meaning the copier has to modify the text to make it work in his environment and the copied code eventually diverges so much from the original that it becomes unrecognisable. This is a problem because the copier of the code becomes responsible for its implementation and it is no longer possible to easily integrate new versions of the code from its origin system following bug fixes and enhancements made by the owner of the code [Mar96b].

Packages are inherently related to code reuse because a class is not the fundamental unit of deployment [Mar96b][Lak96][SGM02]. It would be unusual for a single class copied from one program to be able to be compiled in the context of another program. Chances are that the class would depend on other classes appearing in its methods return types and parameters, as well as in the bodies of its methods’ implementations. If the class performed some domain-specific function then it is likely that at least some of the classes it depends on would also be defined in other source files of the original program (as opposed to classes defined in the programming language’s API). In this way whole packages should be copied [Mar96b], not individual source files each containing a single class.

In terms of package design, packages that are standalone (and of a manageable size) lend themselves to reuse because we have to copy fewer packages (and classes) from one system to the other. While the sheer number of the packages copied is of concern because it increases the amount of code in the system that needs to be understood and can contain bugs, it is not the main problem. Rather the problem is that many of the packages are not likely to be strictly necessary for the given package to provide its functionality, or as Lakos [Lak96, p.14] states “in order for a … subsystem to be reused successfully, it must *not* be tied to a large block of unnecessary code”. This is where flatter, acyclic graphs come in.

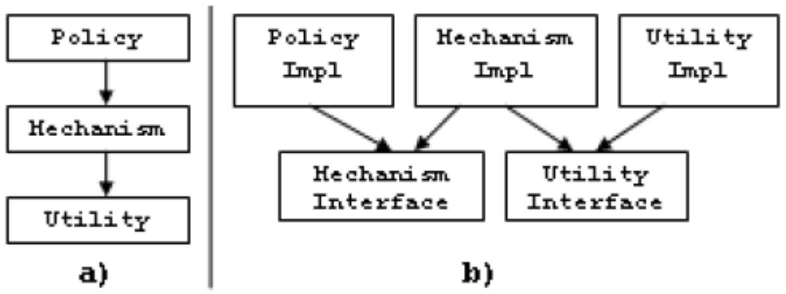


Figure 3: Flattening a tall PDG with the DIP

Tall or cyclic PDGs are not well suited towards reuse and many such graphs can be transformed to become flatter and acyclic through class refactoring and splitting packages into an 'implementation' and an 'interface'. Martin's *Dependency Inversion Principle (DIP)* shows how a tall graph can be transformed into a flat graph [Mar96a]. Figure 3 illustrates the transformation proposed by Martin. The individual packages in Figure 3 have an improved potential for reuse because the 'implementation' packages are more *flexible*. They are more flexible because they can be used with different implementations of the other packages. For instance the `Mechanism` Implementation package in 3(b) can now be used with different implementations of the `Utility` interface. This also means that the reliability of the packages in 3(b) is improved because we do not have to deploy the implementation packages in another system if we do not want them. For instance we can deploy `Mechanism`'s implementation in a new system without having to copy `Utility`'s implementation into the system. Not having to copy the Utility implementation can improve the reliability of the new system because Utility implementation cannot be a further source of bugs if it does not exist in the system.

5.3.2 Testability

Testability is often defined as the ease at which software can be made to demonstrate faults through testing [BCK98, p.88]. Package design purports improved testability by demonstrating faults through execution driven by automated unit tests (as opposed to execution driven by a user) [Lak96].

We define an automated unit test as a piece of code that exercises another piece of code, and automatically compares the expected effect of that execution to the actual effect in order to report success or failure of that test. This type of testing is particularly useful for regression testing i.e. identifying faults that have been caused by unintended effects of a modification to a system outside its apparent scope.

Flat, acyclic PDGs have subsystems that lend themselves well to automated unit testing for reasons similar to why they lend themselves to reuse. If a subsystem in a PDG depends on an interface rather than an implementation we can more easily test it using stubs (or *mock objects*, as they are more popularly referred to nowadays). Stubs increase *controllability* and *observability* during testing [Bin99, p.980]. In terms of controllability we can implement a stub to exercise the boundary values and special cases for the given subsystem's interactions with the package it depends upon. This is essential when these special cases occur as a result of nondeterministic behaviour, or are difficult to set up, or are difficult to trigger (e.g. an out-of-disk-space error) or have callback functions [TH02] in the dependee package's actual implementation.

Flat, acyclic PDGs with stand-alone components of a manageable size are also more cost effective to unit test because they can be tested in 'isolation' [Lak96]. This relates back to testing a subsystem using stubs, rather than the actual implementations it depends on at runtime. The rationale for this claim is that testing in isolation means that stubs and test cases are created just to test the functionality provided by the component itself. This means that the complexity of the test reflects the complexity of the component. Reducing the complexity of the test is important because units tests are also code which costs money to produce. A further advantage of testing in isolation is that the tests provides a small but comprehensive example illustrating the use of that subsystem, helpful to someone wanting to reuse it [Lak96].

5.3.3 Other Quality Attributes

Other claims have been made regarding the effect of package design principles on quality. Coad and Booch imply package design improves *buildability* by facilitating a vocabulary that developers of a system can use in communication [CY91][Boo91]. Another claim is that package design allows managers to determine a *partial ordering* of development activities with respect to time, that allows for parallelism in development effort [Mey95][Lak96][Mar96b]. Probably the most contentious claim is that package design principles lead to a package structure that makes the software system more understandable. While it is clear that packages provide a higher level of abstraction than classes which helps us to avoid information overload [CY91], it is less clear whether the 'interface' and 'implementation'-style of packages particular to flat, acyclic PDG improve understandability because they seem to increase the number of packages in the system (compare Figure 3(a) to (b)).

5.4 Class Reachability Set Size

The *Class Reachability Set Size (CRSS)* metric is computed from the *Class Dependency Graph (CDG)*. A CDG is a directed graph where the vertices are the top-level classes defined in the source files of the software system and the edges represent compilation dependencies. The CRSS for a class is then the number of vertices reachable from the vertex representing that class.

More formally, for a class `C`, the relation DEPENDS-ON(`C`) is the set of classes that must be available in order to compile `C` (ignoring those classes referred to by redundant `import` statements). In practical terms, in Java, it is the set of `.class` files that must be on the classpath in order to compile `C`.java. Another way to think about it is that is the number of distinct types that are referred to by names that appear in `C`.java. CRSS(`C`) is then the distributions size of the set representing the *transitive closure* of the DEPENDS-ON relation as applied to `C`.

Our interest is in the best possible package structure allowable with respect to packages being stand-alone and of manageable size given the relationships between the classes in the system. Just considering the measurements given by CRSS for a single class is not going to do this. What we need is something that is representative of the whole system, not just an individual element of it. Rather than consider something like the mean or standard deviation of CRSS, we use the *distribution* of the CRSS values. As we will argue below, it is the *shape* of this distribution that is important in understanding the best potential quality of a package design for the system.

Since we are computing CRSS for every class in the system, we need to be clear as to which classes' CRSS values are used in the distribution. We only consider 'top level' classes, that is, those that are not nested. Nested classes are not directly represented in the distribution, although their DEPENDS-ON set is computed and contributes to the CRSS value of their lexically enclosing class.

Nested classes are not represented in the CDG because their use seldom constitutes a major design decision [Boo91, p.161]. It is often the case that these classes are not visible outside their lexically enclosing top-level class, which means other classes cannot depend on them. The classes on which a nested class depends are merged with the dependencies of its top-level class in order not to perturb the actual dependencies between packages. This is assuming that a nested class belongs to the same package as its top-level counterpart, which is certainly true for Java. We also consider only classes defined in the source files of a system because these are the only classes within the system whose package membership can be altered. Classes defined in external libraries are often in binary (vs. source) form, so cannot have their package declaration altered. Even if these classes' sources were available it is likely that the developers of a system would be unwilling to take ownership of this code in order to improve its package structure. Considering only classes defined in source files is consistent with other efforts [Lak96][Boo96b].

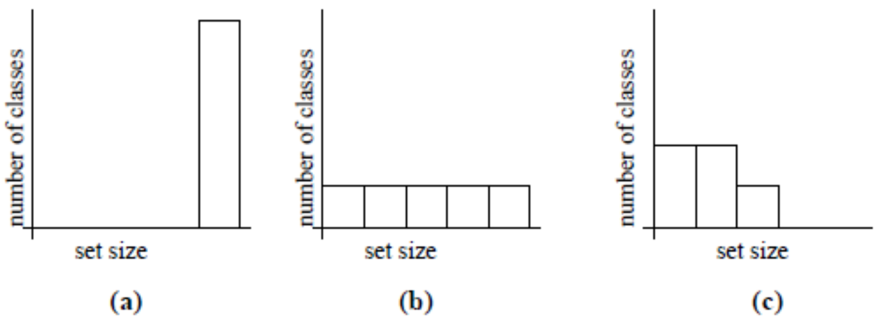


Figure 4: Relationship between PDG structure and CRSS

To get an idea of how the CRSS distribution relates to the structure of a PDG, consider again the PDGs shown in Figure 1. If we assume that each package has the same number of classes and that every class in a package depends on every other class in the same package then we get distributions like those in Figure 4. So, for example, if there are n classes in each package, then every class in a package in Figure 1(a) depends on every other class (5n in total) in the system (a total of 5n classes), whereas only the classes in the middle package (a total of n) on the top row of Figure 1(c) depends on 3n other classes.

Figure 4 gives an indication of the kind of distributions we might see corresponding to PDGs with different characteristics, but what we need to know is, what does a given distribution tell us about the underlying PDG? The main contribution of this paper is that, if the CRSS distribution is such that there are ‘many’ classes with a ‘large’ CRSS value, then the current package structure for this system cannot meet Lakos’ model PDG (see Figure 2). Furthermore, and crucially, this situation indicates that the class relationships are such that there is *no* way to partition the classes to meet the Lakos model PDG, meaning that the only way to improve the package design is to change class relationships.

To see how certain distributions allow us to conclude that the package design is not as good as it could be, we need to be more specific about ‘many’ and ‘large’. Rather than present the algebraic argument, we will give an indicative concrete example.

Suppose that we have a system with 1000 classes in it, and suppose we have decided that a package of ‘manageable size’ would have no more than 50 classes. If the package design for this system does not violate the manageable size principle, there must be least 20 packages, and for this example we will assume there are exactly 20 packages of 50 classes each. The question is, how many stand-alone packages can there be, given a certain CRSS distribution.

The CRSS distribution we will consider is, 500 of the classes (L) have CRSS values of 99 or fewer, and the other 500 classes (R) have CRSS values of 600–699. The classes in L could conceivably be partitioned into 10 (half) stand-alone packages, which is roughly consistent with the Lakos model PDG. So consider a class A in R. It transitively depends on 600 or more other classes, and these 600 classes must be distributed over more than 12 packages (since 50 classes per package). At most 10 of those packages may involve only classes in L, so the package containing A must depend on at least 2 other packages involving classes in R. Since this is true for every class in R, every package involving classes from R must transitively depend on at least 2 other packages involving classes from R. There can only be 10 such packages, so this is only possible if there is a cycle in the PDG, which means the any PDG for these classes cannot be in line with Lakos' tree model PDG.

The example given above may seem like an unlikely extreme cases, however, as we discuss in the next section, distributions similar to this are more common than one might expect. The advantage of the CRSS distribution is that it can be cheaply determined, and so quickly provides a reliable indication of the potential quality of the package design. Of particular advantage is that the information provided is independent of the actual package structure of the system we are measuring (see Section 7.1).

5.5 Results

5.5.1 A Software Corpus

We have developed a tool to compute CRSS from Java source files. We ran our tool over a corpus of Java software in order to determine the distribution of CRSS values in each of its programs. Programs selected for the corpus largely derive from the Purdue Benchmark Suite (PBS) used in an empirical study of type confinement [GPV01]. Programs in the PBS omitted from our corpus were those whose source code was not available. We have replaced these programs with others that we have previously used and whose source is freely available on the Internet.

The distributions of CRSS for each of the programs in our corpus are shown in the histogram of Figure 5. Being a histogram the horizontal axis shows the ranges of values for CRSS and the vertical axis shows the number of classes a given program that have that range of values for CRSS. The axis going 'into' the page shows each of the programs in the corpus, sorted by size, where this is measured in the number of top-level classes defined in the program's source. Again, since Figure 5 is a histogram, the heights of the bars for a given program sum to the number of (top-level source) classes in that program.

Several of the programs in Figure 5 appear to have 'bad' CRSS distributions in that a large proportion of the classes in these systems have relatively high values for CRSS.

We single out Azureus for further discussion because we were initially familiar with it from the perspective of an end-user and because there are space constraints on this paper. A histogram of CRSS values for Azureus is depicted in Figure 6 for the purposes of clarity.

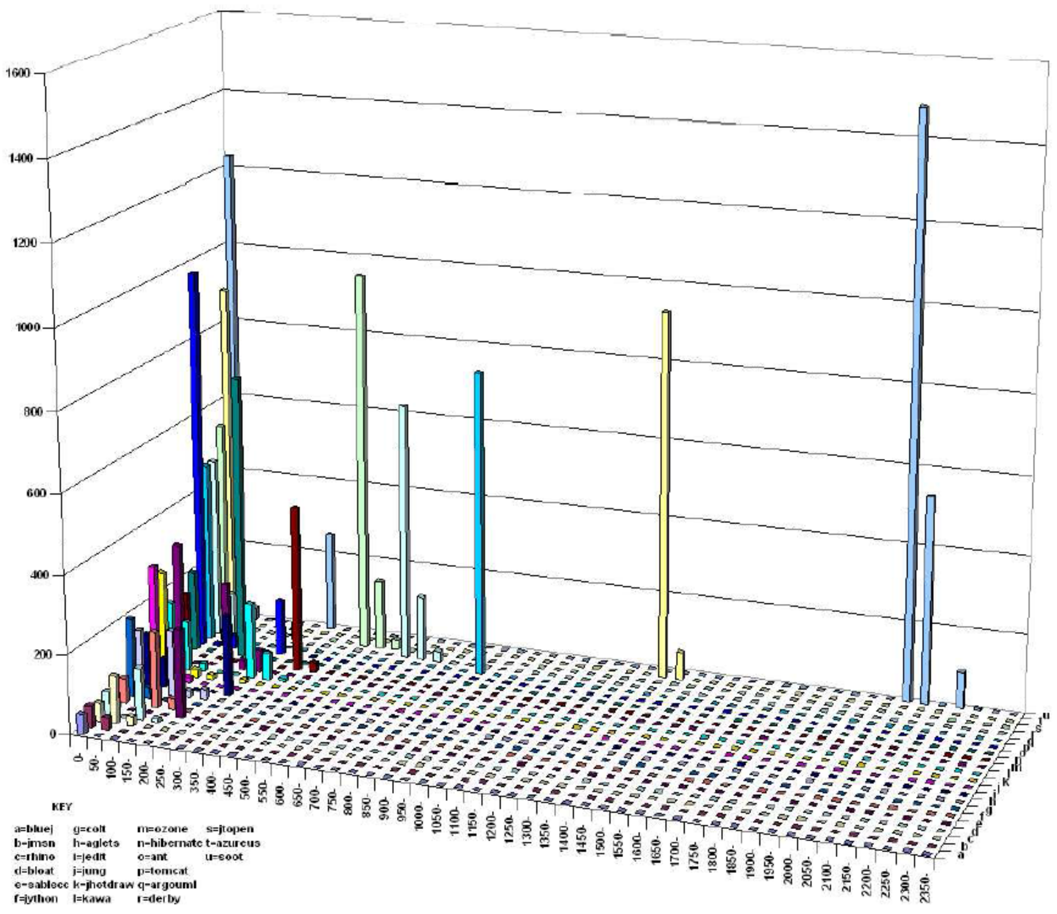


Figure 5: Software Corpus CRSS Distributions

5.5.2 Azureus

Azureus is peer-to-peer file-sharing client for the BitTorrent protocol. It was initially brought to our attention because it frequently appears on Sourceforge’s title page in the top 10 lists for both downloads and development activity. We have used it and found that, at least from a user’s perspective, it is a good piece of software because it is stable and easy to use.

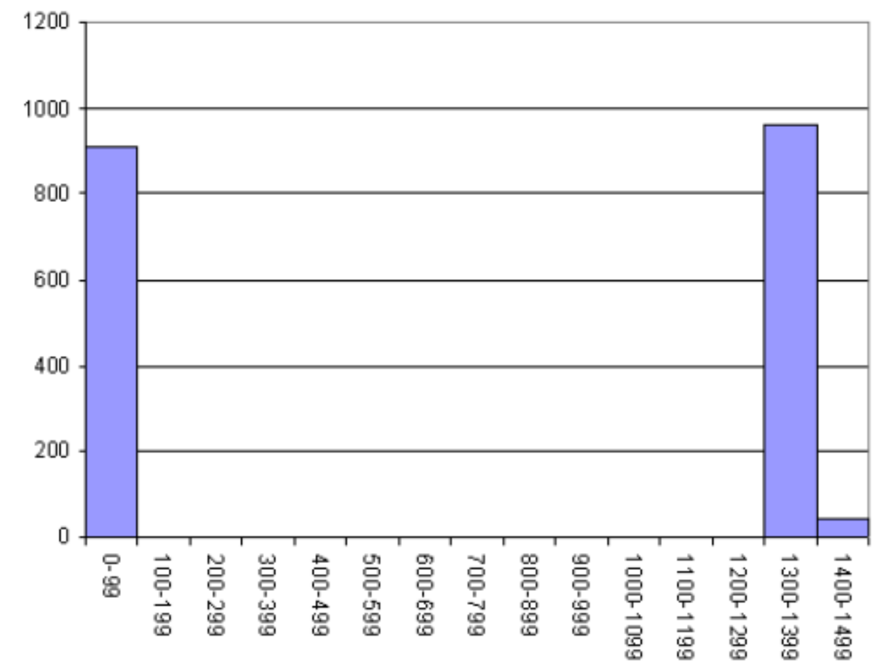


Figure 6: Azureus CRSS Distribution

The histogram of Figure 6 shows that there are approximately 1900 top-level classes defined in Azureus's source files (the sum of the heights of the bars). Of these 1900 classes about 900 have CRSS values of between 0 and 99. This means that each of these classes transitively depend on between 0 and 99 other classes. The remaining two bars combined show that about 1000 classes depend on between 1300 and 1499 other classes. In fact, the transitive nature of CRSS means that none of the classes in the left-hand bar can depend on those in the right-hand bars. If a class from the left-hand bar depended on one in the right hand bars, it too would depend on 1300-1499 other classes so itself would have to be in the right-hand bars.

Table 1 shows a small selection of subsystems we have identified in Azureus many of which are not reflected in its current package structure. In column 2 of this table there is a representative or key class for each subsystem, or one that plays the role of Facade. The CRSS value given for the subsystem is computed from this class. As indicated in the 'CRSS' column of Table 1 each of the subsystems has a key class with a large CRSS value (ignore the 'CRSSrefact' column for now). This indicates that the subsystems depend on a great many other subsystems. Indeed we inspected the reachability sets of the classes in Table 1 and found that these key classes are actually mutually dependent. This means that the subsystems these key classes represent are also mutually dependent and that there must be a cycle among them.

Even without the knowledge that the classes of Table 1 are mutually dependent, the values in 'CRSS' column are still meaningful. For instance, it is hard to believe that a seemingly low-level subsystem like logging can depend on 1372 classes. If the maximum subsystem size of a logging subsystem's peers is 50 classes then it must transitively depends on *at least* 28 other subsystems. Continuing under the assumption that the maximum subsystem size is 50 classes then we can infer from Figure 6 that, irrespective of package structure, there are at least 20 subsystems represented in the right-hand bars and that these subsystems must each transitively depend on at least 26 other subsystems. The degree to which these subsystems is stand-alone is a far cry from Lakos's balanced binary tree reference model.

Subsystem	Facade- or key-class	CRSS	CRSS-refact
debug	org.gudy.azureus2.core3.util.Debug	1372	16
threading	org.gudy.azureus2.core3.util.AEMonitor	1372	26
configuration	org.gudy.azureus2.core3.config.COConfigurationManager	1372	1360
internationalisation	org.gudy.azureus2.core3.internat.MessageText	1372	44
logging	org.gudy.azureus2.core3.logging.LGLogger	1372	12
torrent	org.gudy.azureus2.core3.torrent.TOTorrent	1372	32
time	org.gudy.azureus2.core3.util.SystemTime	1372	5
peer	org.gudy.azureus2.core3.peer.PEPeer	1372	171
disk-io	org.gudy.azureus2.core3.disk.DiskManager	1372	171
...	...	...	...

Table 1: Azureus subsystems

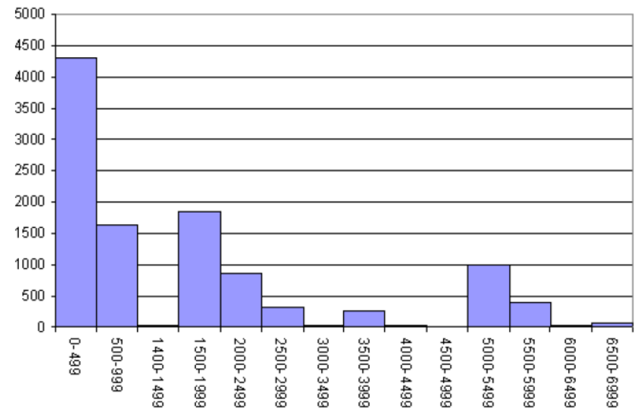


Figure 7: Eclipse CRSS Distribution

5.5.3 5.3 Eclipse

We also collected CRSS values for classes in the open-source IDE Eclipse, version 3.0.2 for Windows[11]. The distribution of these CRSS values are shown in Figure 7. There are approximately 10700 top-level classes in Eclipse's source code. Figure 7

[11]Eclipse is not shown in the corpus distribution because its size diminishes the heights of bars in the other programs too much.

shows a decreasing trend in values for CRSS. Smaller values for CRSS appear to be more common than larger values. This is good because it means that dependencies between classes in Eclipse do not preclude it from having tree-like package structure. The right-most bar in Eclipse's CRSS distribution comprises only about 100 classes each of which transitively depend on 6500-6999 other classes. If the maximum package size at some level of abstraction is 500 classes it is feasible that only one package in the system transitively depends on 13 other packages. The taller the bar in the 6500-6999 the more packages that can potentially transitively depend on 13 other packages, thus the less stand-alone the packages that comprise Eclipse would be.

We do not present a table in the style of Table 1 for Eclipse because its size means that there are likely to be subsystems at many levels of abstraction. Instead we focus on two subsystems we have, in the past, wanted to lift from Eclipse for deployment in other programs. The first is Eclipse's Abstract Syntax Tree (AST) subsystem and the second is Eclipse's Resource Finder subsystem.

Eclipse's AST subsystem provides an Abstract Syntax Tree representation of a Java source file, or a set of Java source files. Other subsystems make use of this subsystem e.g. a Refactoring subsystem uses the AST for refactorings such as rename class, extract interface, override method. A Source Code Navigation subsystem uses this AST to perform operations such as goto declaration, open type hierarchy, find referring types. In essence the AST subsystem is a Java compiler front-end—it parses Java source code, does name bindings and produces an AST. The facade class for Eclipse's AST is `ASTParser`. We found that it has a CRSS value of 1572, which is we think is unusual because Sun's own Java compiler (for Java 5.0), which includes a back-end for writing ASTs to byte code comprises only 71 top-level classes.

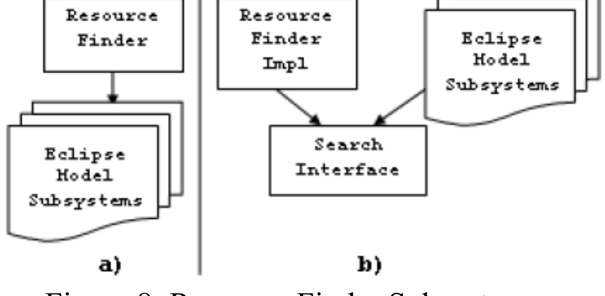


Figure 8: Resource Finder Subsystem

There are several differences between Sun's Java compiler and Eclipse's AST subsystem that could cause a difference in CRSS values but none of which we think explain the magnitude of the difference. The differences are:

• Eclipse's AST subsystem relies on Eclipse project wrapper `IJavaProject` whereas Sun's gets external libraries, resources and source files off the classpath.

• Eclipse's AST subsystem allows the progress of the parsing and name binding to be monitored with `IProgressMonitor` although Sun's compiler also has a mode in which output messages could be interpreted to gauge the progress of the compilation.

• The AST node subclasses in Sun's compiler are `public static` inner classes whereas they are top-level classes in Eclipse's AST.

In any case none of these extra functions provided by Eclipse's subsystem should cause it to transitively depend on approximately 1500 more classes, especially since Sun's compiler provides the extra functionality of compiling to byte code. So we believe that Eclipse's AST subsystem could benefit from the type of refactoring depicted in Figure 3.

We have found the functionality provided by Eclipse's Resource Finder subsystem very useful when dealing with projects with many resource and source files. The Resource Finder dialog can be opened by pressing `(ctrl+shift+r)` in the IDE. This pops up a dialog that works by accepting a regular expression input and finding all files in open projects with filenames that match that regular expression. The facade class for the Resource Finder subsystem is `OpenResourceDialog` and has a CRSS value of 1945. Lifting 1945 other classes in order to reuse this Resource Finder subsystem is impractical considering our code inspections showed that the functionality provided by `OpenResourceDialog` is actually contained within only 5 classes. The problem is that `OpenResourceDialog` transitively depends on classes in Eclipse's model (c.f. view) (as shown in Figure 8(a)) when it should depend on some interfaces that are, in turn, passed into the model as shown in Figure 8(b).

5.6 Refactoring

5.6.1 Strategy

We have developed a refactoring strategy based on the *Dependency Inversion Principle (DIP)* [Mar96a] to reduce the number of classes in a system with large CRSS values. The strategy uses properties of the CDG to identify candidate classes for refactoring. The particular refactoring performed is *extract interface*, which may seem trivial, but we will see that eliminating the dependency on the implementation of the extracted interface is tricky. This trickiness occurs because at some point we must instantiate an interface with its implementation type—we refer to this as the 'problem of instantiation', which is discussed below.

Performing the extract interface refactoring on a class reduces its clients' CRSS values because the client classes no longer transitively depend on any types used in the extracted interface's implementation. The effectiveness of the extract interface refactoring is dependent on many of the types referenced in the interface's implementation not appearing in the signatures of the methods (and possibly fields) on the interface. In this way the CRSS value of the extracted interface is likely to be smaller than the value of its implementation. The transitive nature of reachability sets ensures that the clients of interface are likely to have smaller CRSS now than when they referenced what was effectively the interface's implementation.

It follows that an effective way of reducing the CRSS values of many classes in a system is to extract interfaces from classes that are widely referenced and themselves have high values for CRSS. This is where the CDG comes in—a class is widely referenced if its CDG node has a large in-degree. Thus we identify candidate classes for the extract interface refactoring by sorting the list of classes in the system by in-degree then CRSS. While the extract interface refactoring is fairly simple to perform, dealing with client classes that need to instantiate the interface is not. In order to instantiate the interface we need to reference the interface's implementation. If this is done through the use of a constructor call e.g. `Interface i = new Implementation();` we are in the same situation with respect to the client's CRSS as before because the client still depends on the implementation. If this is done through reflection e.g. `Interface i = Class.newInstance ("Implementation");` we still have a dependency on the implementation though our tool will not detect it and we have lost some of the type-safeness provided by the language. Even if we use a factory class to return an instance of the implementation we still have a transitive dependency on the implementation through the call to the factory method that instantiates the class.

There are a number of ways of dealing with the problem of instantiation that are dependent on the way in which the interface is used. If the interface is instantiated only for a field in the client we can pass in the instantiation through the constructor:

```
public Client {
  private Interface i;
  public Client(Interface i) {
    this.i = i;
  }
}
```

In this way the class that instantiates the client also instantiates the interface's implementation and passes it in through the client's constructor. The client has no reference to the interface's implementation. This technique is often referred to as *dependency injection*. Unfortunately it can result in more involved refactoring of the clients than simply textually replacing all references to the class's name with its extracted interface's name – sometimes extra parameters have to be added to the constructors and clients of the original clients need to be modified to instantiate the interface's implementation.

We concentrate on performing the extract interface refactoring on candidate classes that are singletons because there is a means to instantiate these classes that puts little refactoring burden onto their clients. The ideal implementation of a singleton object through the use of a single static `getInstance`-type method and a private static field holding the instance. In reality we have found that singletons are implemented in a variety of ways (e.g., entirely using static methods and/or entirely using static fields). The solution we have for the problem of instantiation in the context of singletons involves the use of a *registry of singletons* [GHJV95, p.130].

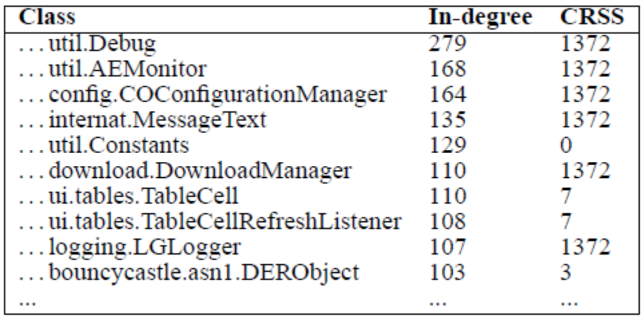
We illustrate how the burden of refactoring on a singleton's clients after the extract interface refactoring is performed on the singleton is reduced through the use of a registry of singletons. We illustrate this refactoring on the class `A` shown below, which gets split into `AIFace` and `AImpl`.

```
//this is the code pre-refactoring
public class Client {
  //inside some method
  A a = A.getInstance();
}
//this is the code post-refactoring
public class Client {
  //inside some method
  AIFace a = (AIFace)SingletonRegistry.get("A");
}
//this is a registry of singletons
public class SingletonRegistry {
  private Map m = new HashMap();
  public put(String key, Object value) {
    m.put(key, value);
  }
  public Object get(String key) {
    return m.get(key);
  }
}
//this line is needed somewhere near the entry
// point of the application to populate the
// registry with instances
singletonRegistry.put("A", new AImpl());
```

While we have used this refactoring only on singletons it can in fact be applied to non-singleton objects too, by also employing the *prototype* pattern [GHJV95]. In this way the *registry of singletons* becomes a *registry of prototypes*. In order to make an object a prototype for this purpose we must also add a method to its extracted interface (e.g., `newInstance`) that returns a new instance of the interface.

5.6.2 Results

We used our refactoring strategy on Azureus. Since Azureus has a variety of ways of implementing singletons e.g. the getInstance-method style, having all static fields, having all static methods we identified singletons manually from the list of candidates partially shown in Table 2.



Class	In-degree	CRSS
. . . util.Debug	279	1372
. . . util.AEMonitor	168	1372
. . . config.COConfigurationManager	164	1372
. . . internat.MessageText	135	1372
. . . util.Constants	129	0
. . . download.DownloadManager	110	1372
. . . ui.tables.TableCell	110	7
. . . ui.tables.TableCellRefreshListener	108	7
. . . logging.LGLogger	107	1372
. . . bouncycastle.asn1.DERObject	103	3
...	...	...

Table 2: Candidates for Extract Interface Refactoring

The classes that we actually refactored were `LGLogger` (1), `COConfigurationManager` (2), `Debug` (3), `FileUtil` (4), `PlatformManager` (5), `MessageText` (6), `TorrentUtils` (7), `LocaleUtil` (8), `DisplayFormatters` (9), `Direct- ByteBufferPool` (10). The effect of these refactorings on the CRSS distribution are shown in Figure 9. The axis going 'into' the page has numbers that correspond to the extract interface operations on the listed classes. Each refactoring improved the distribution of CRSS as expected and after the 10th refactoring only 400 classes had CRSS values of 1300 or more and nearly 1300 classes now transitively depended on less than 100 other classes. The effects on the subsystems we identified earlier are shown in the 'CRSS-refact' column of Table 1. In the cases where the refactored class was the key class in the subsystem (i.e. `Debug`, `COConfigurationManager`, `MessageText` and `LGLogger`) we show the CRSS value for the implementation, not the extracted interface since the former has the larger CRSS value. Indeed an inspection of the reachability sets of these subsystems now shows that they are no longer mutually dependent so could feasibly be arranged into packages without cycles in the PDG.

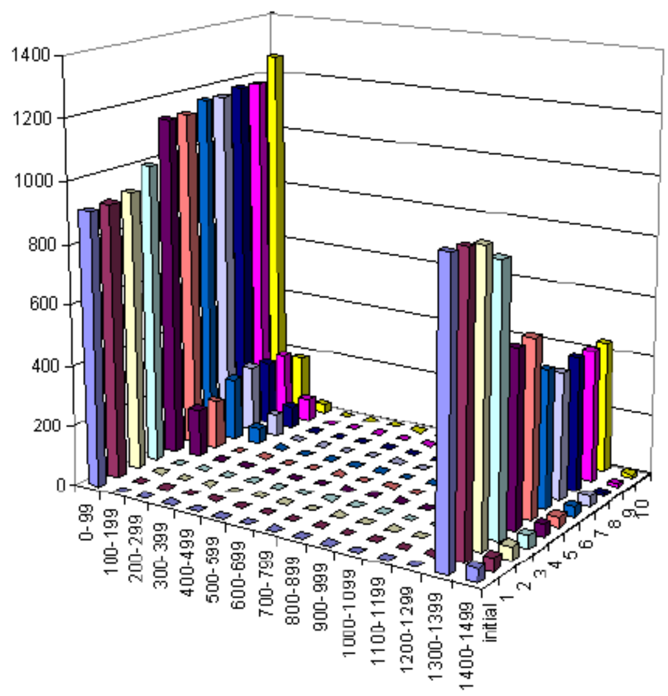


Figure 9: Refactoring Azureus

5.7 Related Work

The work in this paper extends a prior work [MT06] and is also related to work we have done looking at dependency cycles among classes in Java software [MT07b]. Here we review other work that has been done in metrics for package design. Hautus[Hau02] , Lakos [Lak96] and Ducasse et al. [DLP05] have each produced literature on this topic.

5.7.1 Hautus

The design principle stating *a PDG should be a directed acyclic graph* itself implies a simple metric. This metric classifies a given PDG as being either *cyclic* or *acyclic*. Unfortunately this metric is of little practical use, because we want to know the degree to which a cyclic PDG is cyclic. In this way we can estimate the amount of work required to make it acyclic, or determine if a refactoring has made it more or less cyclic. Hautus's PASTA (PAckage STructure Analysis) metric aims to measure the degree of 'cyclicness' in a PDG.

The PASTA metric is defined for a given package as "the weight of the undesirable dependencies between the sub packages divided by the total weight of the dependencies between the sub packages" [Hau02]. The *weight* of a dependency is defined as "the number of references from one package to another". Hautus does not make clear what constitutes a single reference — for instance references can be counted at the level of classes so that a class can reference another at most once, or at the level of identifiers in the source code of a class so that a class can reference another multiple times. The *undesirable* dependencies are defined as a set of dependencies that when removed lead to an acyclic graph. Since there are multiple sets of dependencies that can be removed to lead to an acyclic graph the set is chosen such that it has the minimal weighted sum of references.

As Hautus's metric is stated above it applies to a subgraph of a given PDG. The subgraph is chosen such that all its vertices are children of a given package in the package tree. In order to apply give the PASTA metric a single value for a whole program, rather than a single package, Hautus defines the PASTA metric for a whole program as "the weight of all desirable dependencies in all package divided by the total weight of the dependencies in all packages". This means that some references are counted multiple times since it is the underlying subpackage dependencies that gives a package its dependencies. Hautus states that this effect, of counting some references multiple times, is deliberate because it means that packages at a higher level of abstraction have a greater impact on the metric than those at a lower level of abstraction. Hautus then claims that it is more important to remove cycles between packages at a high-level of abstraction than cycles between packages at lower levels of abstraction.

Hautus's metric differs from our CRSS metric in that it purports only to measure the 'cyclicness' of a PDG. This relates only to the single design principle that a PDG should be acyclic. We have argued that our metric is useful for indicating violations of other metrics, particularly *stand-alone* and *manageable size*.

Hautus has also produced a tool to collect his metric and support refactoring to eliminate cycles between packages. It appears that Hautus's refactoring technique implicitly assumes that classes are correctly partitioned into packages and correspondingly that the way to remove cycles is to break dependencies between classes. This may not be a good assumption because repartitioning classes into a new package structure (especially with the support provided by Eclipse) is a far simpler operation than breaking dependencies between classes. Furthermore Martin claims that package design should be a bottom-up process whereby the class relationships dictate the formation of packages[Mar96b]. Based on this statement it may be possible to use our CRSS metric as a starting point for determining how classes should be partitioned into packages.

5.7.2 Lakos

Lakos has identified several metrics for package design quality. The simplest of Lakos's metrics is *Cumulative Component Dependency (CCD)*. CCD is the sums of the reachability set sizes for all the nodes in given PDG [Lak96, p.187]. Lakos also proposes and *average* and *normalised*- version of this metric. We will discuss the average version. Average Component Dependency is ACD for a given PDG is CCD divided by the number of nodes in that graph.

Tall or cyclic PDGs will tend to have a higher value ACD than flatter, acyclic PDGs with stand-alone components [Lak96, p.195]. In this way ACD is useful for determining the degree to which a PDG follows the acyclic and flat package desing principles. However, it does not take into account the size of a package so cannot be used to measure conformance to the manageable-size principle. It also deals with packages rather than classes so suffers from the same problem as Hautus'.

5.7.3 Ducasse

Ducasse et al. introduce a number of metrics that could be used for measuring the package design quality, though these metrics are dicussed in the context of reverse engineering a system[DLP05]. In particular their paper concentrates on collecting metrics that can be used in visualisations of different types of dependencies between packages so the relationships between these packages can be more quickly and easily understood by a developer new to a system. Metrics from Ducasse et al. that could be useful for measuring package design quality are Number of Provider Packages (PP), Number of Client Packages (CC), Number of Class Clients (NCC) and Number of Classes in a Package (NCP). PP and CC correspond to the outdegree and indegree respectively of a package in a PDG. These could be used to indicate if a package was stand-alone or alternately excessively coupled to other packages. NCC could be used similarly – if many classes depend on a given package it may indicate that these classes packages are excessively coupled and not stand-alone. NCP could be used to indicate if packages were of a manageable size.

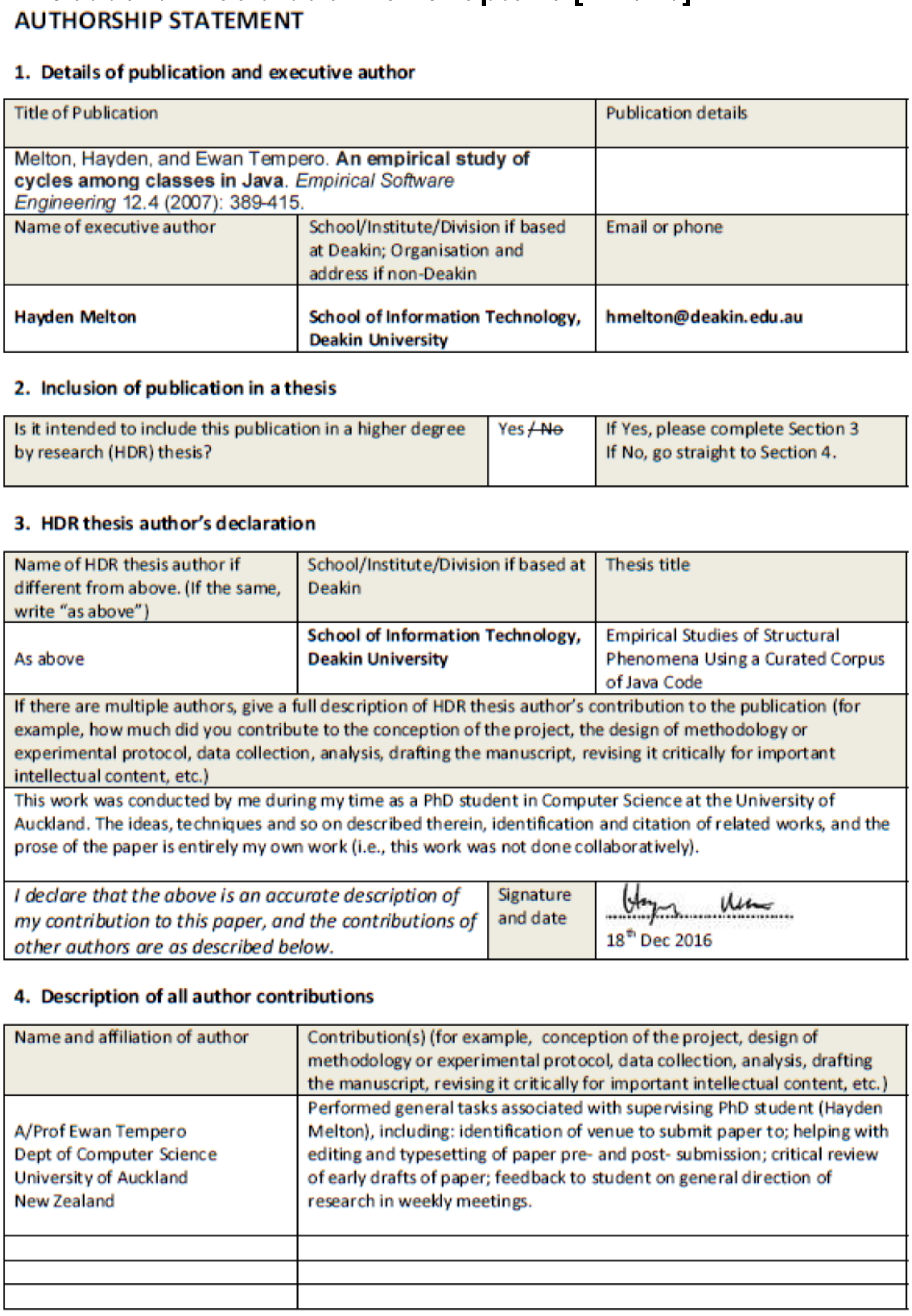
5.8 Conclusions

Package design is believed to have an important effect on reusability and testability, as well as other quality attributes. It is therefore useful to know if the relationships between classes in a system preclude it from having packages that are stand-alone and of a manageable size. In this respect we have developed a simple metric, CRSS, that can be used to identify systems whose packages cannot be stand-alone and of a manageable size.

One distiguishing feature of our metric is that it is for *whole program* analysis—not just for individual elements of a program. Indeed it is the distribution of CRSS values for all the classes defined in the source files of a system that tells us about its best potential package structure.

We have presented empirical studies based on a number of open-source systems that identify distributions of CRSS that are indicative of package designs that cannot comprise packages that are stand-alone and of a manageable size. In order to improve the potential package structure of these systems we have shown how our CRSS metric can be used to identify good candidates for the *extract interface* refactoring. This refactoring can improve the relationships between classes in a system with respect to its potential for a good package structure.

Coauthor Declaration for Chapter 6 [MT07b]



AUTHORSHIP STATEMENT

1. Details of publication and executive author

Title of Publication		Publication details
Melton, Hayden, and Ewan Tempero. **An empirical study of cycles among classes in Java**. *Empirical Software Engineering* 12.4 (2007): 389-415.		
Name of executive author	School/Institute/Division if based at Deakin; Organisation and address if non-Deakin	Email or phone
Hayden Melton	**School of Information Technology, Deakin University**	**hmelton@deakin.edu.au**

2. Inclusion of publication in a thesis

Is it intended to include this publication in a higher degree by research (HDR) thesis?	Yes ~~/ No~~	If Yes, please complete Section 3 If No, go straight to Section 4.

3. HDR thesis author's declaration

Name of HDR thesis author if different from above. (If the same, write "as above")	School/Institute/Division if based at Deakin	Thesis title
As above	**School of Information Technology, Deakin University**	Empirical Studies of Structural Phenomena Using a Curated Corpus of Java Code
If there are multiple authors, give a full description of HDR thesis author's contribution to the publication (for example, how much did you contribute to the conception of the project, the design of methodology or experimental protocol, data collection, analysis, drafting the manuscript, revising it critically for important intellectual content, etc.)		
This work was conducted by me during my time as a PhD student in Computer Science at the University of Auckland. The ideas, techniques and so on described therein, identification and citation of related works, and the prose of the paper is entirely my own work (i.e., this work was not done collaboratively).		
I declare that the above is an accurate description of my contribution to this paper, and the contributions of other authors are as described below.	Signature and date	[illegible] 18th Dec 2016

4. Description of all author contributions

Name and affiliation of author	Contribution(s) (for example, conception of the project, design of methodology or experimental protocol, data collection, analysis, drafting the manuscript, revising it critically for important intellectual content, etc.)
A/Prof Ewan Tempero Dept of Computer Science University of Auckland New Zealand	Performed general tasks associated with supervising PhD student (Hayden Melton), including: identification of venue to submit paper to; helping with editing and typesetting of paper pre- and post- submission; critical review of early drafts of paper; feedback to student on general direction of research in weekly meetings.

5. Author Declarations

I agree to be named as one of the authors of this work, and confirm:

i. that I have met the authorship criteria set out in the Deakin University Research Conduct Policy,

ii. that there are no other authors according to these criteria,

iii. that the description in Section 4 of my contribution(s) to this publication is accurate,

iv. that the data on which these findings are based are stored as set out in Section 7 below.

If this work is to form part of an HDR thesis as described in Sections 2 and 3, I further

v. consent to the incorporation of the publication into the candidate's HDR thesis submitted to Deakin University and, if the higher degree is awarded, the subsequent publication of the thesis by the university (subject to relevant Copyright provisions).

Name of author	Signature*	Date
Ewan Tempero	[illegible]	16/12/2016

6. Other contributor declarations

I agree to be named as a non-author contributor to this work.

Name and affiliation of contributor	Contribution	Signature* and date

* If an author or contributor is unavailable or otherwise unable to sign the statement of authorship, the Head of Academic Unit may sign on their behalf, noting the reason for their unavailability, provided there is no evidence to suggest that the person would object to being named as author

7. Data storage

The original data for this project are stored in the following locations. (The locations must be within an appropriate institutional setting. If the executive author is a Deakin staff member and data are stored outside Deakin University, permission for this must be given by the Head of Academic Unit within which the executive author is based.)

Data format	Storage Location	Date lodged	Name of custodian if other than the executive author
Qualitas Corpus of Java code, both binaries and source code with metadata (see *http://qualitascorpus.com/*)	University of Auckland, New Zealand	Starting 2006, onward	Ewan Tempero

This form must be retained by the executive author, within the school or institute in which they are based.

If the publication is to be included as part of an HDR thesis, a copy of this form must be included in the thesis with the publication.

Chapter 6 An Empirical Study of Cycles among Classes in Java

Advocates of the design principle *avoid cyclic dependencies among modules* have argued that cycles are detrimental to software quality attributes such as understandability, testability, reusability, buildability and maintainability, yet folklore suggests such cycles are common in real object-oriented systems. In this paper we present the first significant empirical study of cycles among the classes of 78 open- and closed-source Java applications. We find that, of the applications comprising enough classes to support such a cycle, about 45% have a cycle involving at least 100 classes and around 10% have a cycle involving at least 1,000 classes. We present further empirical evidence to support the contention these cycles are *not* due to intrinsic interdependencies between particular classes in a domain. Finally, we attempt to gauge the strength of connection among the classes in a cycle using the concept of a *minimum edge feedback set*.

6.1 Introduction

There is a plethora of literature describing how software systems should be structured (e.g. Booch [Boo91];Dijkstra [Dij68]; Lakos [Lak96]; Parnas [Par72]; Stevens et al. [SCM74]). We are interested in determining the extent to which such advice is followed by practitioners of software engineering [Mel06]. Casual observations made by luminaries such as Foote and Yoder [FY97]; Parnas [Par96]; Wirth [Wir95] and Szyperski [SGM02, p.40] would suggest that it is not widely-followed. If this is true, then it implies either there is a lot of bad software out there, or the advice itself is not useful. Either implication is of concern to software engineering researchers. However we cannot rely solely on casual observation. We need empirical evidence to support any claim that design advice is generally not being followed. In this paper, we present an empirical study examining the use of the design principle *avoid dependency cycles among modules*.

We'd like to know the extent to which "avoid cycles" is followed because its advocates have argued that dependency cycles are detrimental to many software quality attributes, including understandability, testability, reusability, buildability and maintainability (Kung et al. [KGH+95b]; Lakos [Lak96]; Martin[Mar96b]; Parnas [Par96]). Despite this purported detriment, folklore would suggest that this principle is not widely-followed: it has been stated (Briand et al.[BLW03]; Hashim et al.[HSR05]; Kung et al.[KGH+93]; Winter[Win98]; Lakos [Lak96, p.3]) and implied (Binder [Bin99]; Jungmayr [Jun02]; Martin [Mar96b]) that dependency cycles among the classes of Object-Oriented (OO) software systems are common.

To date empirical evidence of the extent to which cycles pervade OO systems is somewhat lacking. It rests on differing metrics collected from a handful of mostly small Java ([BLW03][HSR05][Hau02]) and C++ [KGH+95b] applications. To the best of our knowledge there has been no large-scale empirical study published of dependency cycles in OO software. The main contribution of this paper is thus a detailed empirical study of dependency cycles among classes across 78 open- and closed-source Java applications. We focus on Java because it is widely used and there is a significant amount of Java software generally available.

The remainder of this paper is organised as follows. In Section 2 we motivate the study by discussing the ways in which cycles among a program's organisational units purportedly affect specific quality attributes. We identify the types of dependency and cycle to which the principle applies. In Section 3 we discuss the method by which we conducted our empirical study into the prevalence of cyclic dependencies among Java classes. In Section 4 we present the results of our study of cycles among the classes of Java applications in a software corpus. In Section 5 we discuss the implications of our findings. We draw conclusions and summarise our findings in Section 6.

6.2 Motivation

Our motivation for studying dependency cycles in code comes from the amount of advice that has been given to avoid them. In this section we review the origins of "avoid cycles" and related design principles and present the arguments that have been made on the effect cycles have on specific software quality attributes. Finally we formalise the notions of cycle and dependency applicable to this study.

To the best of our knowledge Parnas was the first to discuss the design principle *avoid cycles among modules* [Par78][12]. Parnas argued that when two modules are cyclically dependent (i.e., each calls routines declared in the other) neither can be tested until both are “present and working.” The consequence then, of long cycles involving many modules, according to Parnas, is that “one may end up with a system in which nothing works until everything works.”

Over the years there have been other design principles proposed that also have the effect of avoiding (or reducing) dependency cycles. Stevens et al. state the design principle *minimise coupling between modules* Stevens et al.[SMC74]. A design with dependency cycles has higher coupling than its acyclic analog (e.g., if modules A and B are in a cycle then B has higher coupling than if the only dependency is from A on B). Riel states “Derived classes must have knowledge of their base class by definition, but base classes should not know anything about their derived classes” [Rie96, p.81]. Disallowing the dependency of a base classes on its derived classes prevents a dependency cycle between the base and derived classes. Riel also states that “In applications that consist of an object-oriented model interacting with a user interface, the model should never be dependent on the interface” [Rie96, p.36]. This has the effect of eliminating cycles between the model and user interface because the user interface normally depends on the model in any case. Finally Booch [Boo95] says “. . . all well structured object-oriented architectures have clearly defined layers.” Long dependency cycles have the potential to encompass several layers. When this is the case layers are not “clearly defined” because a layer should only depend on the layers below it.

6.2.1 Cycles among Classes

Many have discussed cycles in the context of the OO paradigm, but the most comprehensive discussion is by Lakos. Lakos states that cyclic dependencies among the *components* of a C++ program inhibit understanding, testing and reuse [Lak96, p.185]. Lakos’ notion of a C++ *component* is roughly equivalent to a .java file in

[12] Dijkstra (1968) argued for a hierarchical structure more than 10 years prior to Parnas’ discussion of cycles, but whether we can interpret Dijkstra’s argument as the origin of the *avoid cycles* design principle is somewhat contentious. After all, there exist structures that are not *strictly* hierarchical as Dijkstra uses the term, but that are acyclic nonetheless (see e.g., Szyperski 2002, p.162).

Java. Since there is typically one top-level class per source file in Java we will, for the purposes of this section, discuss cycles in terms of (top-level Java) classes.

Understanding. Lakos and Fowler have both argued that cycles are detrimental to understanding. Lakos says that in a cyclically dependent system there is no reasonable starting point and no piece of the system that can make sense on its own [Lak96, p.3]. Fowler says cyclically dependent systems are harder to understand "because you have to go around the cycle many times" [Fow01]. We present a more precise argument for cycles inhibiting understanding below.

When we look at a class' source code we want to be able to understand as much of it as possible in isolation, without having to look at the source code of other classes in the system. Sometimes, however, it is difficult to understand a class in isolation. When we cannot understand a class in isolation it is often the case that we look at the source code of the classes on which it depends (e.g., calls methods on). If we restrict ourselves to understanding a class based only on the classes on which it depends, then cycles make understanding a system's classes more difficult. To see why consider the two systems depicted in Fig. 1. Both systems have two classes, X and Y, but differing dependencies (represented by arrows). In order to understand class X in Fig. 1a we (at most) need to examine the source code of X and Y. In order to understand Y we at most need only examine the source code of Y itself. Compare this now to the cyclically dependent system of Fig. 1b where the worst case for understanding Y is examining the source code of both Y and X. So the argument is that understanding the average class of the system depicted in Fig. 1b is more work, and therefore more difficult than understanding Fig. 1a.

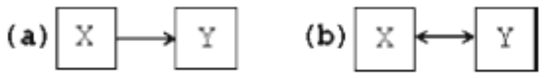


Figure 1 Classes in two, small, hypothetical systems

From the discussion about Fig. 1 we can glean that in the worst case our strategy for understanding a class is *transitive*. Consider now, class K of the system depicted in Fig. 2a. In order to understand K we may also have to look at the source code of L. But then to understand L we may also have to look at the source code of M. Since M depends on nothing we should always be able to understand it in isolation. So in Fig. 2a the worst case for understanding class I involves looking at the source code for itself and for J, K, L, M. For class J the worst case is looking at the source code of itself and classes K, L and M. The process continues similarly for classes K, L and M. So we may conclude that the average worst case for understanding a arbitrary class from the system of Fig. 2a involves looking at the source code of 3 classes (=(5+4+3+2+1)/5). Compare this now to the system of Fig. 2b. In this system all five classes are involved in one big cycle, so in order understand any single class we may have to examine the other 4 classes in the system too. Clearly this is worse than the average worst case of Fig. 2a.

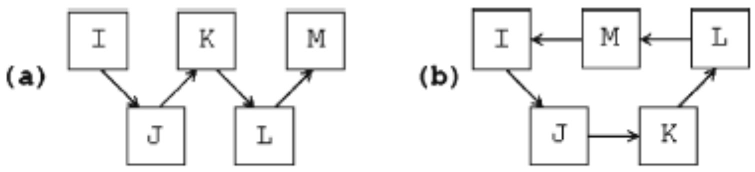


Figure 2 Classes in two, larger, hypothetical systems

Testing. There are strong arguments that cyclic dependencies among classes inhibit *testing in isolation* [Lak96, p.161–174] and *integration testing* [Lak96, p.174–187][BLW03][HSR05][KGH+95b][Bin99, p.983–984]. In terms of testing a class in isolation, if two classes are cyclically dependent then it is impossible to test either one without the other [Lak96, p.161–174]. If there are many cyclically dependent classes in a system then none of the classes in the cycle can be tested truly independently (in isolation) from the others. Cyclic dependencies impede *integration testing* by preventing a topological ordering of classes that can be used as a test order [BLW03][KGH+95b][Lak96]. Many researchers have dealt with the problem of cyclic dependencies among classes in integration testing by breaking cycles through the creation of stubs [BLW03][KGH+95b]. Binder argues that stubs can be problematic for a number of reasons [Bin99, p.983] and cites several works advocating the outright elimination of cycles from the design of a system.

Reuse. Lakos [Lak96] and Martin [Mar96b] have argued that cycles inhibit the verbatim reuse of source code. In this form of reuse source files are copied from one program to another, without (1) having to modify the textual content of these source files in order to make them compile in the new environment and (2) introduce stubs for classes on which they depend. The crux of the argument is that we want to be able to copy as few source files from one application to the other so we can (1) reduce the compilation time of our application, (2) reduce the potential search space if we need to thoroughly understand the functionality (through inspection of the source) provided by the copied files (3) reduce the amount of code that could potentially contain bugs.

6.2.2 Cycles among Packages

The *package* construct is a feature of some programming languages (such as Java, Ada and C++) that allows the subsystem to which a class belongs to be reflected in its source code. Dependency cycles among classes can cause dependency cycles among packages. To see why consider one desirable property of a package—that it is of *manageable size* [Lak96, p.481][Mey95, p.51]. Meyer states that for a package to be of manageable size it should contain 5–40 classes and, more fundamentally, it should be able to be developed by 1–4 people and entirely understood by a single person [Mey95, p.51]. If there is a cycle involving more cycles than the maximum size of a package, then there must be a cycle in the package structure as well.

Lakos [Lak96, p.494], Martin [Mar96b] and Fowler [Fow01] state that there should be no dependency cycles among the packages of an application. They claim cycles among packages inhibit *Production* [Mar96b][Lak96,p.512–514], *Marketing* [Lak96, p.494], *Development* [Lak96, p.494][Mar96b] and *Usability* [Lak96, p.495][Mar96b].

6.2.3 Meaning of Dependency

In this paper we are interested in the meaning of dependency as it applies to the design principle *avoid dependency cycles among modules*. If we apply this principle to Java in the way Lakos applies it to C++ then we should avoid dependency cycles among an application's .java source files. (Previously we referred to these .java files simply as classes, using the term *class* to refer to a top-level class that includes all the nested classes it lexically encloses.) Again, deriving from Lakos, the type of dependency to which the principle applies is a static or compilation dependency (cf. dynamic or runtime dependencies). We adapt the dependency relations proposed by Lakos to Java below.

For a Java source file A:

· uses(A) is the set of all .java files that declare types that A refers to in its text. We do not include dependencies in this set (or any of the other sets below) that are due to redundant imports because these are superficial and good tool support already exists to remove these (e.g., Eclipse's "clean imports" feature).

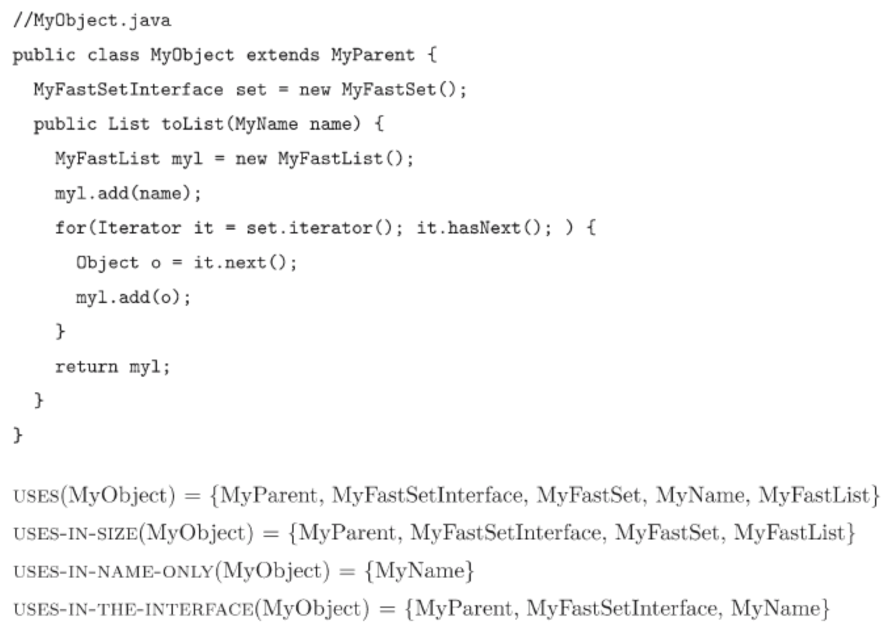
· uses-in-size(A) is the set of all .java files that declare types on which methods are called and fields are accessed in the text of A. Constructor invocations are considered as method calls and supertypes of types declared in A are also included in this set. This set is related to that used to compute the CBO metric [BDW99][CK91].

· uses-in-name-only(A) = uses(A) \ uses-in-size(A), that is, those types that are referred to in A's text, but on which no methods are called or fields are accessed.

· uses-in-the-interface(A) is the set of all .java files that declare types that appear in the interface of A. By *interface* we mean the methods and fields that A declares that are accessible from source files other than A i.e, all fields and methods that are *not* declared private. So this set includes types that appear in the return type, formal arguments and thrown exceptions of non-private methods, and the declared types of non-private fields. We also include the direct supertypes of the class so that the transitive form of uses-in-the-interface includes all the methods and fields that can be called on it (even those types that appear solely in supertypes).

We show the computation of the above relations for the source file of Fig. 3. Note that List, Iterator and Object correspond to classes in the Java API so do not appear in any of the relations' sets. We ignore types that are declared in external libraries (e.g., the Java API) because application developers have no control over cycles among these types, and these types are assumed to be tested and correct. All the other types used in Fig. 3 are assumed to be declared in the application's source files.

We will use all these different dependency relations to help distinguish "bad" or "unnecessary" cycles from those that cannot be sensibly avoided. Lakos argues that some cycles cannot be sensibly avoided due to *intrinsic interdependency* between the real world objects the classes model [Lak96, p.213]. Lakos illustrates intrinsic interdependency with a graph modelled with Node and Edge classes. Conceptually an edge exists to connect a source node to a destination node. It is therefore likely that a client of the Edge class would be interested in the nodes this edge connects. Thus Edge provides methods getSrcNode and getDstNode that cause a Edge to depend on Node. Clients of Node are likely to have an analogous requirement meaning Node depends on Edge. Thus the conceptual relationships between edges and nodes in a graph have caused Edge and Node to be cyclically dependent.



```
//MyObject.java
public class MyObject extends MyParent {
  MyFastSetInterface set = new MyFastSet();
  public List toList(MyName name) {
    MyFastList myl = new MyFastList();
    myl.add(name);
    for(Iterator it = set.iterator(); it.hasNext(); ) {
      Object o = it.next();
      myl.add(o);
    }
    return myl;
  }
}
```

USES(MyObject) = {MyParent, MyFastSetInterface, MyFastSet, MyName, MyFastList}
USES-IN-SIZE(MyObject) = {MyParent, MyFastSetInterface, MyFastSet, MyFastList}
USES-IN-NAME-ONLY(MyObject) = {MyName}
USES-IN-THE-INTERFACE(MyObject) = {MyParent, MyFastSetInterface, MyName}

Figure 3 Computation of the four dependency relations on a simple Java class

Intrinsic interdependencies cannot be sensibly avoided and, according to Lakos, generally manifest themselves as cycles in the uses-in-the-interface relation. In Section 4 we will use cycles in the uses-in-the-interface relation to give an approximate upper bound on cycles due to intrinsic interdependency. We point out that this is only an approximation of cycles due to intrinsic interdependency because types may inappropriately appear in the interfaces of classes due to poor design decisions (e.g., a class from the view of the application appears in a model class violating a well-known object-oriented design heuristics[Rie96]).

The uses-in-name-only relation is important because Lakos argues that a source file can be tested, understood and reused independently of any types it transitively uses but does not transitively uses-in-size (recall uses-in-name-only is defined as uses \

uses-in-size) [Lak96, p.247–256]. We should then be most concerned about cycles in the uses-in-size relation with respect to testing, reuse and understanding. As for the uses relation, this is the dependency we care about with respect to package design—long cycles in this relation can cause cycles among packages (see Section 2.2).

6.2.4 Meaning of Cycle

There are many types of cycles. A *simple cycle*, for instance, is a path with no repeated vertices that starts and ends on the same vertex. The types of cycle in which we are interested are *Strongly Connected Components* (SCCs). A SCC is defined as a subgraph of a directed graph induced by a maximal set of mutually reachable vertices [GY04, p.128]. SCCs are the type of cycle most applicable to our study because all the nodes in a SCC are all cyclically dependent on one another. Additionally, SCCs provide a higher-level view of simple cycles in a system because a SCC must comprise at least one simple cycle, and all nodes in a given simple cycle must appear in the same SCC. In our data a SCC is usually a conglomeration of many simple cycles interacting in complex, intertwined ways.

In order to illustrate the notion of a SCC consider the directed graph of Fig. 4. The vertex sets for the SCCs comprising more than 1 vertex in this graph are: {A, B, C, D} and {F, G} and {H, I, J}. For the analysis performed in this paper we represent .java source files as vertices and one of the uses, uses-in-size and uses-in-the-interface relations as edges. We then measure cycle size in terms of the size of each SCC's vertex set.

In order to determine the extent to which cycles pervade OO systems we ought to measure more than just the size of SCCs. This is because the strength of connection among the nodes in a SCC can vary greatly. Consider the 3 SCCs of Fig. 5. Each SCC comprises 5 vertices but intuitively the strength of connection among the nodes in each of the SCCs varies. In the graph of Fig. 5a we can break all cycles by removing just one edge. In the graph of Fig. 5b we need to remove at least 5 (logical) edges to break all cycles. For the graph of Fig. 5c the minimum number of edges we need to remove in order to break all cycles is less obvious, but it is certainly more than 5. If these SCCs represented source file dependency graphs of three different systems we would likely be most concerned about the structure of the system in Fig. 5c with respect to *avoid dependency cycles* because the breaking all its cycles would involve breaking the most dependencies among the source files.

In graph theory a *Minimum Edge Feedback Set* (MEFS) is the smallest set of edges we need to remove from a graph to break all cycles in it. Skiena notes that determining a MEFS for a graph is a NP-complete problem [Ski98], which helps explain our difficulty in determining a MEFS for the graph of Fig. 5c. Fortunately Eades et al. [ELS93] present a good heuristic (referred to herein as Eades' Heuristic) for computing an edge feedback set that is close to minimal. We refer to a close-to-minimal edge feedback set as *mEFS*. We do not restate this heuristic in its entirety but in essence it greedily produces a vertex sequence (v_1, v_2, ..., v_n) that represents a topological ordering. The mEFS is then the set of all of the edges that go from right-to-left in the vertex sequence. In our study we adapt Eades' heuristic to take into account some constraints in our problem domain. We then use the size of the mEFS as a measure of the strength of connection among cyclically dependent source files.

6.3 Methodology

In order to measure cycles among classes in object-oriented software systems we conglomerated a corpus of Java software and built tools to infer dependencies from source code and byte code. In this section we describe our Java corpus and the tools we used extract dependencies from Java byte code and source. We also discuss our adaptation of Eades' Heuristic.

6.3.1 Corpus

The applications used in our study are given in the Appendix. They were chosen to vary along several dimensions: domain, size, origin and open/closed source. The values of these attributes is given in the table in the Appendix: "#Classes" is size measured in terms of number of .java source files; "O/C" indicates whether the software is open- or closed-source; and "V" indicates whether we have multiple versions (releases) of the software—this is a interesting feature of our corpus, we have multiple versions of 22 of the 78 applications in it. "Origin" is the organization or website from which we obtained the software. Some organisation and application names have been obfuscated with the letters A-H because intellectual property agreements mean we cannot identify them.

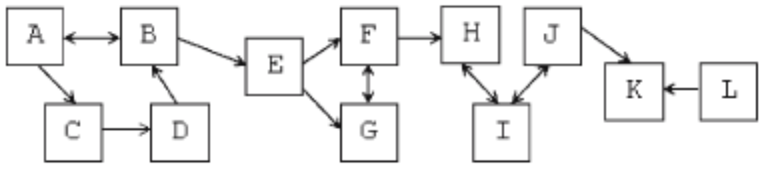


Figure 4 Graph with SCCs

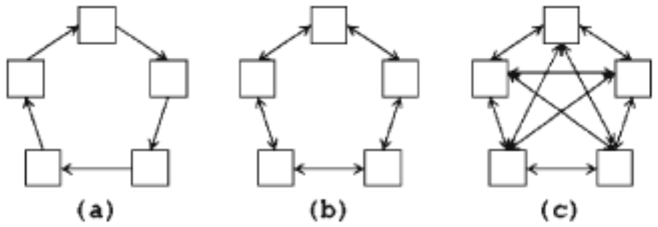


Figure 5 Three SCCs each with a differing strength of connection among its nodes

In collecting software for our corpus we first amalgamated corpora used in other published papers [GM05][GPV01]. All accessible applications from these existing corpora were added to ours. Further applications were then added to the corpus based on software that we were familiar with (e.g. Azureus, ArgoUML, Eclipse, NetBeans). Finally we identified popular (widely downloaded) and actively developed open-source Java applications from various websites, including: developerWorks[13], SourceForge[14], Freshmeat[15], Java.net[16], Open Source Software In Java[17] and The Apache Software Foundation[18].

6.3.2 Tools

We built two tools to infer dependencies between Java classes. One operates on Java source code and the other on byte code. The tool that operates on source code is described in detail in an earlier paper[MT06]. At the time we analysed company B's

[13] http://www-128.ibm.com/developerworks/views/java/downloads.jsp
[14] http://sourceforge.net/
[15] http://freshmeat.net/
[16] http://community.java.net/projects/
[17] http://java-source.net/
[18] http://apache.org/

software the source code tool was the only one we had and it did not have the capability to compute the uses-in-size relation. Correspondingly the data pertaining to this relation is missing for the applications originating from company B. Due to licensing restrictions on the Java parser this tool utilises it is not publically available.

The second tool we developed infers dependencies from Java byte code. It is built on top of the Byte Code Engineering Library (BCEL)[19]. We used it to infer dependencies for all the remaining applications in the corpus. This tool (including source) is available for download[20]. Briefly, the tool examines the entries in a compiled class' constant_pool table (see [LY99, ch.4]). The entries in this constant pool are often a subset of that class' source code dependencies. This is because some type information can be thrown away during compilation such as the declared types of local variables and the types referred to by redundant imports. Additionally dependencies may be lost if public static final fields declared as primitive types are inlined by Java compiler (see the problem of "inconstant constants" for a more detailed discussion of this Gosling et al. [Gos00, ch.13]). The potential loss of dependencies by analysing byte code means our results for all the applications not originating from company B are actually a lower limit on the cycles among classes in the uses-in-size and uses relations.

6.3.3 Computing a mEFS

In Section 2.4 we noted that SCC size alone was not enough to characterise the extent to which cycles pervade a system, and so we also need to gauge strength of connection among the nodes in a SCC with mEFS size. We have adapted Eades' Heuristic to determine a mEFS more suitable for our problem domain. Our adaptation of Eades' Heuristic ensures that edges due to a dependency of a derived type on its supertype do not appear in the mEFS. This is because re-factoring to break such relationships is more difficult than breaking other types of relationships [Jun02].

[19] http://jakarta.apache.org/bcel/
[20] http://www.cs.auckland.ac.nz/~hayden/

Our adaptation of Eades' Heuristic involves performing a stable sort on the vertex sequence so that supertypes appear to the left of their subtypes in the sequence. It is always possible to sort in this way because there can be no cycles in the supersubtype relationship. Since Eades' Heuristic is non-deterministic—there are no rules to discriminate the order nodes in the vertex sequence that equally meet the greedy criterion—our adaptation of the heuristic is also non-deterministic. In order to assure a good mEFS for our results we ran it 100 times over each SCC, each time forcing different permutations inherent in the construction of the vertex sequence as allowed by Eades' Heuristic. We then took the smallest mEFS for our results.

6.4 Results

In this section we present the data we have collected on cyclic dependencies among the classes of applications in our corpus. We begin with an overview of results showing the proportion of applications in the corpus that have SCCs of growing sizes in the dependency graphs of each of the relations (uses, uses-in-size and uses-in-the-interface). We then show the break down of an application's classes into SCCs of varying sizes (again for each of the relations) for the latest version of each application in the corpus. Next we show, again using a breakdown of classes into SCCs, how the cyclic dependencies change over an application's version history for the 22 applications of which we have multiple versions. Finally we show the mEFS for the largest SCCs in each of the applications and their versions.

6.4.1 Overview

The plot in Fig. 6 provides the highest level view of the prevalence of cyclic dependencies among classes in Java software. The *x*-axis of this plot represents size in number of classes and the *y*-axis represents the proportion of applications in the corpus that are of *at least* this size for the top-most data series ("application size"), or have a SCC of *at least* this size for the bottom-three data series. So as not to bias the results towards any particular application (recall there are multiple versions of 22 of the 78 applications in the corpus) only the latest version of each application is considered in this chart. Thus the chart shows the proportion of applications with cyclic dependencies of growing sizes, and the proportion of applications of growing sizes.

We can see from the dependency relation series of Fig. 6 that:

· For the uses-in-the-interface relation about 69% of the applications in the corpus have a SCC of size >10, about 13% have a SCC >100 and no applications have a SCC >1,000. In fact, the largest SCC in this relation is 542 classes.

· For the uses relation about 85% of the applications in the corpus have a SCC of size >10, about 40% have a SCC >100 and 3% of the applications have a SCC >1,000. In fact, the largest SCC in this relation is 2,145 classes.

· For the uses-in-size relation about 81% of the applications in the corpus have a SCC of size >10, about 36% have a SCC >100 and 3%of the applications have a SCC >1,000. In fact, the largest SCC in this relation is 1,909 classes.

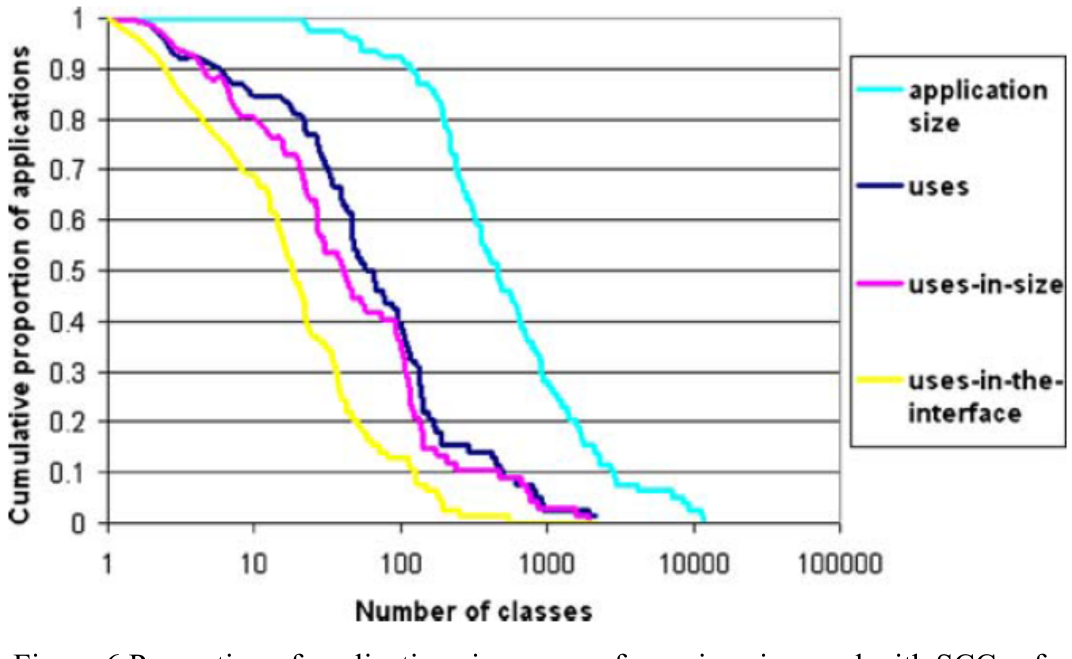


Figure 6 Proportion of applications in corpus of growing sizes and with SCCs of growing lengths on normal-log scale

The distribution of size is shown on the same plot as distribution of SCC sizes because in order for an application to have an SCC (in any of the relations) of a given size, it must comprise at least that number of classes. We can see from Fig. 6 that about 92% of the applications comprise at least 100 top-level, source-defined classes, and that 30% of the applications comprise at least 1,000 classes. So combining the information size and SCC information we can infer that about 10% (=3%/30%) of applications that are large enough to contain a SCC of size 1,000 in the uses or uses-in-size actually do contain one.

6.4.2 SCC Snapshot Data

In this section we show the breakdown of an application's classes into SCCs of growing sizes for each of the dependency relations. Again, we do this just for the latest version of each application in the corpus—applications for which we have multiple versions are dealt with in the next section. Each of the bar charts depicted in Figs. 7, 8 and 9 can be read as follows. The *x*-axis represents the application sorted alphabetically by name, and the *y*-axis represents the number of classes in SCCs. The stacked bars represent the number of classes involved in SCCs of growing sizes (>1,000, >500, >100, >50, >20, >1 and >0). Classes in the bar representing '>0' must be in SCCs of size 1, so are not cyclically dependent with any other classes.

Consider the bar for Eclipse in the chart of Fig. 7. The bottom-most bar in Eclipse's stack corresponds to classes involved in SCCs in the uses relation of size >500. The height of this bottom-most bar is about 700. This means about 700 classes are involved in SCCs of size >500. In fact, we can deduce that the largest SCC in Eclipse must be 700 from this bar because there is no way to split 700 into two parts, both >500 in size. The second bar from the bottom in Eclipse's stack corresponds to classes involved in SCCs >100 in size. This bar finishes at about 2,600 on the *y*-axis so we can infer there are about this many classes involved in SCCs >100 in size. Additionally we can infer that there are 1,900 $(= 2,600 - 700)$ classes involved in SCCs >100 and <501 in size. To determine the number of classes in Eclipse involved in cycles in the uses relation we need only determine where the >1 bar finishes (at about 5,000 for Eclipse). This means that close to half (=5,000/11,500) of Eclipse's classes are involved in cycles in the uses relation. Similar analysis can be performed on any bar-stack in the following charts.

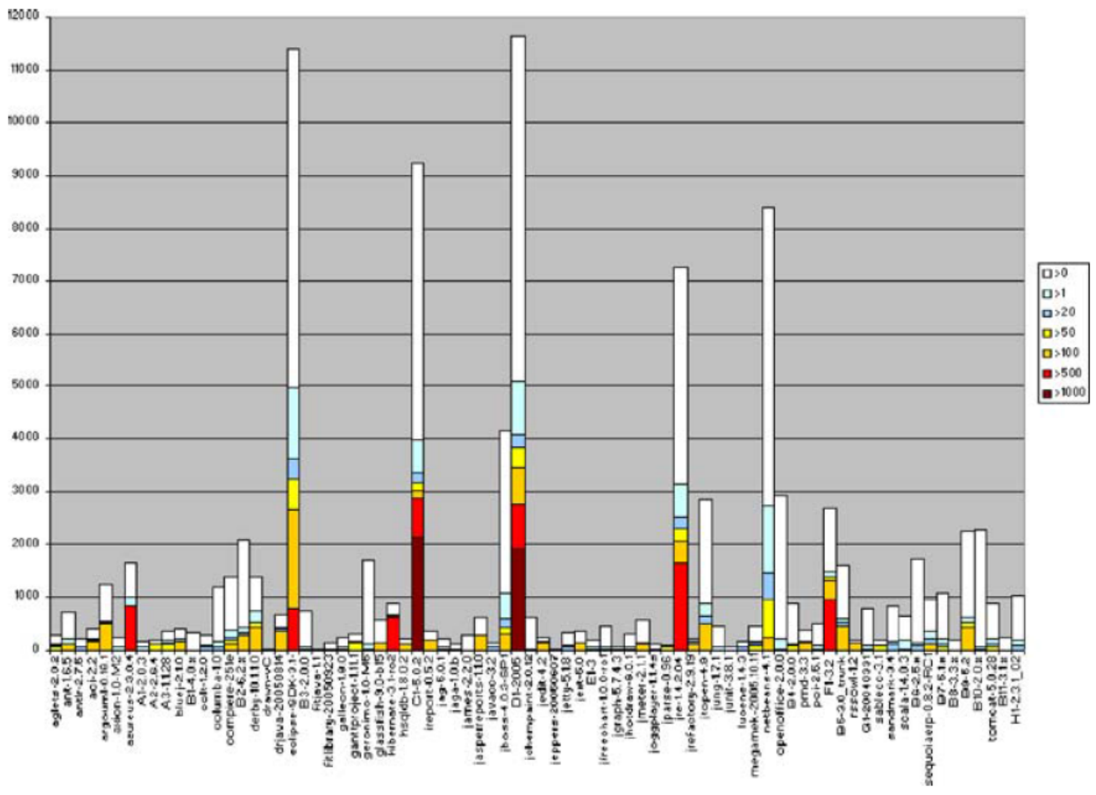


Figure 7 SCCs in uses relation over corpus

Some interesting observations from the chart of Fig. 7 are as follows:

· Applications C1-5.0.2 and D1-2005 have the largest SCCs in the uses relation—both have bars corresponding to '>1,000.' D1 must have a SCC of about 1,900 (since there is no way to split 1,900 into two parts both >1,000) and C1 has approximately 2,100 involved in SCCs >1,000 in size. Though we cannot infer from the graph that C1's SCC is of size 2,100, we checked the raw data its SCC is indeed of this size.

· Azureus and Hibernate can be singled out for having a large proportion of their classes involved in a 'large' SCC >500 in size. The chart shows that Azureus has about 1,700 classes total and about 800 of these classes are involved in a single SCC. Similarly it can be read from the plot that Hibernate comprises about 900 classes and 700 of these are involved in a single SCC.

· Eclipse versus NetBeans is an interesting comparison because both of these applications come from the same domain and provide similar functionality. Eclipse has a SCC of size 700 whereas the largest SCC in Netbeans is not >250. Indeed a lower proportion of classes are involved in cycles in Netbeans (2,700 out of 8,400) than Eclipse (5,000 out of 11,000). This provides evidence to support the view that cycles are not inherent to particular domains.

· Some application have a very small proportion of their classes involved in cycles (e.g., Columba, B3-2.0.0, James, Open Office, B6-2.5.× and B10-2.0.×). This suggests it is possible, in a practical sense, to largely avoid cycles, even in the uses relation.

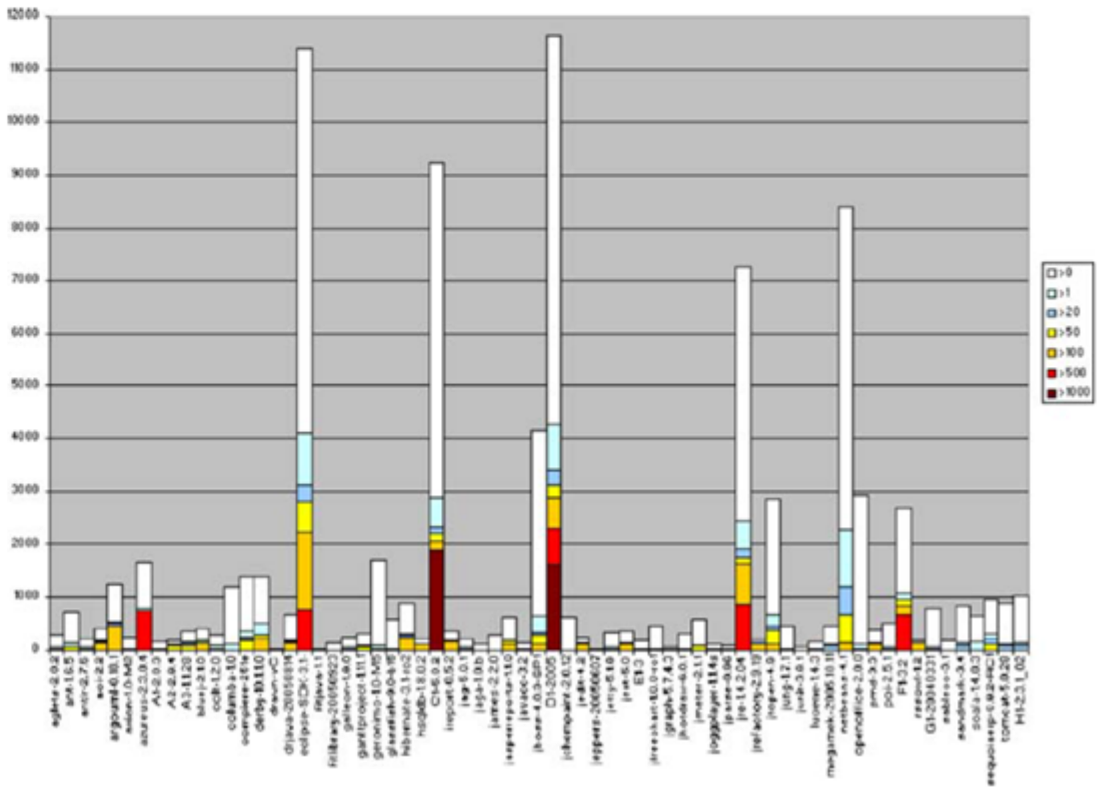


Figure 8 SCCs in uses-in-size relation over corpus

Figure 8 depicts a chart of the SCCs in the uses-in-size relation. Although it looks similar to the plot of the uses relation in Fig. 7, many of the applications show a reduction (albeit slight) in the number of classes participating in cycles. In some sense this is hardly surprising since uses-in-size(x) ⊆ uses(x), by its very definition. On the other hand it indicates that some types are not used in-size—rather they are used in-name-only.

Some interesting observations from the chart of Fig. 8 are as follows:

· Hibernate shows a significant difference between the SCCs in the uses and uses-in-size relations. In the uses relation we observed 700 was the largest SCC yet in the uses-in-size relation we observe that the largest SCC is only around 300 classes. It is tempting to say that this means that types are used in-name-only extensively in

Hibernate, however this is not a valid conclusion since one type used in-name-only in a single class can be that which breaks a large SCC.

C1-5.0.2 and D1-2005 both still contain a SCCs >1,000 in size. Many applications still contain SCCs >500, >100 and >50 in size. Again, we cannot conclude from this that types are not widely used in-name-only. We can conclude from this that types are not used effectively used in-name-only (i.e., to break SCCs in the uses relation in these applications).

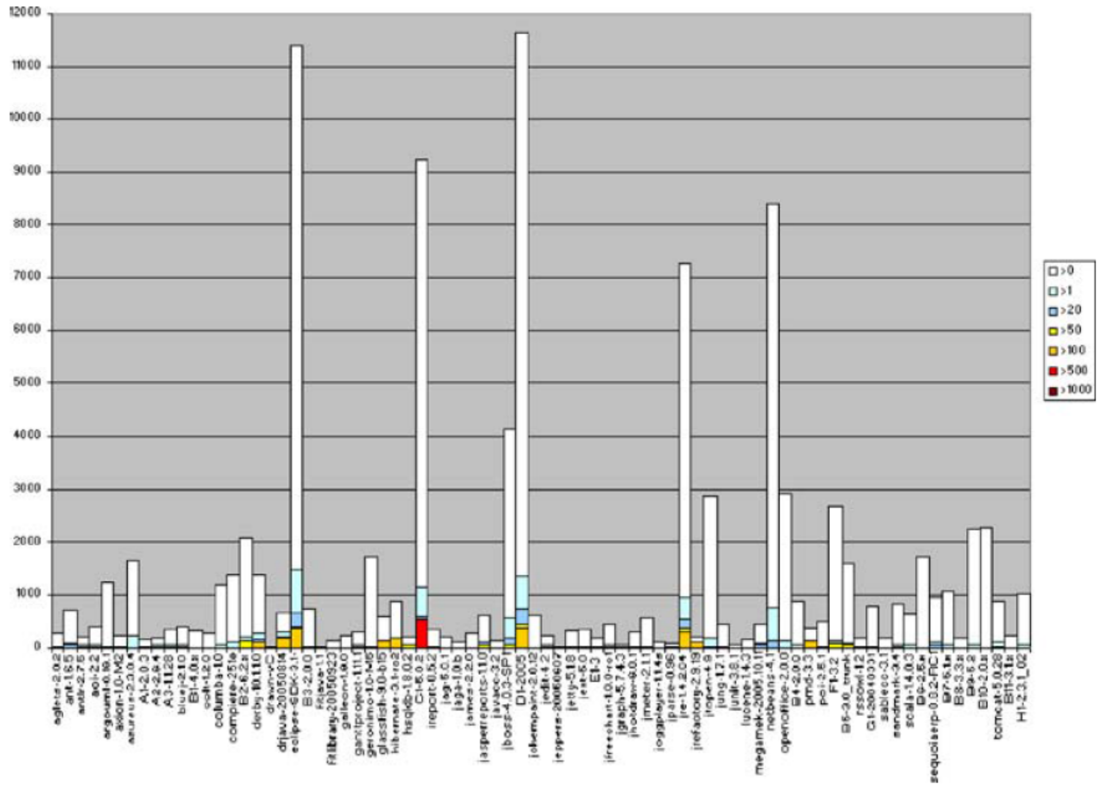


Figure 9 SCCs in uses-in-the-interface relation over corpus

Figure 9 depicts a chart of the SCCs in the uses-in-the-interface relation. Examination of the SCCs in this relation allow us to determine a rough upper-bound for cycles in a system due to intrinsic interdependency. If this is a reasonable upperbound then the plot of the SCCs in this relation compared to the charts of SCCs in the uses and uses-in-size seems to show that most cycles are "bad" or "unnecessary" cycles. For instance, consider the application D1-2005. In this application the largest SCC in the uses relation is >1,000 classes, but the largest SCC in the uses-in-the-interface relation is only >100 classes. Also for this application there are far fewer classes involved in cycles in the uses-in-the-interface relation (around 1,300) than in the uses relation (around 5,000).

Some interesting observations from Fig. 9 are as follows:

· Only one application has a SCC in the uses-in-the-interface relation >500 in size (C1-5.0.2).

· The largest SCC in the uses-in-the-interface is dramatically smaller than that of the uses and uses-in-size relations for most of the applications in the corpus. Consider Eclipse and Netbeans in illustration—the largest SCC in Netbeans is <51 (was >100 in the uses relation) and the largest in Eclipse is now <501 (was around 700). Even in Azureus where the largest SCC was around 800 for the uses relation it is <21 for the uses-in-the-interface relation.

We can infer from the dramatic decrease in SCC size going from either of uses or uses-in-size to uses-in-the-interface that types referred to *only* in the private parts of a class are the major contributor to large SCCs in the uses or uses-in-size.

6.4.3 SCC Time-series Data

We noted earlier that a feature of our corpus is that it contains multiple versions for 22 of the 78 applications in it. This allows us to examine how the SCCs in each of the dependency relations changed as the application evolved (i.e., had new features added, was enhanced, had defects fixed etc).

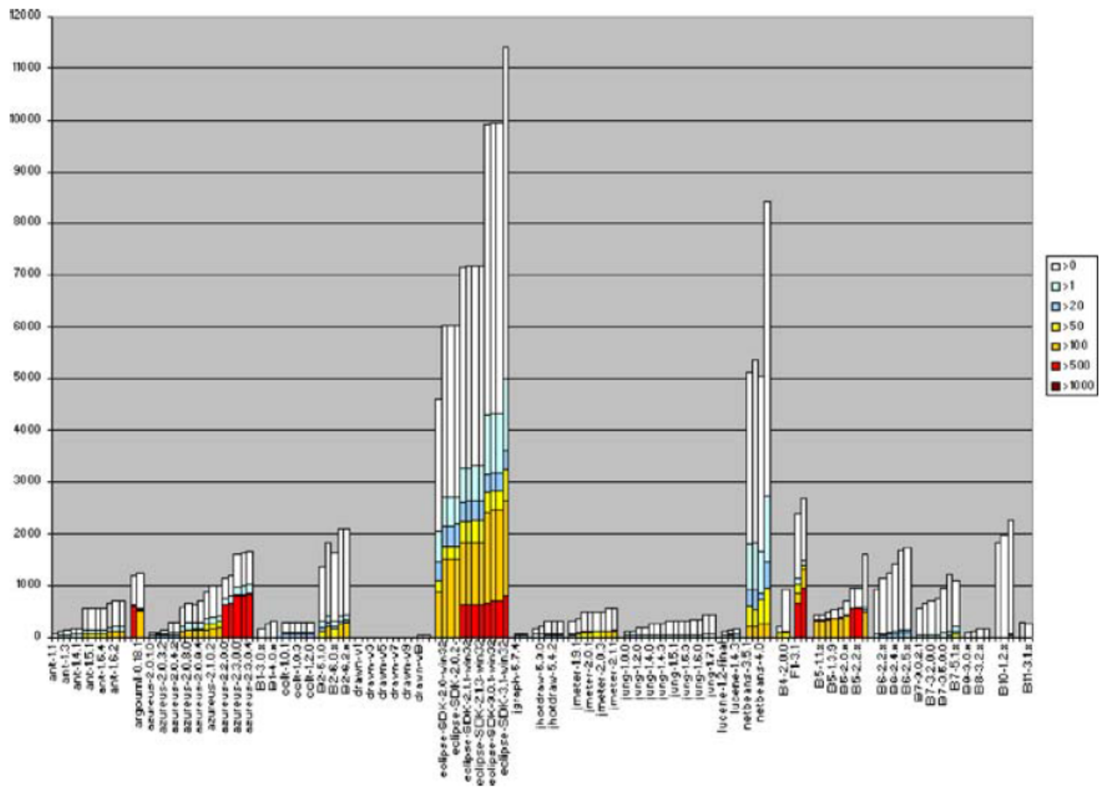


Fig. 10 SCCs in uses relation over time

Figures 10, 11 and 12 show the breakdown of each application into SCCs for each of its versions in using the stacked bar graph technique introduced for Figs. 7, 8 and 9. We first consider the plot of Fig. 10, which shows how cycles in the uses relation change over time. From this chart we can infer:

· The number of classes comprising an application tend to grow over time. This is hardly surprising since in order to add new features to a program we often create new classes as well as modify existing ones.

· The number of classes involved in each of the SCC categories (i.e., >1,000, >500, >100, >50, >20, >1) tend to increase for an application over time. That said, consider B2 and B5: in the middle of B2's version history the largest SCC dips from >100 to >50 and at the end of B5's version history dips from >500 to >100. Our discussion with company B revealed that at these points in version history both these products underwent major re-factorings (or rewrites) to improve their internal structure. A consequence of this was reducing the largest SCC. This is particularly interesting because developers at the company had no knowledge of cycles during these re-factorings, yet their improvement of the applications' designs based on their notion of good design also reduced the size of the SCCs.

· The size of the largest SCC in ArgoUML also decreased between versions. We think that this was due to some re-factoring activity. The change history for ArgoUML[21] shows that for version 0.17.1 and 0.17.3 (versions between the two versions we have of ArgoUML) that the changes were "Removed deprecated methods" and "Changed persistence mechanism," respectively. Removing deprecated methods certainly has the potential to remove dependencies as does changing the persistence mechanism.

· There is also a dip in the total number of classes participating in SCCs in Netbeans from version 3.6 to 4.0. Again we believe that this may be due to a rewrite of the 'project'model for Netbeans 4.0 as stated on its What's New webpage[22]: "Projects have been completely redesigned in NetBeans IDE 4.0."

Azureus is also worthy of mention because unlike Eclipse, where enhancements tend to increase the number of classes participating in cycles of each of the SCC groups (i.e., >500, >100, >50 etc), in Azureus the trend is that classes attach themselves to largest SCC causing it to grow. This large SCC has a greater potential to affect package structure than many small SCCs (see Section 2.2).

Figure 11 shows how cycles in the uses-in-size relation change over time The trends for the cycles in this plot mirror those in the uses plot of Fig. 10.

Consider the chart of Fig. 12, which shows how cycles in the uses-in-the-interface relation change over time. One interesting thing to note from this plot is that there is also a dip in the total number of classes participating in SCCs in Netbeans from version 3.6 to 4.0 (as per the uses relation). There are however no discernible dips in this relation's SCCs corresponding to those dips in the uses relation's SCC for applications B2, B5 and ArgoUML.

[21] http://argouml.tigris.org/project_schedule.html
[22] http://www.netbeans.org/community/releases/40/whats-new-40.html

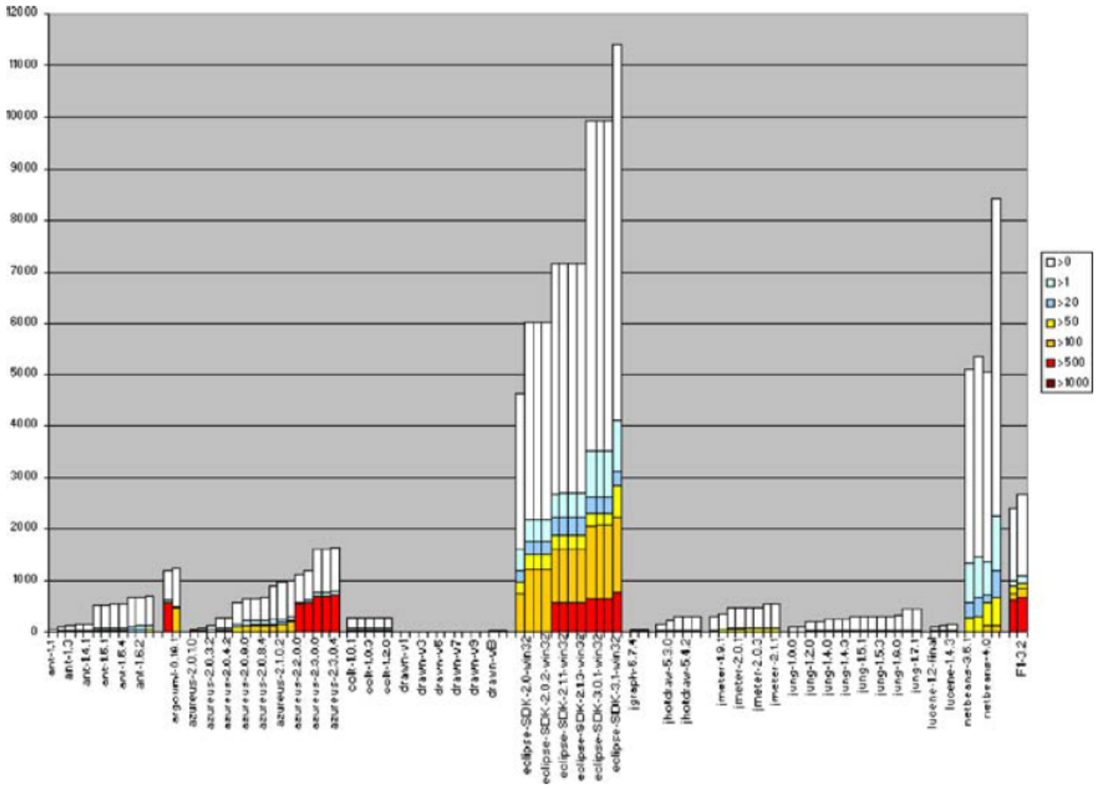


Figure 11 SCCs in uses-in-size relation over time

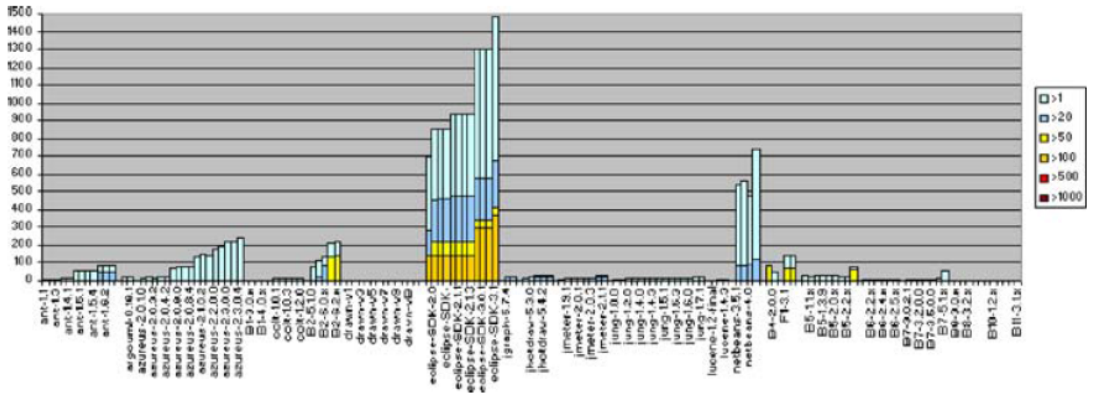


Figure 12 SCCs in uses-in-the-interface relation over time

6.4.4 mEFS Data

In this section we attempt to gauge the strength of connection among the source files in a SCC by computing a mEFS using the adaptation of Eades' Heuristic described in Section 3.3. To get a sense as to how much our adaptation affected the result, we compare the size of the smallest mEFS returned by our algorithm to that returned by Eades' Heuristic. In Figs. 13 and 14 it appears that the "Eades' mEFS" series does not feature in any of the bars. This is because the best mEFS returned from our algorithm was always almost equal in size to that returned by Eades' Heuristic.

When the two mEFS sizes differed the difference was at most one edge. Recall that we ran our modified algorithm 100 times, each time forcing a permutation inherent in the vertex sequence, and chose the smallest mEFS returned for our results.

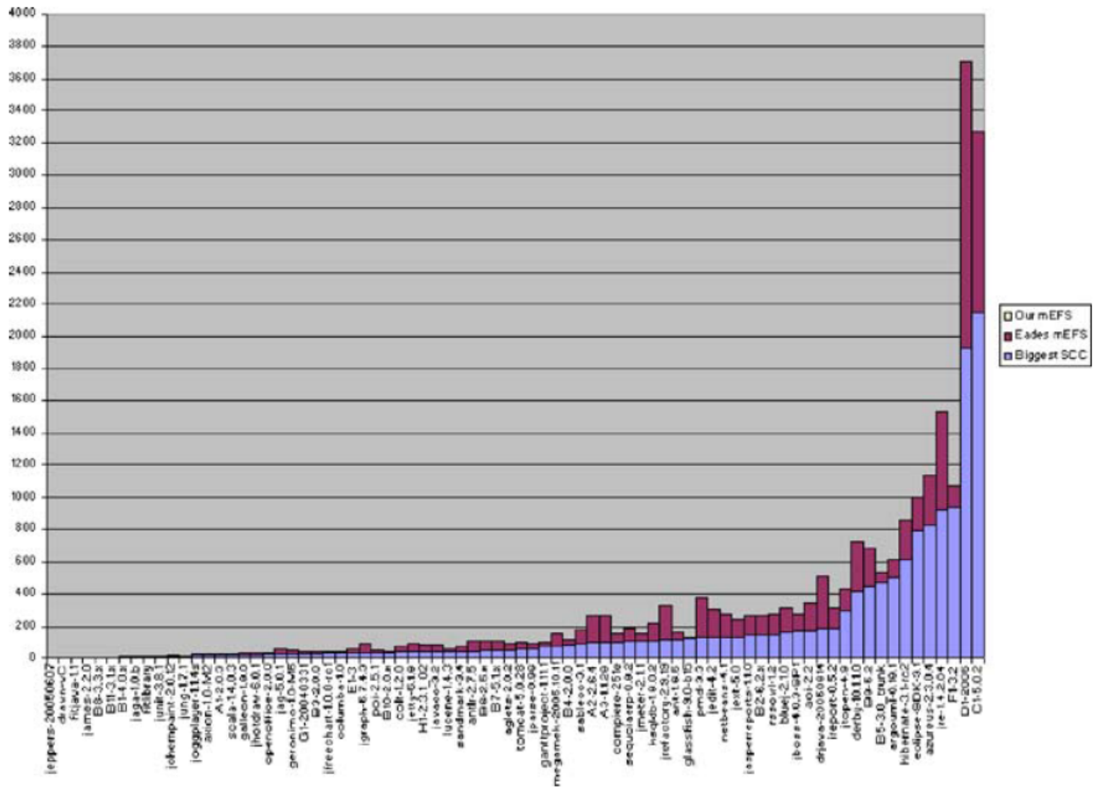


Fig. 13 mEFS for largest SCC in uses relation over corpus

Figure 13 shows the largest SCC in each application from the corpus and the mEFS size for this SCC. It shows that, in reality, not all SCCs are equally strongly connected.

The *x*-axis on this plot is the application and the *y*-axis shows the number of classes in the SCC as well as the number of edges in the mEFS. The *y*-axis simultaneously represents both number of classes and number of edges but this is a consequence of stacking the mEFS size bar on top of the SCC size bar for each application. By stacking the bars in this way (and sorting entries on the *x*-axis by their biggest SCC size) we can easily visually compare the sizes of the mEFS sets for similar sizes of SCC.

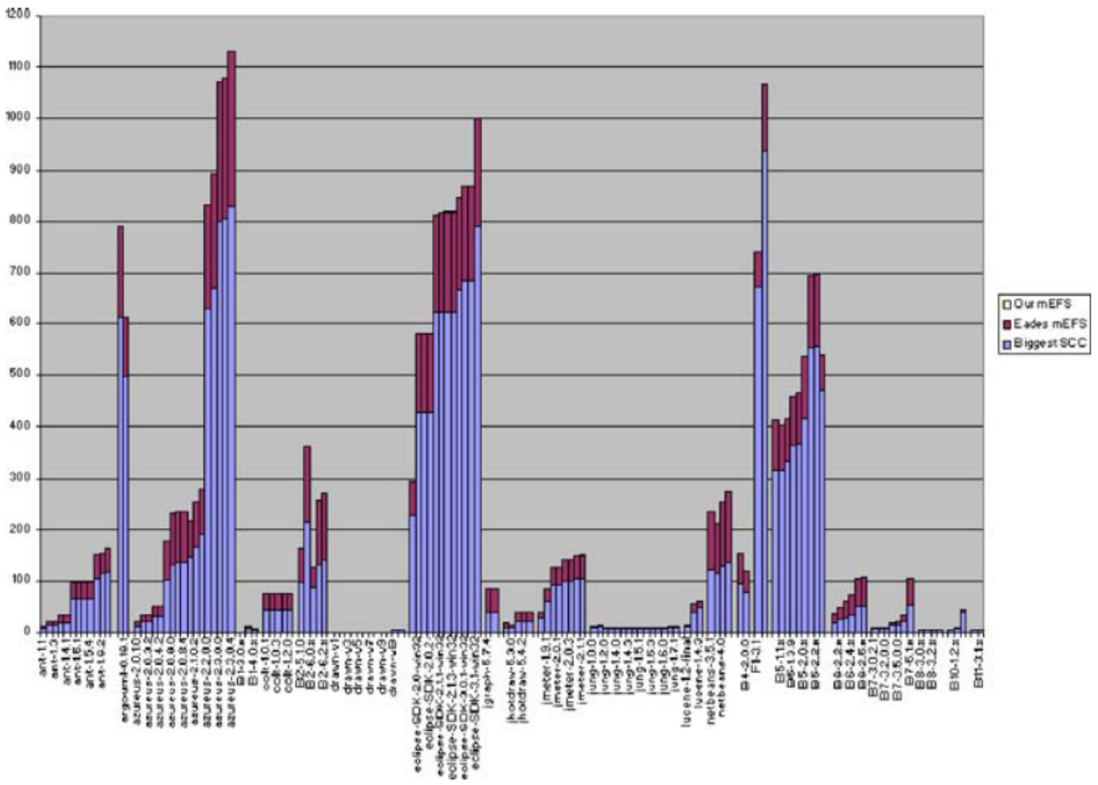


Figure 14 mEFS for largest SCC in uses relation over time

Interesting things we can infer from the plot of Fig. 13 are:

·PMD, JEdit, NetBeans and Jext all have a SCC that is of approximately the same size (about 130 classes) yet the sizes of mEFSs for each of the SCCs varies greatly. Of these applications PMD has the biggest mEFS so we can surmise that refactoring PMD to break up this SCC is probably going to be more work than re-factoring say Jext (the application with the smallest mEFS).

·Glassfish is particularly interesting because it has the smallest mEFS (4) for the size of its SCC (128). This means that its SCC is relatively weakly-connected compared to the other applications.

Figure 14 shows the growth in the largest SCCs in the uses relation for each application over time. It also shows the corresponding mEFS for each SCC. It is interesting because the mEFS size tends to remain constant if the SCC size remains constant between an application's consecutive versions. We thought that it might be possible for the SCC to become more "strongly connected" between versions of an application if the classes in it had dependencies added. It seems however that usually the SCC retains the same strength of among nodes if it does not grow in size.

6.5 Discussion

The results in this paper have several interesting implications. Firstly, we saw that for any given application the SCCs uses-in-the-interface relation were typically *much* smaller than the SCCs in both the uses and uses-in-size relations. We noted that this meant that types appearing *only* in the private parts of a class were the major contributor to large SCCs in the latter dependency relations. Further investigation is required to better understand the mechanisms by which a type can only appear in the private part of a class, and not be used, even transitively, in the class' uses-in the-interface relation.

With respect to types appearing only in the private part of a class, at some level we should be pleased to observe this phenomenon because a well-designed class hides information (its implementation details) from its clients [Boo91, p.45][Lak96, p.155]. On the other hand, while these classes may seem well-designed from the perspective of information hiding when considered in isolation, their interactions with all the other classes in a system, expressed through the transitive closure of their dependencies, can inhibit a system's *overall* structure. It follows that OO metrics suites should be extended to consider dependency relationships transitively. Existing metrics such as those in the CK Metrics [CK91] are seldom computed such that they consider dependency transitively.

Secondly, it is argued in the instructional literature that for large-scale software systems *overall* structure is the most important aspect of organization [Lak96]. Despite this we have seen SCCs in the uses relation involving greater than 1,000 classes in two large-scale commercial systems (C1 and D1). We see similar SCCs in the uses relation involving greater than 500 classes in 5 other medium to large-scale, open and closed source systems. In our discussions with company B, subsequent to collecting data from them, we ascertained that two systems, B5 and B10 had to be thrown away because their source code had become too unwieldy. These systems had a higher proportion of classes involved in (long) cycles than the other systems from company B. This is empirical evidence, albeit weak, to suggest cycles are in fact detrimental to maintainability.

On the other hand, many systems with long cycles are considered state-of-the-art in their domains (e.g., Eclipse, JRE, Hibernate and ArgoUML). This has potentially interesting consequences. Does it mean that these applications are more difficult to test, maintain and understand than they would be without cycles, or that cycles do not have a significant effect on these quality attributes after all? Further investigation is definitely needed in this area.

Thirdly, the data in this paper provides empirical evidence to support claims that have been made by other researchers. Foote et al. for instance claim that the most frequently deployed architectural pattern is the *Big Ball of Mud*: a haphazardly structured system whose source code lacks organisation. If we take "haphazardly structured" and "lacks organisation" to mean that the structure of the software system does not compare favourably with the instructional literature then, at the level of *overall structure*, we believe that our data is the first empirical evidence to support this.

Foote et al. also claim that without intervention (i.e., continuous re-factoring) a design can, and will, degrade over time. This is consistent with Lehman's second "law" of software evolution: as a system evolves its complexity increases unless work is done to maintain or reduce it" [LRW+97]. Again if we take "degrade" to mean that the structure diverges further from the instructional literature then our data supports this claim. For most of the applications for which we had multiple versions, the number of classes involved in SCCs and the size of SCCs tended to grow. We noted that dips in SCC size for several applications (B2, B5, ArgoUML and Netbeans) corresponded to major re-factoring efforts. We have no knowledge of majoring re-factoring efforts that did not result in a reduction in SCCs, although it is a distinct possibility that these may exist. A detailed study of various application's histories with respect to re-factoring and cycles could be an area of future work.

6.5.1 Netbeans vs. Eclipse

One of the most interesting comparisons of applications in our corpus is between Netbeans and Eclipse. We can reasonably compare these applications with respect to many criteria because both come from the same domain (IDEs), both provide similar functionality and both purportedly have plug-in style architectures. The Netbeans team[23] claim that the IDE is modular in that the core runtime is a generic desktop application that can be used for applications other than IDEs and all of the features of the IDE (e.g. the Java code editor) comprise plug-in modules. The Eclipse team[24] similarly claim that "The Eclipse Platform is an IDE for anything, and for nothing in particular." The Java capabilities of Eclipse are all provided through plugin modules.

With respect to cycles we saw the largest SCC in the uses relation for Netbeans is 135 and Eclipse is 791. If these cycles are confined within individual modules, which they should be because we do not want our modules to be cyclically dependent, then we can infer that Eclipse must have a module comprising at least 791 classes, whereas it is possible that the biggest module in Netbeans comprises only 135 classes. Indeed Netbeans tends to have smaller SCCs in the uses relation than Eclipse, which may suggest according our argument about how cycles affect package structure in Section 2.2 that Netbeans has finer-grained modules than Eclipse.

6.5.2 Threats to Validity

In Section 3.2 we noted that some type information is lost in the conversion from source code to byte code. We noted that this was not problematic for the context of this study because it meant that the dependencies (and thus cycles) we were able to detect from .class files were a (non-strict) subset of those appearing among .java files. We went on to say the results presented in this paper were thus a lower-bound on the actual cycles. In fact, they are a lower bound on cycles for another reason too. The dependencies analysed in this paper take into account only *compilation dependencies*. There could be further "logical" dependencies we were unable to infer because these were expressed through reflection or dynamic class loading. So, in terms of our results, some of the applications we noted as having few cycles (small SCCs) may still be poorly designed with respect to cycles if such compilation dependencies are being avoided with techniques that are not type-safe.

[23] http://www.netbeans.org/products/platform/howitworks.html
[24] http://www.eclipse.org/whitepapers/eclipse-overview.pdf

In this paper we implied that our results show that cycles are common among classes of OO systems in use the world today. This assumes that our Java corpus is representative of real world OO software. Some OO languages prohibit cyclic dependencies (e.g., Component Pascal [SGM02, p.154]) so our results cannot be generalised to software written in these languages. Also, we tried to ensure some notion of representativeness by selecting applications to vary along several dimensions (size, domain, origin and open or closed-source) but have no statistical argument to support representativeness. Size is another issue that affects the representativeness of our corpus—it comprises only 78 applications when there are probably hundreds of thousands of Java programs in the world today (SourceForge alone listed over 16,000 projects as being written in Java as at 5 October 2005).

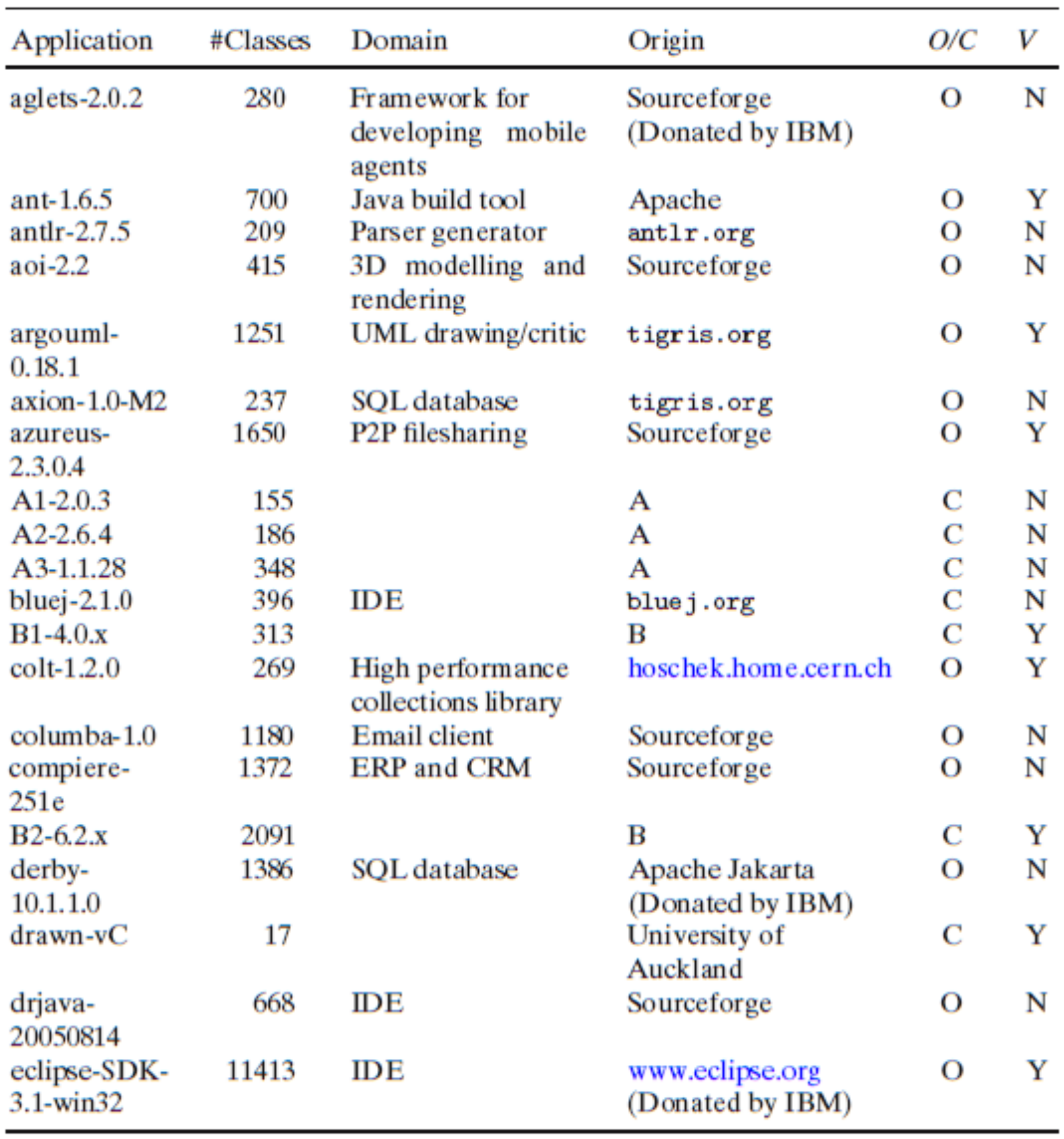
6.6 Conclusions

We have presented the first empirical study on the existence of cyclic dependencies in code. Our motivation for carrying out this study was the apparent contradiction between the software design literature that advises against having cyclic dependencies, and the folklore that suggests that dependency cycles are common in software. Our study found large and complex cyclic structures in almost all of the 78 applications we studied. This provides strong evidence supporting the folklore, at least in the context of Java.

Now the question has to be why, with all the advice to the contrary, are cycles so prevalent? We note that there is in fact no empirical evidence showing the relationship between cyclic dependencies and any software quality attribute and so one reason could be that the advice is just wrong. If this is true, then given the compelling arguments for this advice, we would have to wonder about other design advice that has equally compelling arguments. If the advice is correct, then our study suggests there is lots of "bad" software around.

There is still a great deal of research to be done. Our study raises a number of questions. Some questions suggested by our study include: do our results hold for all Java software; do our results hold for other object-oriented programming languages; is there a relationship between cyclic dependencies and the various software quality attributes mentioned; if cycles are indeed “bad,” then how is it that so much software has them; how do we remove or reduce cyclic dependencies; and, how do we avoid introducing them in the future? Also, these questions should be asked of all other design advice that has been given.

Appendix: Corpus Details



Application	#Classes	Domain	Origin	*O/C*	*V*
aglets-2.0.2	280	Framework for developing mobile agents	Sourceforge (Donated by IBM)	O	N
ant-1.6.5	700	Java build tool	Apache	O	Y
antlr-2.7.5	209	Parser generator	antlr.org	O	N
aoi-2.2	415	3D modelling and rendering	Sourceforge	O	N
argouml-0.18.1	1251	UML drawing/critic	tigris.org	O	Y
axion-1.0-M2	237	SQL database	tigris.org	O	N
azureus-2.3.0.4	1650	P2P filesharing	Sourceforge	O	Y
A1-2.0.3	155		A	C	N
A2-2.6.4	186		A	C	N
A3-1.1.28	348		A	C	N
bluej-2.1.0	396	IDE	bluej.org	C	N
B1-4.0.x	313		B	C	Y
colt-1.2.0	269	High performance collections library	hoschek.home.cern.ch	O	Y
columba-1.0	1180	Email client	Sourceforge	O	N
compiere-251e	1372	ERP and CRM	Sourceforge	O	N
B2-6.2.x	2091		B	C	Y
derby-10.1.1.0	1386	SQL database	Apache Jakarta (Donated by IBM)	O	N
drawn-vC	17		University of Auckland	C	Y
drjava-20050814	668	IDE	Sourceforge	O	N
eclipse-SDK-3.1-win32	11413	IDE	www.eclipse.org (Donated by IBM)	O	Y

(continued)

Application	#Classes	Domain	Origin	*O/C*	*V*
B3-2.0.0	722		B	C	N
fitjava-1.1	37	Automated testing	fit.c2.com	O	N
fitlibraryforfitnesse-20050923	124	Automated testing	Sourceforge	O	N
galleon-1.8.0	243	TiVo media server	Sourceforge	O	N
ganttproject-1.11.1	310	Gantt chart drawing	Sourceforge	O	N
geronimo-1.0-M5	1719	J2EE server	Apache	O	N
glassfish-9.0-b15	582	J2EE server	dev.java.net	O	N
hibernate-3.1-rc2	902	Persistence object mapper	Sourceforge	O	N
hsqldb-1.8.0.2	217	SQL database	Sourceforge	O	N
C1-5.0.2	9240		C	C	N
ireport-0.5.2	347	Visual report design for JasperReports	Sourceforge	O	N
jag-5.0.1	208	J2EE application generator	Sourceforge	O	N
jaga-1.0.b	100	API for genetic algorithms	jaga.org	O	N
james-2.2.0	259	Enterprise mail server	Apache	O	N
jasperreports-1.1.0	633	Reporting tool	Sourceforge	O	N
javacc-3.2	125	Parser generator	dev.java.net	O	N
jboss-4.0.3-SP1	4143	J2EE server	Sourceforge	O	N
D1-2005	11644		D	C	N
jchempaint-2.0.12	612	Editor for 2D molecular structures	Sourceforge	O	N
jedit-4.2	234	Text editor	Sourceforge	O	N
jeppers-20050607	20	Spreadsheet editor	Sourceforge	O	N
jetty-5.1.8	327	HTTP Server/servlet container	Sourceforge	O	N
jext-5.0	353	IDE	Sourceforge	O	N
E1-3	192		E	C	N
jfreechart-1.0.0-rc1	469	Chart drawing	Sourceforge	O	N
jgraph-5.7.4.3	50	Graph drawing	Sourceforge	O	Y

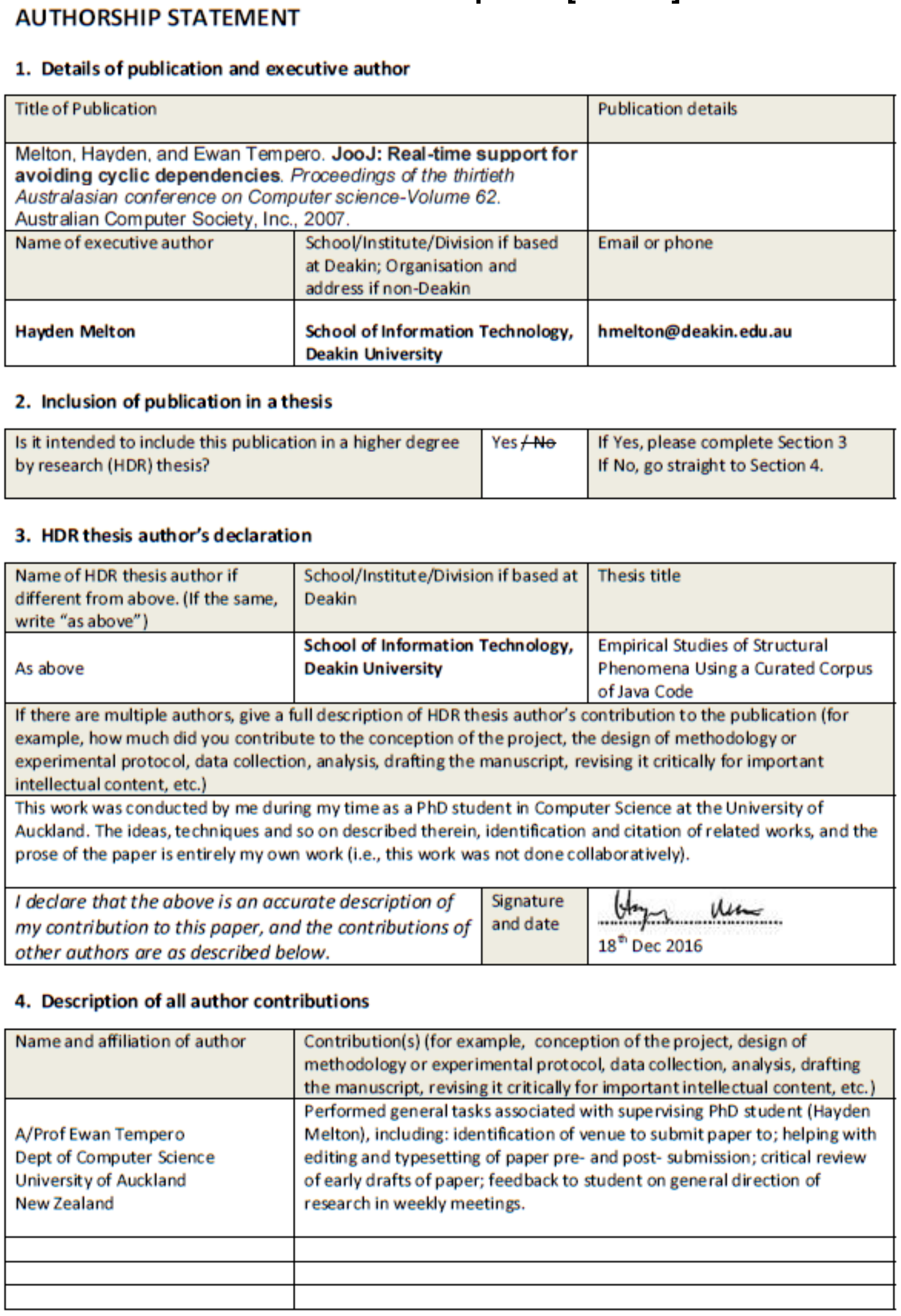
(continued)

Application	#Classes	Domain	Origin	O/C	V
jhotdraw-6.0.1	300	Graph drawing	Sourceforge	O	Y
jmeter-2.1.1	560	Java performance testing	Apache	O	Y
joggplayer-1.1.4s	114	MP3 player	joggplayer.webarts.bc.ca	O	N
jparse-0.96	69	Java compiler front-end	www.ittc.ku.edu/JParse	O	N
jre-1.4.2.04	7257	JRE	Sun	O	N
jrefactory-2.9.19	211	Re-factoring tool for Java	Sourceforge	O	N
jtopen-4.9	2857	Java toolbox for iSeries and AS/400 servers	Sourceforge (Donated by IBM)	O	N
jung-1.7.1	454	Graph drawing	Sourceforge	O	Y
junit-3.8.1	48	Unit testing framework	Sourceforge	O	N
lucene-1.4.3	170	Text indexing	Apache	O	Y
megamek-2005.10.11	455	Game	Sourceforge	O	N
netbeans-4.1	8406	IDE	netbeans.org (Donated by Sun)	O	Y
openoffice-2.0.0	2925	Office suite	openoffice.org (Donated by Sun)	O	N
B4-2.0.0	902		B	C	Y
pmd-3.3	375	Java code analyser	Sourceforge	O	N
poi-2.5.1	480	API to access Microsoft format files	Apache	O	N
F1-3.2	2684		F	C	Y
B5-3.0-trunk	1589		B	C	Y
rssowl-1.2	189	RSS Reader	Sourceforge	O	N
G1-20040331	786		G	C	N
sablecc-3.1	199	Compiler/ Interpreter generating framework	Sourceforge	O	N
sandmark-3.4	837	Software watermarking/ security	www.cs.arizona.edu/ sandmark	O	N
scala-1.4.0.3	654	Multi-paradigm programming language	scala.epfl.ch	O	N
B6-2.5.x	1738		B	C	Y

(continued)

Application	#Classes	Domain	Origin	*O/C*	*V*
sequoiaerp-0.8.2-RC1-all-platforms	936	ERP and CRM	Sourceforge	O	N
B7-5.1.x	1093		B	C	Y
B8-3.3.x	178		B	C	Y
B9-5.2	2241		B	C	N
B10-2.0.x	2273		B	C	Y
tomcat-5.0.28	892	Servlet container	Apache	O	N
B11-3.1.x	238		B	C	Y
H1-2.3.1-02	1029		H	C	N

Coauthor Declaration for Chapter 7 [MT07c]



AUTHORSHIP STATEMENT

1. Details of publication and executive author

Title of Publication		Publication details
Melton, Hayden, and Ewan Tempero. **JooJ: Real-time support for avoiding cyclic dependencies**. *Proceedings of the thirtieth Australasian conference on Computer science-Volume 62.* Australian Computer Society, Inc., 2007.		
Name of executive author	School/Institute/Division if based at Deakin; Organisation and address if non-Deakin	Email or phone
Hayden Melton	**School of Information Technology, Deakin University**	**hmelton@deakin.edu.au**

2. Inclusion of publication in a thesis

Is it intended to include this publication in a higher degree by research (HDR) thesis?	Yes / ~~No~~	If Yes, please complete Section 3 If No, go straight to Section 4.

3. HDR thesis author's declaration

Name of HDR thesis author if different from above. (If the same, write "as above")	School/Institute/Division if based at Deakin	Thesis title
As above	**School of Information Technology, Deakin University**	Empirical Studies of Structural Phenomena Using a Curated Corpus of Java Code
If there are multiple authors, give a full description of HDR thesis author's contribution to the publication (for example, how much did you contribute to the conception of the project, the design of methodology or experimental protocol, data collection, analysis, drafting the manuscript, revising it critically for important intellectual content, etc.)		
This work was conducted by me during my time as a PhD student in Computer Science at the University of Auckland. The ideas, techniques and so on described therein, identification and citation of related works, and the prose of the paper is entirely my own work (i.e., this work was not done collaboratively).		
I declare that the above is an accurate description of my contribution to this paper, and the contributions of other authors are as described below.	Signature and date	[illegible] 18th Dec 2016

4. Description of all author contributions

Name and affiliation of author	Contribution(s) (for example, conception of the project, design of methodology or experimental protocol, data collection, analysis, drafting the manuscript, revising it critically for important intellectual content, etc.)
A/Prof Ewan Tempero Dept of Computer Science University of Auckland New Zealand	Performed general tasks associated with supervising PhD student (Hayden Melton), including: identification of venue to submit paper to; helping with editing and typesetting of paper pre- and post- submission; critical review of early drafts of paper; feedback to student on general direction of research in weekly meetings.

5. Author Declarations

I agree to be named as one of the authors of this work, and confirm:

i. that I have met the authorship criteria set out in the Deakin University Research Conduct Policy,

ii. that there are no other authors according to these criteria,

iii. that the description in Section 4 of my contribution(s) to this publication is accurate,

iv. that the data on which these findings are based are stored as set out in Section 7 below.

If this work is to form part of an HDR thesis as described in Sections 2 and 3, I further

v. consent to the incorporation of the publication into the candidate's HDR thesis submitted to Deakin University and, if the higher degree is awarded, the subsequent publication of the thesis by the university (subject to relevant Copyright provisions).

Name of author	Signature*	Date
Ewan Tempero	[illegible]	16/12/2016

6. Other contributor declarations

I agree to be named as a non-author contributor to this work.

Name and affiliation of contributor	Contribution	Signature* and date

* If an author or contributor is unavailable or otherwise unable to sign the statement of authorship, the Head of Academic Unit may sign on their behalf, noting the reason for their unavailability, provided there is no evidence to suggest that the person would object to being named as author

7. Data storage

The original data for this project are stored in the following locations. (The locations must be within an appropriate institutional setting. If the executive author is a Deakin staff member and data are stored outside Deakin University, permission for this must be given by the Head of Academic Unit within which the executive author is based.)

Data format	Storage Location	Date lodged	Name of custodian if other than the executive author
Qualitas Corpus of Java code, both binaries and source code with metadata (see *http://qualitascorpus.com/*)	University of Auckland, New Zealand	Starting 2006, onward	Ewan Tempero

This form must be retained by the executive author, within the school or institute in which they are based.

If the publication is to be included as part of an HDR thesis, a copy of this form must be included in the thesis with the publication.

Chapter 7 JooJ: Real-time Support for Avoiding Cyclic Dependencies

The design guideline *avoid dependency cycles among modules* was first alluded to by Parnas in 1978. Many tools have since been built to detect cyclic dependencies among a program's organisational units, yet we still see real applications riddled with large dependency cycles. Our solution to this problem is to proactively check for dependency cycles as a developer writes code. In this way a cycle can be identified and eliminated the moment any fragment of code is written that induces one. This approach is analogous to a well-known manufacturing quality assurance technique known as *poka-yoke*. We demonstrate the feasability our 'realtime checking' approach via an Eclipse plugin we have built called JooJ.

7.1 1 Introduction

Over the years there have been many guidelines proposed for writing effective code. Roughly speaking these guidelines fall into three categories — those pertaining to (1) style, (2) correctness and (3) design. Style guidelines aim to improve the readability of code through consistent naming and formatting (e.g., *Code Conventions for the Java Programming Language* [Sun99]). Correctness guidelines aim to help programmers avoid common or subtle errors (e.g., "Class overrides equals() without overriding hashCode()"[Blo01]). Design guidelines aim to help programmers make decisions about the internal structure of a system (e.g., Riel's *Object-Oriented Design Heuristics* [Rie96] and Design Patterns [GHJV95]).

There are many tools currently available for checking conformance of Java code to style, correctness and design guidelines. We are interested in those that provide continuous (or proactive) checking as opposed to those that are run intermittently, at a developer's discretion. The Eclipse Integrated Development Environment (IDE) is a good example of a tool that proactively checks Java code against style and correctness guidelines. As the developer enters code into Eclipse it is analysed in 'real-time' for problems (e.g., syntax error, unused local variable, unparameterised use of a generic type etc). In this way the developer gets immediate feedback about some aspects of the quality of his code. The importance of this immediacy is evident in a well-known aphorism: that it's cheaper to fix problems earlier in the development process than later[Pre01, p.197-198].

While 'real-time' code analysis has successfully been implemented by Eclipse (and other IDEs) for supporting correctness and style guidelines it seems that there are few, if any, tools available that take this approach to supporting *design guidelines* at the level of source code. We believe one reason for this is that often it is computationally more expensive to analyse code for design guidelines than to do so for style and correctness guidelines. This is because many design guidelines, especially the one in which we are interested in, provide advice about structuring of the whole system and cannot be determined solely through the analysis of a single source file.

Another reason why design guidelines may not be supported through real-time code analysis is that it is often difficult to determine a satisfactory measure for a design guideline from source code. In the case of module cohesion, for instance, there have been numerous metrics presented that can be automatically computed from source code (see [BDW98] for example) yet none is widely accepted or even used by practitioners. Fortunately the design guideline in which we are interested does not suffer from this measurement problem.

In this paper we present a tool we have developed to determine the feasibility of proactively supporting the design principle avoid dependency cycles among modules through real-time source code analysis. Our tool, JooJ (pronounced "Joo-jay"), has been developed as a plugin for Eclipse. It transparently extends the style and correctness checking already provided by Eclipse.

The remainder of the paper is organised as follows. In Section 2 we review the design principle JooJ supports and discuss the motivation for JooJ. In Section 3 we give an overview of JooJ's expected user interface and features. In Section 4 we discuss some of the details of JooJ's implementation. In Section 5 we evaluate the performance of JooJ in terms of time and space. In Section 6 we review other cycle-detecting and real-time analysis tools. Finally, in Section 7, we draw conclusions from this work.

7.2 Background and Motivation

Software design guidelines guide the decisions developers make about the internal structure of a system. They help us to structure a system in a way that makes it easy to understand, test, modify, reuse and so on. The design guideline relevant to this paper is avoid dependency cycles among modules. Dependencies among the source files of an application are a natural consequence of modularisation. In dividing a program up into modules we break it up into more manageable parts, but these parts must collaborate in order to provide the functionality of the system as a whole. It is these collaborations that cause dependencies.

7.2.1 Impact of Cycles

Parnas was the first to discuss the effect dependency cycles among a program's modules might have on software quality attributes[Par78]. He argued that when two modules were cyclically dependent neither could not be tested, build or reused independently of the other. When there are long dependency cycles encompassing many modules Parnas argued that we might end up with a system where no single part of the works until all the rest of it works.

The most comprehensive work on cycles in the context of the Object-Oriented (OO) paradigm is by Lakos. He states that cycles among the source files of C++ programs inhibit understanding, testing and reuse[Lak96, p.185], and that cycles among packages inhibit development, marketing, usability, production and reliability[Lak96, p.494-495].

Other design guidelines also support the "avoid cycles" guideline. For instance, Riel states "Derived classes must have knowledge of their base class by definition, but base classes should not know anything about their derived classes"[Rie96, p.81]. Disallowing the dependency of a base classes on its derived classes prevents a dependency cycle between the base and derived classes. Stevens et al. state the design guideline *minimise coupling between modules*. A design with dependency cycles has higher coupling than its acyclic analog (e.g., if modules A and B are in a cycle then B has higher coupling than if the only dependency is from A on B). Booch says "…all well structured object-oriented architectures have clearly defined layers"[Boo95]. Long dependency cycles make it difficult to divide a system's classes into clearly defined layers, where classes in a given layer can only depend on others in lower layers.

If a cyclic dependency exists, then the question arises as to how to remove it. This must involve removing a dependency, and so breaking a collaboration, but which one? Lakos provides some advice on deciding which dependency to break but this advice often relies on characteristics of the problem domain (e.g., this object is more "primitive" than that). Many OO design guidelines also provide advice. For example, if one class is "a part of" another, then the other must always depend on it, whereas any dependency by the part on the whole is not always necessary. Similarly, a subclass must always depend on its parent, but a parent should not depend on any of its children. In a model-view-controller design, the view must depend on the model, but *the model of an application should not depend on its view*[Rie96, p.36]. What this means is that it does not make sense to remove some dependencies, so we must provide some means to manage such dependencies, a point we return to in Section 3.

7.2.2 Definition of Cycle

We have adapted Lakos' work with cycles in C++ to Java [MT07b]. For simplicity of explanation, we assume all "top-level" classes are declared in separate .java source files. This means that for a class A, A.java and A.class refer to the same entity, and we will use these interchangeably. There are several subtle variations on the definition of "dependency", particularly with regard to differences between Java 5 and its predecessors. These variations are discussed in our adaptation of Lakos' work[MT07b]. Our tool can cope with each of these, and it is sufficient for our presentation to use the simplest: A class A DependsOn a class B if it needs B.class on the classpath in order to compile.

7.2.3 Prevalence of Cycles

Our main motivation for providing tool support to avoid dependency cycles comes from an empirical study we performed [MT07b]. The results of this study indicate that not only do cycles exist in many Java applications, but they are often large and complex. In our study we analysed a corpus of 78 real, open- and closed-source Java applications and found that:

• Two commercial applications each had a single long cycles involving over 2000 top-level Java classes.
• Eight out of the 78 applications had single long cycle involving over 500 classes.
• Two popular, widely-downloaded, open source projects (Azureus and Hibernate) had more than half their classes involved in one big cycle.
• Close to 40% of the applications in the corpus had a single cycle involving more than 100 classes.

These results astonished us. They support a claim made by Foote et al. that the most frequently deployed software architecture is the *Big Ball of Mud* (Foote & Yoder 2000). They also justify the large amount of research that has been done on stubbing to break dependency cycles for integrating testing [HSR05],[BLW03] [Bin99, p.980-985]. More importantly these results *strongly* motivate the need for a tool to help prevent cycles ever appearing in source code. If we could prevent cycles appearing in a system's source code, as advocated by Lakos [Lak96] and others[Bin99, p.984], then there would be no need for stubbing — an activity Binder identifies as potentially risky, expensive, difficult, and inadequate in the presence of large complex cycles [Bin99,p.983-984].

7.2.4 The Need for Real-time Feedback

There are already many tools for supporting *avoid dependency cycles* in Java (e.g. ByeCycle, Classycle, Dependency Finder, PASTA tool, JDepend, Lattix LDM, see Section 6). The majority of these tools take a batch-style type approach to supporting these principles. The prevalence of dependency cycles in real-world Java software indicates that either these tools are ineffective or software developers do not care much about avoiding dependency cycles. The sheer number of these (mostly free) tools makes it difficult to believe the other alternative: that developers 'just don't know' about their existence.

The problem with batch-style tools is that they do not allow problems to be fixed at the same time they are created. Two important reasons why cycle-causing code is hard to change retrospectively are:

Code is more resistive to change after it has been written. Imagine we are oblivious to a cyclic dependency induced by a statement we have just written. Without tool support this is likely because there is no way to tell if a statement induces a cyclic dependency simply by looking at a single source file — yet this is the way we edit and view source files, one at a time. We then write more statements that depend on the initial cycle-inducing one (and possibly inducing more cycles themselves). Eventually we get around to running our cycle detecting tool and discover the cycle. We are now faced with the task of figuring out how to change or move that statement (and its dependent statements) to break the cycle, and all the while not inducing new, different cycles.

The alternative is that we are informed as soon as we write a statement inducing a cycle. Instead of continuing we can remove the cycle at that point in time (for example by *escalating* [Lak96,p.215-228] that statement to a new or existing higher-level class). The effort involved in making changes to remove the cycle is now limited to dealing with just one statement.

Changing other people's code is hard. Imagine that another developer wrote the cycle inducing statements, but neglected to run or take notice of the output from our batch-style cycle tool. Now we have to change his code. We may introduce a bug in doing so if we fail to understand all the pre- and post-conditions of his code. We have to spend comparatively more time working out what someone else's code does. If we cannot understand the code or feel the risk regression from improving its structure is too high we may leave the code as it is. Over time the cycle may grow and grow until it encompasses most of the classes in the system, then the system will have to be thrown away and rewritten from scratch. Indeed cycle growth and throwing systems away are phenomena we have reported [MT07b].

Consider now the possibility that developers 'just don't care' about avoiding dependency cycles, or that it is a very low priority. As Foote et al. state "[software] architecture frequently takes a back seat to more mundane concerns such as cost, time-to-market, and programmer skill" [FY97]. We argue real-time, integrated tool support for avoiding dependency cycles can make developers care, and help ensure design principles do not take a back seat to more 'mundane' concerns.

Before we (the authors) started using Eclipse (3.1.1) we were unaware of variable declarations in a class that were unused, or variables whose values were assigned by never read from, or unused private methods. Now when we write code, we are immediately informed by Eclipse of these problems (and others) through yellow squiggly underlines of individual statements. Slowly but surely we started taking heed of this feedback as we coded. Continuous 'micro-refactorings' to eliminate these problems are now part of our personal coding styles. We suspect that there is a psychological force that drives us to fix statement with yellow squiggly lines under them. We must, of course, fix statements with red squiggles beneath them because these are compilation errors. (We note that in the mid-90's Microsoft Word was changed to include continuous checking of spelling and grammar and that again, with this feature, we are compelled to get rid of the squiggles as soon as they appear).

7.2.5 Wider Perspectives

The notion of preventing problems before they occur, or early in the production process, has been around for a long time in the manufacturing industry. In the 1960s an engineer at Toyota called Shigeo Shingo used the term *poka-yoke*, which means 'mistake-proofing', to describe this approach to quality assurance. A poka-yoke device aims to prevent potential quality problems before they occur or rapidly detects them as they are introduced [Pre01, p.214-215]. Pressman [Pre01, p.215] states that an effective poka-yoke device exhibits the following characteristics: (1) it is simple and cheap, (2) it is part of the process and (3) it is located near the process task where the mistakes occur. Indeed it can be argued that Eclipse's style and correctness guideline checking is an effective poka-yoke device because it brings checking closer to the activity of typing out code than batch-style tools. It also rapidly detects problems as they are created. The effectiveness of our tool, JooJ, can be argued in a similar fashion.

7.2.6 Applicability

It has been claimed that avoiding dependency cycles among modules is most applicable to large-scale software systems [Mar96b][Lak96]. Martin define large in the context of C++ as 50,000 LOC or more (Martin 1996) and Lakos defines large as in the same context as having “hundreds of header files” [Lak96, p.11]. The question we try to address here is *to what proportion of the world’s Java software is our tool applicable?*

A distribution of size in terms of number of classes in the Java corpus of a previous work [MT07b] is shown in Figure 1. The x-axis represents the number of .java files in a system and the y-axis represents the proportion of applications in the corpus that comprise *at least* that many .java files. So from this chart we can see that about 30% of the applications in the corpus comprise at least 1000 .java files. About 15% comprise at least 2000 .java files. If the corpus used to generate this plot is representative sample of real-world Java software, and we define large as 1000 .java files, then the support provided by JooJ is applicable to around 30% of the world’s Java software.

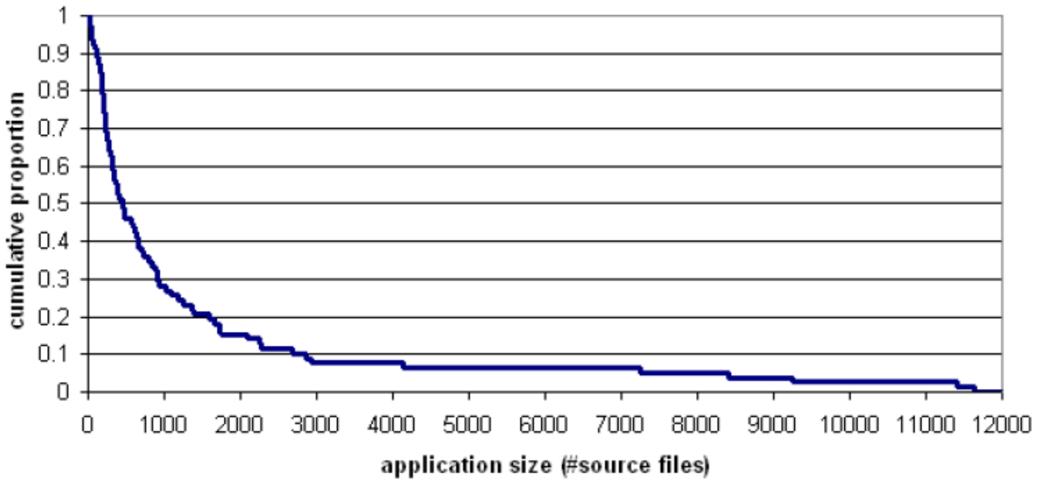


Figure 1: Distribution of application size across 78 Java applications

If we do not consider the corpus to be a representative sample of real-world Java software then consider what Fayad et al. have to say: “While a 100,000 source line program was a significant undertaking 20 years ago, the typical shrink-wrapped software product today embodies at least that many lines of code. While it is extremely difficult to identify a cost figure, it appears that smaller groups are developing larger programs. This suggests that smaller groups need some of the software methodologies developed for large-scale projects…” [FLW00]. The implication of this is that large-scale software design guidelines are becoming more and more relevant, as even small companies are capable of building large-scale software systems.

One final statement from Booch implies we should consider applying large-scale software design principles even to small software systems, because it is these systems that often grow into larger, unwieldy ones: "…I see in Java a phenomenon I've seen too many times before: simple systems that work well have a nasty way of evolving into big systems that sputter and breakdown and collapse of their own sheer weight. Furthermore, try to scale development techniques that work well for simple systems and you'll fail: the sustainable development of large complex systems requires fundamentally different techniques than heroic programming efforts offer" [Boo96, p.208].

Lakos expresses a similar view (Lakos 1996, p.xxvi) and indeed this is our view. We even have empirical evidence to support the notion that small systems often grow into large ones [MT07b]. The argument then is that design guidelines aimed at large-scale software systems should also be applied to small systems. The point of JooJ is to reduce the burden of applying *avoid dependency cycles* to Java code.

7.3 JooJ

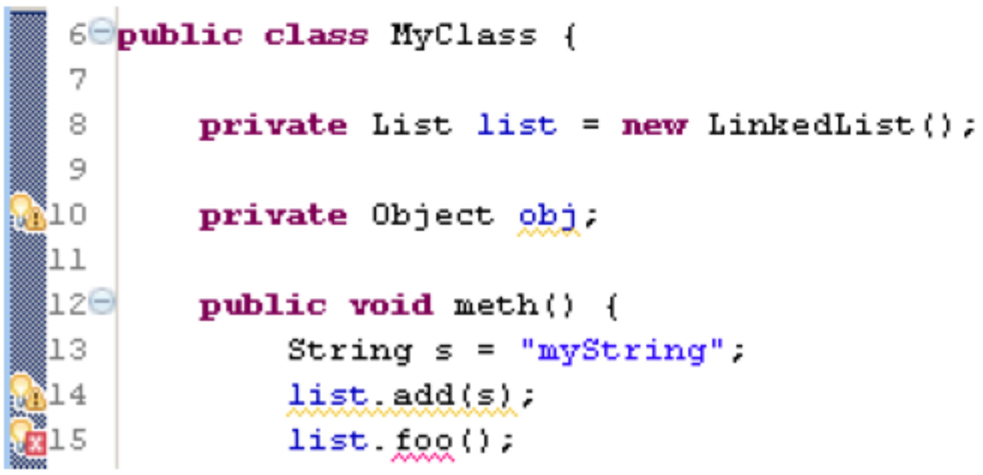
JooJ is a tool to support the design guideline *avoid dependency cycles*: (1) for new and existing Java code; (2) in real-time; (3) in an integrated fashion.

By 'new and existing Java code' we mean it supports code that is being written for a new system and code that is being written to maintain (e.g., extend or fix bugs) an existing system. We overload this phrase by also defining it to mean Java 5 (new) and Java 1.4 and earlier (existing). As we will see shortly there are different challenges in supporting the design principles for different versions of Java; and for new and existing systems.

By 'in real-time' we mean that Java code is analysed for the design guideline as it is being written. By 'in an integrated fashion' we mean that JooJ is an Eclipse plug-in that transparently extends the style and correctness checking that is already built in to Eclipse 3.1.1.

7.3.1 User Interface

JooJ's user interface (UI) is no different from that of Eclipse's built-in style and correctness checking. This means that using JooJ is non-invasive because Eclipse users are already familiar with its UI metaphor. We review the user interface of style and correctness checking in Eclipse order to put JooJ's UI in context.

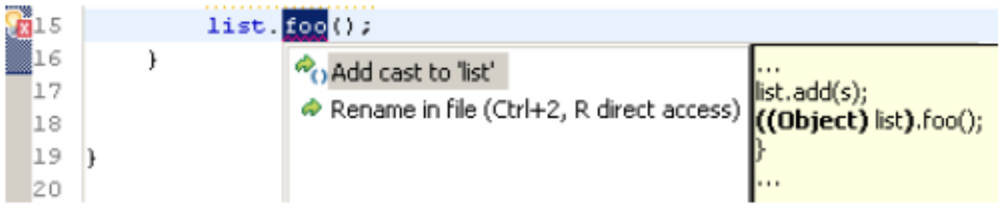


```
 6 public class MyClass {
 7
 8     private List list = new LinkedList();
 9
10     private Object obj;
11
12     public void meth() {
13         String s = "myString";
14         list.add(s);
15         list.foo();
```

Figure 2: Style and correctness checking in Eclipse

Figure 2 is a screen dump from Eclipse's Java editor. Besides the syntax highlighting it has several 'annotations' that are not available in standard text editors. The first of these annotations are the *squigglies*[25] on lines 10, 14 and 15. These squigglies indicate that there are problems with the code. The red squiggly on line 15 indicates a compilation error—*the method 'foo' is undefined for type List*. The yellow squigglies on lines 10 and 14 respectively indicate that *references to the generic type List<E> should be parameterised* and that *the field 'obj' is never read locally*. Although not shown in Figure 2 a description of the problem that leads to each squiggly appears as a tooltip when the mouse is hovered over it. Also not evident in Figure 2, but of particular importance is that the squigglies are continuously recomputed as text is typed into the Java editor.

[25] This is the term used for these wavy, coloured underlines in the Eclipse help documents



```
15            list.foo();
16      }
17
18
19  }
20
```

```
...
list.add(s);
((Object) list).foo();
}
...
```

Figure 3: Refactoring suggestions for style and correctness violations in Eclipse

Another annotation evident in Figure 2 is the appearance of lightbulbs icons in the left margin on the lines where squigglies occur. Clicking on the lightbulb of line 15 causes a popup to appear as shown in Figure 3. This popup is referred to in the Eclipse documentation as *code assist* or *content assist*. The code assist in Figure 3 presents a list of refactorings that can be performed to correct the problem. In the case of line 15 the code assist is suggesting casting the variable reference list to a subtype in the hope that a subtype of list's declared type declares a method foo(). The yellow tooltip to the right of the code assist shows the text that will result as a consequence of performing the selected refactoring.

The user interface we are building for JooJ is no different to that illustrated above. If a statement causes a cyclic dependencies then it gets a squiggly under it. If the dependency is in a Strongly Connected Component (SCC) (of size >1) then it gets a orange squiggly beneath it. If the dependency is in the Edge Feedback Set (EFS) computed by JooJ then it gets a magenta squiggly beneath it. Both SCC and EFS are discussed below.

In terms of the lightbulb annotations that provide specific code transformations to fix problems we are also currently in the process of extending JooJ to support the specific refactorings proposed by Lakos (e.g., escalation, demotion, dumb data, manager class etc)[Lak96, ch.5] for breaking cycles. This is actually a difficult problem because as we noted in Section 2 it is often the case that a cycle inducing statement has many dependent statements in the context of its source file. In order to remove the cycle inducing statement we must also move its dependent statements.

7.3.2 SCC and EFS

A subgraph S of another (directed) graph G is a Strongly Connected Component (SCC) if all of the vertices in S are mutually reachable in G and no additional vertices can be added from G to S that meet this criterion. A vertex is considered reachable from itself. In the context of our problem the vertices of S are classes that are all cyclically dependent, and indeed this is why it is SCCs in which we are interested.

A Minimum-Edge Feedback Set (MEFS) is the smallest set of edges that when removed from a (directed) graph G cause it to become a Directed Acyclic Graph (DAG). Equivalently it makes G a graph with SCCs all of size 1. In the context of our problem the MEFS represents the smallest set of dependencies that when removed break all cycles.

In JooJ the SCCs are computed using an linear-time algorithm presented Cormen et al.[CLR90, p.489]. Its cost (and implementation) is roughly equivalent to two depth-first searches. Computation of a 'small' edge feedback set is done in JooJ using linear-time a heuristic proposed by Eades et al[ELS93]. We call the output of Eade's algorithm a *mEFS* to distinguish it from a *MEFS* — the computation of which is NP-complete (Skiena 1998).

7.3.3 Dependency Removal

There are different challenges for eliminating dependency cycles from newly written code and code that is part of an existing system. If a system is built from scratch using JooJ as a design critic then it is likely that every cycle that appears in the system can be eliminated by the developer the instant it appears.

In existing systems however there are often many classes in large SCCs [MT07b]. As discussed in Section 2, there are domain-dependent dependencies that should never be removed. JooJ maintains an "exclusion set" of dependencies specified by the user that are never included in the edge feedback set it computes for each SCC.

To support specification of the exclusion set, and to generally support the user understanding the structure of the dependencies, JooJ provides a visualisation of the source types on which a class depends using JUNG[26]. In the visualisation, types are depicted as vertices (labeled with their fully qualified names) and edges represent dependencies. Edges are coloured differently depending on their membership—if an edge is in the edge feedback set it is magenta, if any other edge participating in a SCC it is orange, and other edges are black. A user of JooJ can add to the exclusion set by selecting edges in the visualisation.

7.4 High-Level Operation

JooJ is able to detect cycles among the classes defined in an application's .java files. It does not need to deal with classes defined in external libraries (such as the API) because if these libraries are truly external their classes cannot have any compilation dependencies on the application's classes. Also, as in previous work[MT07b], JooJ only considers dependencies within the body of a class; and the dependencies of nested classes and inner classes are merged with their top-level counterparts. In this way redundant import statements causing dependency cycles are ignored. This is desirable because the dependencies caused by redundant import statements are superficial; and Eclipse already has a feature to eliminate these redundant imports.

JooJ models a project's dependencies with the following data structures:
• A map from an Eclipse resource identifier (which is stable across Eclipse sessions) for a compilation unit to the top-level classes that this compilation unit defines. Call this map R, as in resource.
• A map from fully qualified top-level class names to the fully qualified names of that class's direct supertypes. Call this map S, as in supers.
• A map from fully qualified top-level class names to the fully qualified names of the classes it directly depends on. Call this map D, as in depends on.
• A map from resource identifier for a compilation unit to the latest filesystem timestamp for its corresponding file. Call this map T , as in timestamp.
• A list of SCCs.
• The mEFS for the current SCC.

[26] `http://jung.sourceforge.net/`

During an Eclipse session a project's .java files are opened in the Java editor, examined and modified. As these events occur Eclipse notifies JooJ and it updates its internal data structures to keep the dependency data structures consistent with the changing .java files. The events and updates they cause are described below.

Startup. When JooJ is attached to a particular project it first determines if it has been attached to that project before. If this is the first time the project has been seen by JooJ then all of the .java files in the project are turned into ASTs one-by-one and the dependency data structures are populated for the first time. This can take several minutes, and is discussed further in Section 5.

If JooJ has processed the project before then the dependency data structures are loaded from text files stored in the project's directory (see the shutdown event). Sometimes a .java file has been changed outside Eclipse, between Eclipse sessions. JooJ detects this situation by comparing the filesystem timestamp of each .java to that in R. Changed files have to have their dependencies recomputed as if they were modified in an Eclipse session. The types of changes that can happen to a file discussed shortly.

File Contents Changed. If a file has been modified then JooJ leverages Eclipse's Java Development Tools (JDT) API to turn a .java file into an Abstract Syntax Tree (AST). It then visits the AST in order to determine the modified .java file's new dependencies (i.e., its top level type, its super types and the other source classes it *DependsOn*). There are several different types of changes to a .java file and the way they affect the dependency data structures are explained below:
• *Dependencies for compilation unit unchanged*. If a file is changed it is possible that no new type was added to it, and that no types were added or removed from usage in it. We can easily determine this by comparing the supertypes and dependencies of the changed class, to that stored in S and D respectively. If they are unchanged we need not take any further action except to update the positions of the squigglies in the user interface.
• *Dependencies for compilation unit added.* If a dependency is added then we need to update one or more of the maps. If the dependency is added as a supertype we need to update S and D. If the dependency is added in the body of the class then we need

to update just D. We also need to update the SCC set if the dependency is not already in the class's SCC. We need to recompute the class's SCC's mEFS.

• *Dependencies for compilation unit removed.* Update S and D and recompute SCC and mEFS.

• *Fully qualified name of top-level type changed.* This situation occurs when the class's package is changed, or the top-level type is renamed. In this situation the .java file containing the class will eventually have to be renamed or moved directories in order for it to compile. Eclipse models the movement/renaming of files as a remove and then add event. Thus we discuss this situation under the guise of these events.

File Added. Sometimes a new source file is added to a system. In most cases this does not affect the bindings existing files. However if we refer to a type in a source file before we create that type then adding a new .java file (and its type) can affect the dependencies of other files. So when an new type is added we leverage Eclipse's Java Search facility to find existing references to this type and update the R, D and S correspondingly. After this we compute the SCC and mEFS for the newly added type.

File Removed. Sometimes a source file is removed from the system. Usually this means that a type is removed from the system, unless the same type is declared in two different source files. So we examine R to ensure the type has been removed from the system (i.e., it isn't declared in other .java files). If it has been removed we update R, S and D to remove all references to the removed type.

File Renamed/Moved. As previously stated Eclipse models this as a removal of a file and the addition of a new one. These are the canonical events that JooJ receives from Eclipse so the updates to the dependency data structures for this situation have already been discussed.

Shutdown. JooJ writes all the dependency maps to the project directory on disk. This saves time during the next startup because the dependencies for each .java file do not have to be recomputed from scratch.

7.5 Evaluation

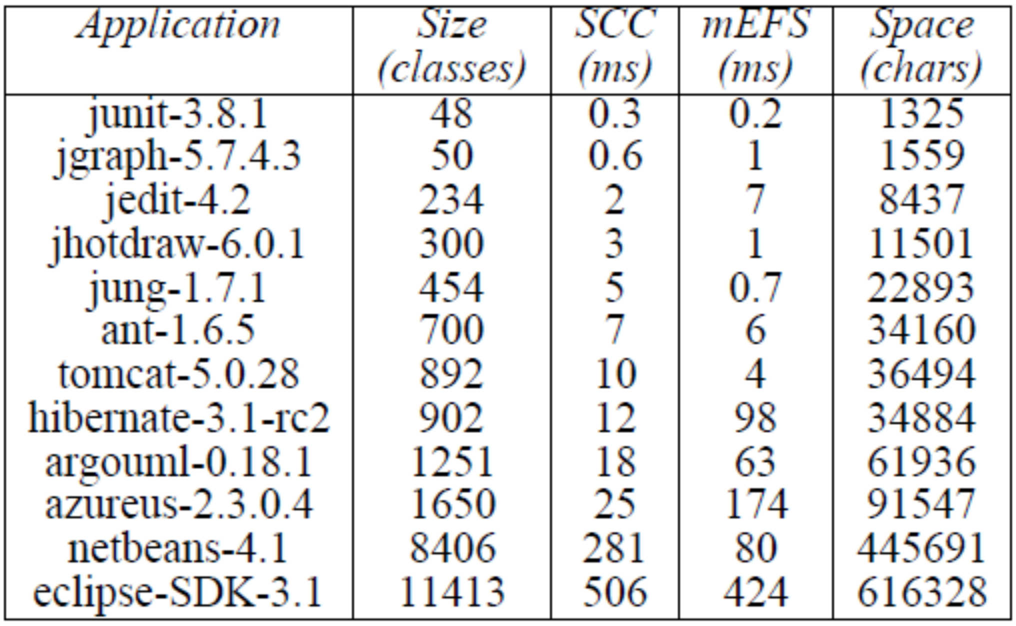
7.5.1 Performance

Much of the design of Eclipse has been influenced by a desire to make it scalable so users can leverage it to develop even large projects comprising thousands source files[AL04, p.338]. Scalability is of particular importance to JooJ because the design principle it supports is primarily for large scale systems. In this section we evaluate the runtime performance of the algorithms implemented by JooJ on 12 open source projects ranging in size from 48 to 11,413 .java files. All of these benchmarks were done on a machine with fairly modest 'specs'—an Intel P4 3.2 GHz with 1GB of RAM running Windows XP SP2.

We computed these benchmarks by writing a small program to load the dependency text files stored by JooJ in each project's directory into memory. We were then able to run the algorithms on the data structures populated with the information from these text files. The data structures used were identical to those implemented in JooJ. The recorded running time of the algorithms does not include the time taken to load the text files.

7.5.1.1 Algorithms

The time taken in milliseconds to compute all the SCCs from the internal data structures used by JooJ is shown in the 'SCC' column of Table 1. This was computed by timing 100 consecutive runs of the algorithm and taking the average. Recalling the SCC algorithm previously described we can infer that the cost of this algorithm is about the same as the cost of two DFSs.



Application	*Size (classes)*	*SCC (ms)*	*mEFS (ms)*	*Space (chars)*
junit-3.8.1	48	0.3	0.2	1325
jgraph-5.7.4.3	50	0.6	1	1559
jedit-4.2	234	2	7	8437
jhotdraw-6.0.1	300	3	1	11501
jung-1.7.1	454	5	0.7	22893
ant-1.6.5	700	7	6	34160
tomcat-5.0.28	892	10	4	36494
hibernate-3.1-rc2	902	12	98	34884
argouml-0.18.1	1251	18	63	61936
azureus-2.3.0.4	1650	25	174	91547
netbeans-4.1	8406	281	80	445691
eclipse-SDK-3.1	11413	506	424	616328

Table 1: Algorithm performance

The time taken in milliseconds to compute the mEFS for all the SCCs in each applications dependency graph is shown in the 'mEFS' column of Table 1. Again this was computed by timing 100 consecutive runs of the algorithm and taking the average. Recall that our implementation of this algorithm takes SCCs as input. We do not include the time taken to compute these SCCs in this measurement since this is already shown in the 'SCC' column.

So from the results in the 'SCC' and 'mEFS' columns of Table 1 we can infer that the absolute worst case for computing a class's SCC and the mEFS for that SCC is the sum of these two values. For Eclipse, we could (in the worst case) expect close to a 900ms delay after we change a file and its dependencies have been computed before we can update the statements in the Java editor with squigglies if they are causing cycles. We think that even this worst case delay is acceptable because writing code is inherently slow—we find we spend a lot of time staring at the screen thinking compared with actual typing.

But the worst case is not the typical case. As we described in Section 4 we do not have to recompute a class's SCC and its mEFS after every change to that class. Many times a dependency added to a class is already in the SCC so we can skip computing this and only have to compute the mEFS. Furthermore, we do not have to compute the mEFS for all SCCs, like we did for the benchmark. We only have to compute the mEFS for the SCC the class is involved in. If the SCC is small (e.g., 50 classes) then the mEFS algorithm takes only a few milliseconds, as if it were computing all the SCCs for a small application like junit, jgraph, jedit or jung.

7.5.1.2 Data Structures

JooJ maintains a 'master list' of strings representing the top-level types declared in the application. When the dependency data structures are populated the strings are drawn from this 'master list' so we can have equality-by- reference semantics for our DFS algorithm; and so we can reduce the amount of memory JooJ requires for each project. Table 1 shows the space requirements in number of characters for each of the applications. This was computed by concatenating all the strings in the 'master list' for each project and calling length() on it.

The size of Eclipse 3.1's 'master list' is about 600,000 characters (as shown in Table 1). If we remove this 'master list' and allow different instances of the lexically equal string then we have found that the space required for the strings in JooJ's dependency data structure for Eclipse can grow to about 6,000,000 characters. So maintaining a 'master list' can reduce the space demands of JooJ (at least in terms of strings) by a factor of 10. In fact, by maintaining a master list of strings it may be the overhead of the data structures (i.e., the HashMaps and LinkedLists that dominate JooJ's space requirements for a project.

7.5.1.3 Eclipse API

The SCC and mEFS algorithms are really only half the story when it comes to performance. These algorithms operate on adjacency list representations of class dependency graphs. The actual dependencies must be computed from the text of .java files. In order to do this we leverage Eclipse's JDT. We use the JDT to create ASTs and use *bindings* in order to resolve a name to the type to which it refers. It is well-documented in the Eclipse API that bindings are expensive (time and space-wise) to create. But we must recompute the bindings for a .java file each time it is changed so we need to know how long this takes.

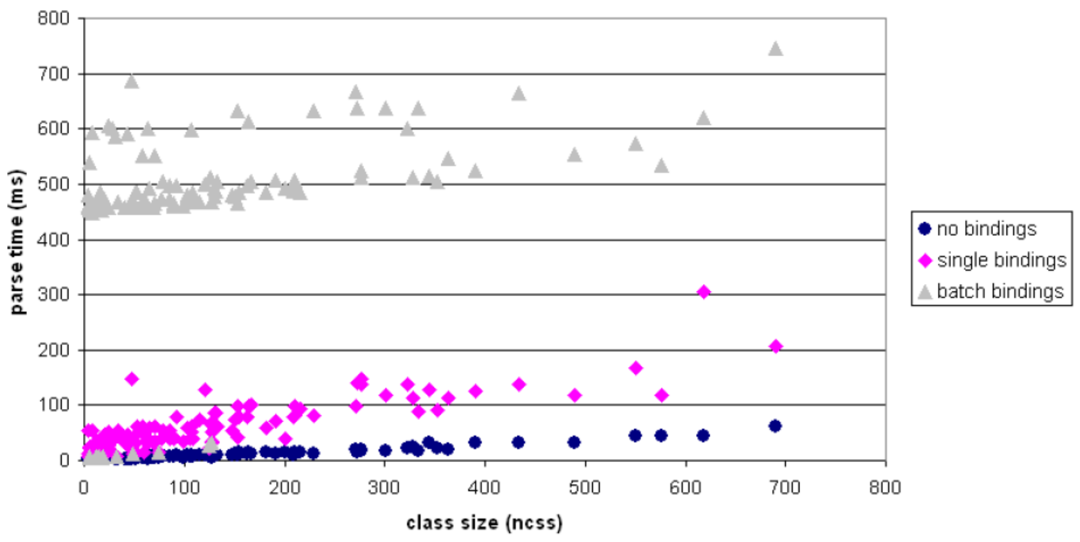


Figure 4: Time to create ASTs for a sample of .java files

Figure 4 shows the time taken to create an AST from a .java file using Eclipse's ASTParser class. The files were chosen at random from Ant — we couldn't easily select a hodgepodge of files from different applications because a source file requires the context of its application in order to compile (and compute bindings). The x-axis of the graph represents the size of the class in non-comment source statements (found by counting ';' and '{' characters not in comments). The y-axis represents the time taken (ms) to construct an ASTParser instance, create an AST, and visit the ASTs bindings to determine its dependencies.

There are 3 series on the graph that correspond to three options in using ASTParser. The first series 'no bindings' shows the amount of time taken to create an AST without bindings. This is a baseline so we can see how much bindings actually cost. The next series shows what we term'single bindings' because it uses the ASTParser in a way appropriate only for a single compilation unit (using the setSource and getAST methods). The next series 'batch bindings' shows the performance of the ASTParser when it is to parse just a single file in 'batch' mode (by calling the createASTs method). Interestingly using batch mode for a single file appears to be much slower than using it for a single compilation unit. This was not stated in the API, and indeed we only discovered the single compilation unit mode late in the development of JooJ.

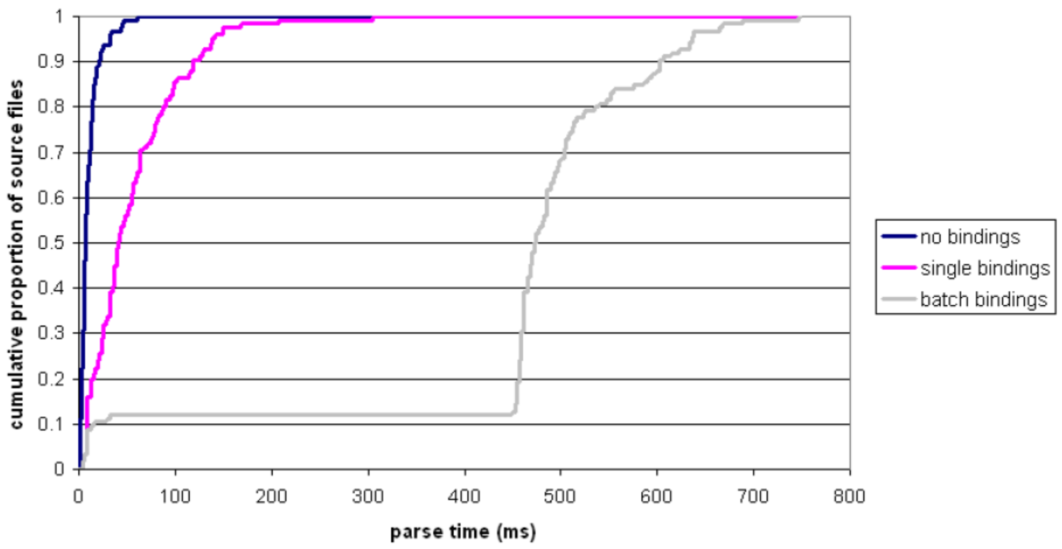


Figure 5: Distribution of AST creation times with and without bindings

Figure 5 is another view of the data in Figure 4. In this plot the x-axis represents time (ms) to create the AST under each of the conditions. The y-axis represents the proportion of .java files from our sample that will have parsed within the given time. So we can see from this plot that using single bindings about 80% of source files will have parsed within 100ms. Almost 100% of source files will have parsed within 200ms.

There are some issues in collecting the data of Figures 5 and 4 that necessitate further discussion. Firstly Eclipse maintains a Least Recently Used (LRU) cache of a project's resources[AL04, p.338]. In order to ensure we were not measuring the time to load a resource from disk into memory we creating consecutively created 10 ASTs for each of the files but only measured the time taken to process the last 9. In this way we could be fairly sure that the .java file's contents was cached in Eclipse for our measurements. This is a reasonable thing to do because JooJ operates on the file a programmer is editing, which necessarily must be already in memory.

Finally, when JooJ is first attached to a project it must compute the bindings (dependencies) for all of the .java files in that project. We determined that doing this for Ant takes close to 25 seconds. This equates to an average of 36ms per file. The next time we attached JooJ to Ant it took less than 1s to load the text files on disk into memory and compare the time stamps of the loaded .java files to those JooJ wrote to disk when our Eclipse session was last terminated.

7.6 Related Work

7.6.1 ByeCycle

ByeCycle[27] is a tool that is very similar to JooJ in that it checks for cycles among classes in ‘real-time’. However the primary feature of ByeCycle is a visualisation of the cycles a class is involved in. Also the granularity of its updates appears to be limited to when a file is saved or loaded, not as code is keyed in.

We have used ByeCycle and found JooJ offers several advantages over it. JooJ allows the dependencies that create a cycle to be related back to their corresponding statements in the source code. JooJ also determines all cyclic dependencies among classes whereas ByeCycle condenses classes outside the current class’ package into packages. In this way it appears that only the packages on which the class directly depends are analysed meaning ByeCycle does not perform whole program analysis. Presumably this is due to the screen real estate available for visualisation of cycles.

Also JooJ computes a mEFS and uses this to aid a developer decision where in the source code to break a cycle. Finally JooJ computes both definitions of *DependsOn* [MT07b] (one for Java 1.4 and below, and one for Java 5) meaning it allows ‘necessary cycles’ (i.e. those expressing *intrinsic interdependency*) to be expressed in a type-safe fashion.

7.6.2 Design Level Tools

There are several tools available that do ‘real-time’ checking of a UML design. ArgoUML [RHR98] may well have been the first of these tools. It provides several types of design critics pertaining to correctness, completeness, optimisation, alternatives, evolvability, presentation, experience and organisation that are continually evaluated against a design. A ‘todo’ list continuously updated by these critics with suggestions for the improvement of a design.

[27] http://byecycle.sourceforge.net/

Egyed [Egy06]has also produced a tool *UML/Analyser* that checks the consistency of UML diagrams against one-another in 'real-time'. One of the motivators for his tool is that apparently the consistency critics in ArgoUML are not able to keep up with an engineer's changes to a large UML model. Both Egyed's tool and ArgoUML differ from JooJ in that that operate at the design phase, rather than the coding (or implementation) phase.

7.6.3 Batch-style Cycle Tools

There are a plethora of other batch-style cycle checking tools for Java. Classycle[28] searches for cyclic dependencies among the classes of a Java application by analysing bytecode. This is problematic because the system has to be in a compilable state for it run. JooJ does not require this because Eclipse's bindings work even in the presence of many forms of compilation errors.

JDepend[29] analyses .class files in order to find cycles among packages. Again this tool operates on bytecode files. Also it does not detect SCCs, only cycles found during a DFS: "cyclic dependency detection may not report all cycles reachable from a given package. The detection algorithm stops once any given cycle is detected". Hautus's Package Structure Analysis (PASTA) tool (Hautus 2002) is also geared towards finding cycles among packages. It provides a visualisation of the package structure and tries to, much like JooJ, identify the smallest set of dependencies required to break all cycles among packages. Again it should be noted there cycles among packages do not necessarily imply cycles among classes so these tools solve slight different problems. In a sense finding cycles among classes is a more fundamental problem because if there are large SCCs of classes then there cannot be an acyclic package structure[MT07a].

Lattix LDM [SJSJ05] is another Eclipse plugin we have discovered similar to JooJ. It allows detection of cycles and allows specification of the 'dominance' relation among packages. It differs from JooJ in that abstracts away from the actual source code with a table known as a Dependency Structure Matrix (DSM). Allowable and undesirable dependencies are shown in this matrix at apparently at the granularity of

[28] http://classycle.sourceforge.net/
[29] http://www.clarkware.com/software/JDepend.html

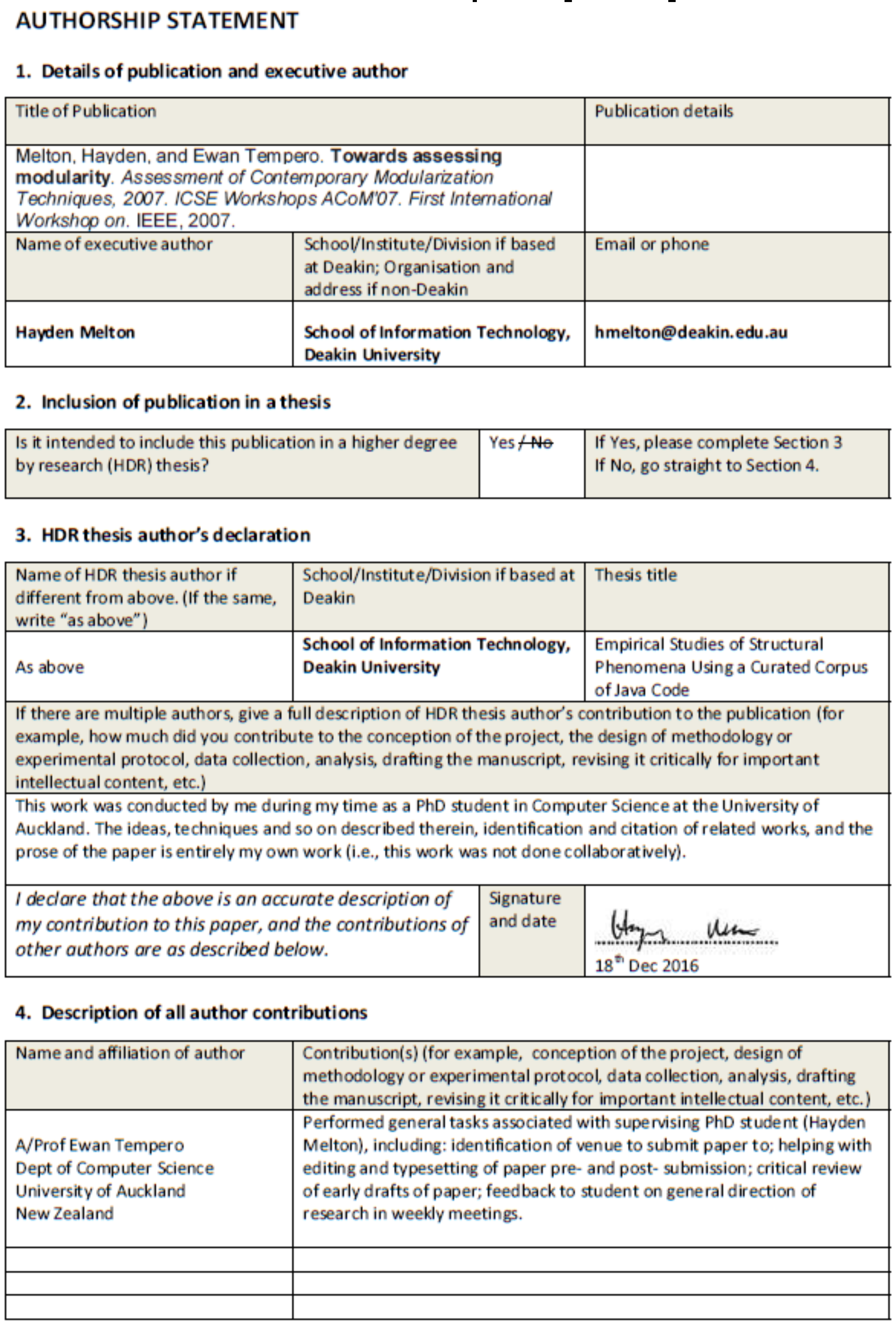
the package rather than class. It also appears that this tool does not do real-time checking (a press release states it can be "automatically synchronized with every build") and the UI appears to take over from the Java editor where code is typed. We think JooJ is a more effective poka-yoke device on the basis that it detects problems more quickly and at the activity that creates them (coding, not doing a software build).

7.7 Conclusions

We believe there should be real-time support for design guidelines that apply to the *whole program*. We have demonstrated the feasibility of doing so for the *avoid dependency cycles* design guideline by developing JooJ, an Eclipse plugin that provides real-time notification of violations of this guideline. In a broader context JooJ can be thought of as a poka-yoke approach to software quality assurance because it aims to prevent and detect violations of software design guideline, as or before they occur.

While we have established the feasibility of real-time cycle detection, determining its usability, that is, whether programmers will actually avoid dependency cycles, will require a higher quality implementation than the prototype we currently have. Producing such an implementation is currently underway. We would also like to expand JooJ to support other design principles that also require whole program analysis.

Coauthor Declaration for Chapter 8 [MT07e]



AUTHORSHIP STATEMENT

1. Details of publication and executive author

Title of Publication		Publication details
Melton, Hayden, and Ewan Tempero. **Towards assessing modularity**. *Assessment of Contemporary Modularization Techniques, 2007. ICSE Workshops ACoM'07. First International Workshop on*. IEEE, 2007.		
Name of executive author	School/Institute/Division if based at Deakin; Organisation and address if non-Deakin	Email or phone
Hayden Melton	**School of Information Technology, Deakin University**	**hmelton@deakin.edu.au**

2. Inclusion of publication in a thesis

Is it intended to include this publication in a higher degree by research (HDR) thesis?	Yes / ~~No~~	If Yes, please complete Section 3 If No, go straight to Section 4.

3. HDR thesis author's declaration

Name of HDR thesis author if different from above. (If the same, write "as above")	School/Institute/Division if based at Deakin	Thesis title
As above	**School of Information Technology, Deakin University**	Empirical Studies of Structural Phenomena Using a Curated Corpus of Java Code
If there are multiple authors, give a full description of HDR thesis author's contribution to the publication (for example, how much did you contribute to the conception of the project, the design of methodology or experimental protocol, data collection, analysis, drafting the manuscript, revising it critically for important intellectual content, etc.)		
This work was conducted by me during my time as a PhD student in Computer Science at the University of Auckland. The ideas, techniques and so on described therein, identification and citation of related works, and the prose of the paper is entirely my own work (i.e., this work was not done collaboratively).		
I declare that the above is an accurate description of my contribution to this paper, and the contributions of other authors are as described below.	Signature and date	[illegible] 18th Dec 2016

4. Description of all author contributions

Name and affiliation of author	Contribution(s) (for example, conception of the project, design of methodology or experimental protocol, data collection, analysis, drafting the manuscript, revising it critically for important intellectual content, etc.)
A/Prof Ewan Tempero Dept of Computer Science University of Auckland New Zealand	Performed general tasks associated with supervising PhD student (Hayden Melton), including: identification of venue to submit paper to; helping with editing and typesetting of paper pre- and post- submission; critical review of early drafts of paper; feedback to student on general direction of research in weekly meetings.

5. Author Declarations

I agree to be named as one of the authors of this work, and confirm:

i. that I have met the authorship criteria set out in the Deakin University Research Conduct Policy,

ii. that there are no other authors according to these criteria,

iii. that the description in Section 4 of my contribution(s) to this publication is accurate,

iv. that the data on which these findings are based are stored as set out in Section 7 below.

If this work is to form part of an HDR thesis as described in Sections 2 and 3, I further

v. consent to the incorporation of the publication into the candidate's HDR thesis submitted to Deakin University and, if the higher degree is awarded, the subsequent publication of the thesis by the university (subject to relevant Copyright provisions).

Name of author	Signature*	Date
Ewan Tempero	[illegible]	16/12/2016

6. Other contributor declarations

I agree to be named as a non-author contributor to this work.

Name and affiliation of contributor	Contribution	Signature* and date

* If an author or contributor is unavailable or otherwise unable to sign the statement of authorship, the Head of Academic Unit may sign on their behalf, noting the reason for their unavailability, provided there is no evidence to suggest that the person would object to being named as author

7. Data storage

The original data for this project are stored in the following locations. (The locations must be within an appropriate institutional setting. If the executive author is a Deakin staff member and data are stored outside Deakin University, permission for this must be given by the Head of Academic Unit within which the executive author is based.)

Data format	Storage Location	Date lodged	Name of custodian if other than the executive author
Qualitas Corpus of Java code, both binaries and source code with metadata (see *http://qualitascorpus.com/*)	University of Auckland, New Zealand	Starting 2006, onward	Ewan Tempero

This form must be retained by the executive author, within the school or institute in which they are based.

If the publication is to be included as part of an HDR thesis, a copy of this form must be included in the thesis with the publication.

Chapter 8 Towards Assessing Modularity

8.1 Introduction

It's noted in this workshop's call for papers that despite the emergence of a large number of "modularisation techniques" (e.g., aspects, design patterns, and so on), there are no standard approaches or "rules of thumb" for assessing the benefits and drawbacks of using these techniques in the construction of real software systems. In this paper we argue that the first step in assessing such techniques should be to determine their effect on modularity. Only then can we be sure that they have even been correctly classified as "modularisation techniques".

To determine the effect of a technique on a system's modularity we first need to agree on what *modularity* actually means. Despite modularity being a concept in software design for almost 50 years [Pre01], we still don't have a single, precise, widely-accepted definition for it [Fen94]. A consequence of this is that the claims that have been made about the effect of a technique on modularity (let alone other software quality attributes) are cryptic, and moving targets for systematic validation—any attempt to disprove them can be derailed by the claimants changing their favoured definition of modularity.

Most software engineering textbooks discuss modularity, but usually only in terms of its expected benefits and specific modularisation techniques; relatively few actually define it. The implication of this is that the dictionary definitions of modularity suffice for its meaning in software. Modularity is the extent to which something is modular, and many dictionaries define modular as *constructed with standardized units for flexibility and variety in use*[30]. We think the dictionary definition isn't particularly suitable for software because it's not clear (1) what "standardised" means; (2) what constitutes an increase in modularity: more standardisation, more flexibility, more variety of uses, or some combination of the above; and (3) if "flexibility" and "variety in use" entirely, or even accurately, describe the rationale for making a software system modular.

[30] www.dictionary.com

Some software-specific definitions for modularity are surveyed by Booch [Boo91, p.49-53]. On closer inspection the "definitions" Booch surveys aren't really definitions at all though—just like what's said in most textbooks, they're discussions of its benefits and techniques for achieving it. Booch's own definition, that *modularity is the property of a system whose modules are cohesive and loosely-coupled*, is problematic too because *cohesion* and *coupling* are themselves only loosely-defined.

8.2 Definition, Usage and Assessment

The definition of modularity we advocate is *the degree to which something comprises discrete (or independent) parts* [IEE90]. It's a good definition because it's concise, yet it doesn't mislead or constrain us in the particular rationale we have for making a system modular. It also makes clear what constitutes an increase in modularity: an increase in the number of parts that can be considered independent from one another.

The flipside of this definition is that we must be careful in our usage of the term *modularity* by always specifying a perspective. A system that comprises parts that can be considered independent from one perspective (e.g., unit testing) may not comprise parts that can be considered independent from another (e.g., verbatim reuse of source files)[31].

The definition we advocate is superior to many in that it makes no judgement on the "goodness" of modularity. Contrary to what's implied by most of the literature, a system that's modular with respect to say change, does not necessarily mean changes made to it will require less effort—as noted by Fenton, modularity is an *internal* quality attribute [Fen94]. All we can say of such a system is that changes will be confined to relatively few modules. If a system is *too modular* [Pre01], then the

[31] The need to discuss modularity with reference to a specific perspective is also noted by Meyer [Mey95]. Unfortunately he claims that no single, concise definition of modularity is possible and goes on to define it from five perspectives that he considers total, thus unnecessarily constraining our perspectives on it.

effort required to make a change might be higher because the sheer number of parts it comprises might make finding the appropriate part to change more difficult.

To demonstrate an approach to assessing the effect a technique has on modularity consider two design principles that pertain to the overall structure of a system: (1) *avoid dependency cycles among source files* and (2) *favour a "flatter" rather than "taller" source file dependency graph*[32]. The application of these design principles in shown in Figure 1: in (a) neither of the design principles has been followed, in (b) only "avoid cycles" has been followed, in (c) both have been followed because its structure is both acyclic and "flatter" rather than "taller".

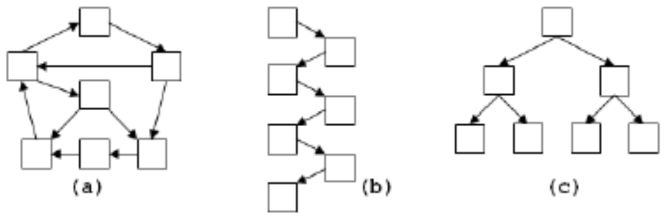


Figure 1. Source file dependency graphs of three different software systems.

Lakos argues why following these principles leads to systems that are easier to understand, test, and reuse [Lak96]. Although he makes no mention of modularity in his arguments, the way in which they're couched closely relates to the definition of modularity we advocated earlier. Thus, from certain perspectives, these design principles can be accurately classified as "modularisation techniques".

The crux of all of Lakos' arguments is that following the design principles leads to systems whose individual parts (source files) *transitively* depend on fewer other source files. A source file's transitive compilation dependencies play an important role in the extent to which we can consider it independently from the other source files in a system in *verbatim reuse*, *understanding*, *testing in isolation* and *integration testing* [Lak96]. We do not have the space to espouse the arguments for all of the activities mentioned so concentrate only on doing so for verbatim reuse of source files [Lak96].

[32] In our recent work we've been looking at these principles from an empirical perspective [MT07b].

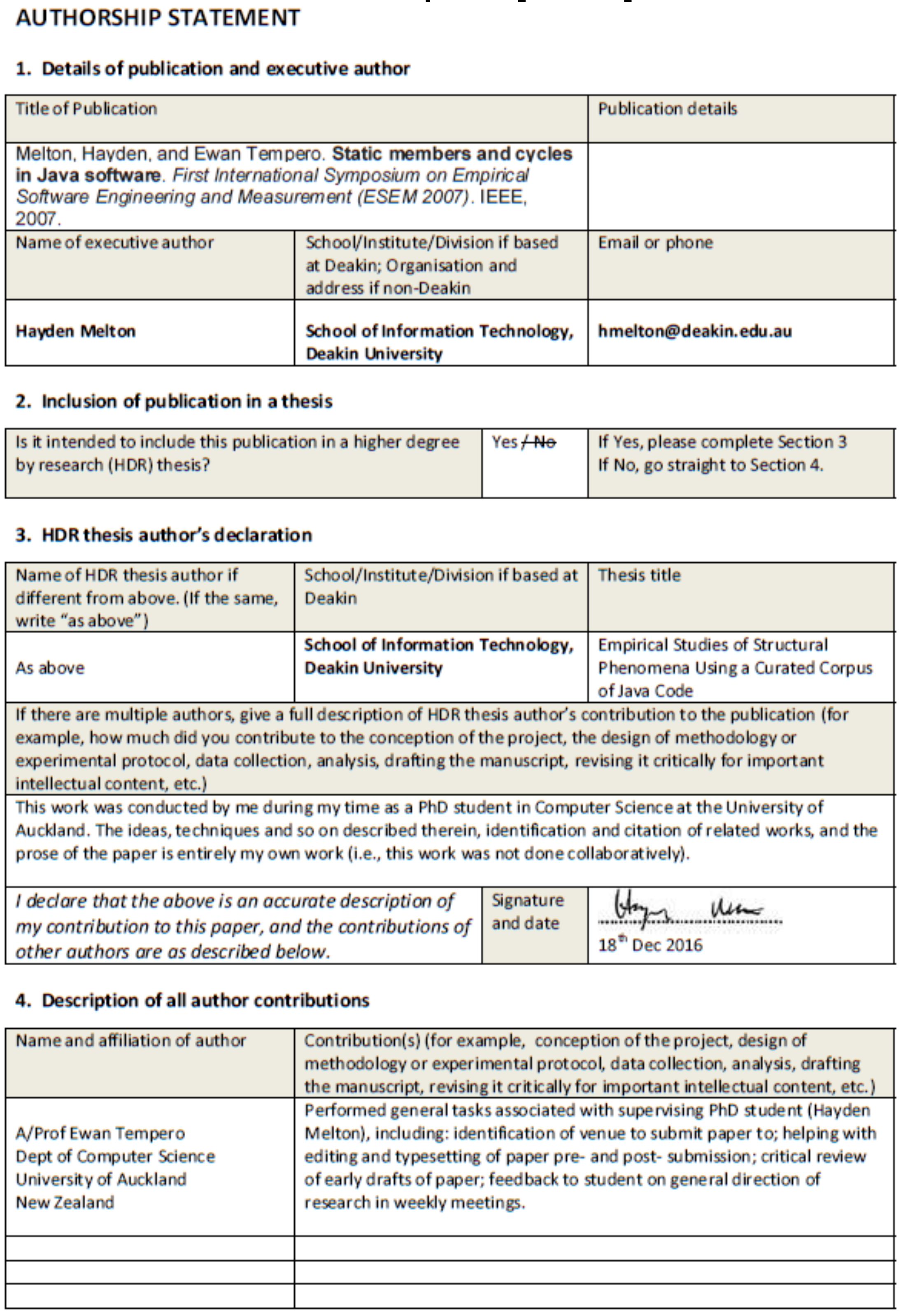
Verbatim reuse of source files is about deploying a source file from one system in the context of another without (1) modifying its text to eliminate any of its dependencies or (2) introducing stubs to artificially satisfy its dependencies. To ensure a source file successfully compiles in the context of the new system we must also deploy all the source files on which it depends, and all the source files on which the others depend, and so on. So to reuse a source file in this way we have to deploy all the other source files on which it *transitively* depends. If we look at the system of Figure 1(a) there are no source files we can reuse independently of any of the others; in (b) there's one that can be reused entirely independently of all the others; in (c) there's four. Furthermore, to reuse the average source file from (c) we'd have to deploy fewer other source files than for the average source file in (b), and in turn from (c). So according to both of these criteria, which are essentially two simple metrics for modularity, and with respect to this form of reuse, (c) is more modular than (b), which in turn, is more modular than (a).

Though we can rank the systems of Figure 1 by modularity from the perspective of verbatim reuse, this ranking does not necessarily match that we'd get if we ranked the systems by the *external* software quality attribute of *reusability*. This is a criticism of the position we've taken in this paper—nothing can be said from it on the effect of a technique on external software quality attributes, which are the things we really care about. Fenton et al. argue criticizing the measurement of internal product attributes (such as modularity) for not *initially* being shown to be predictors of external quality attributes is not helpful though, because without good measures of internal attributes we have little hope of subsequently developing models for such prediction [FM96].

8.3 Conclusion

In this paper we've argued that our first step in assessing a "modularisation technique" should be in determining if it is even correctly classified as such. We've advocated a specific definition of modularity to allow us to do this, and have shown the definition to be practical by demonstrating an approach for assessing the effect two design principles have on it.

Coauthor Declaration Chapter 9 [MT07d]



AUTHORSHIP STATEMENT

1. Details of publication and executive author

Title of Publication		Publication details
Melton, Hayden, and Ewan Tempero. **Static members and cycles in Java software**. *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*. IEEE, 2007.		
Name of executive author	School/Institute/Division if based at Deakin; Organisation and address if non-Deakin	Email or phone
Hayden Melton	**School of Information Technology, Deakin University**	**hmelton@deakin.edu.au**

2. Inclusion of publication in a thesis

Is it intended to include this publication in a higher degree by research (HDR) thesis?	Yes ~~/ No~~	If Yes, please complete Section 3 If No, go straight to Section 4.

3. HDR thesis author's declaration

Name of HDR thesis author if different from above. (If the same, write "as above")	School/Institute/Division if based at Deakin	Thesis title
As above	**School of Information Technology, Deakin University**	Empirical Studies of Structural Phenomena Using a Curated Corpus of Java Code
If there are multiple authors, give a full description of HDR thesis author's contribution to the publication (for example, how much did you contribute to the conception of the project, the design of methodology or experimental protocol, data collection, analysis, drafting the manuscript, revising it critically for important intellectual content, etc.)		
This work was conducted by me during my time as a PhD student in Computer Science at the University of Auckland. The ideas, techniques and so on described therein, identification and citation of related works, and the prose of the paper is entirely my own work (i.e., this work was not done collaboratively).		
I declare that the above is an accurate description of my contribution to this paper, and the contributions of other authors are as described below.	Signature and date	[illegible] 18th Dec 2016

4. Description of all author contributions

Name and affiliation of author	Contribution(s) (for example, conception of the project, design of methodology or experimental protocol, data collection, analysis, drafting the manuscript, revising it critically for important intellectual content, etc.)
A/Prof Ewan Tempero Dept of Computer Science University of Auckland New Zealand	Performed general tasks associated with supervising PhD student (Hayden Melton), including: identification of venue to submit paper to; helping with editing and typesetting of paper pre- and post- submission; critical review of early drafts of paper; feedback to student on general direction of research in weekly meetings.

5. Author Declarations

I agree to be named as one of the authors of this work, and confirm:

i. that I have met the authorship criteria set out in the Deakin University Research Conduct Policy,

ii. that there are no other authors according to these criteria,

iii. that the description in Section 4 of my contribution(s) to this publication is accurate,

iv. that the data on which these findings are based are stored as set out in Section 7 below.

If this work is to form part of an HDR thesis as described in Sections 2 and 3, I further

v. consent to the incorporation of the publication into the candidate's HDR thesis submitted to Deakin University and, if the higher degree is awarded, the subsequent publication of the thesis by the university (subject to relevant Copyright provisions).

Name of author	Signature*	Date
Ewan Tempero	[illegible]	16/12/2016

6. Other contributor declarations

I agree to be named as a non-author contributor to this work.

Name and affiliation of contributor	Contribution	Signature* and date

* If an author or contributor is unavailable or otherwise unable to sign the statement of authorship, the Head of Academic Unit may sign on their behalf, noting the reason for their unavailability, provided there is no evidence to suggest that the person would object to being named as author

7. Data storage

The original data for this project are stored in the following locations. (The locations must be within an appropriate institutional setting. If the executive author is a Deakin staff member and data are stored outside Deakin University, permission for this must be given by the Head of Academic Unit within which the executive author is based.)

Data format	Storage Location	Date lodged	Name of custodian if other than the executive author
Qualitas Corpus of Java code, both binaries and source code with metadata (see *http://qualitascorpus.com/*)	University of Auckland, New Zealand	Starting 2006, onward	Ewan Tempero

This form must be retained by the executive author, within the school or institute in which they are based.

If the publication is to be included as part of an HDR thesis, a copy of this form must be included in the thesis with the publication.

Chapter 9 Static Members and Cycles in Java Software

The static modifier is a convenient way to make class members "global" in object-oriented software systems. Given this, we wondered if static members significantly contribute to the long dependency cycles among the classes that we observed in a previous empirical study of Java software. In this paper, we examine 81 open source Java applications. We find empirical evidence that classes that declare a non-private static field or method that is accessed from within another class are likely to be involved in dependency cycles.

9.1 Introduction

It is generally accepted in the software engineering community that software structure strongly impacts software quality attributes such as understandability, maintainability, reusability, and testability. Much of the advice about how to write "good" software is couched in terms of how to structure it (e.g., [Par72] [Par79] [Lak96]). Our interest is in better quantifying this supposed impact. As a first step, we have been looking at how real software systems are actually structured [BFN+06][MT07b]. Now we are interested in identifying causes of different structural phenomena. In this paper, we investigate the extent to which static members of classes contribute to cyclic structures in software.

We are interested in looking into the causes of dependency cycles because many authors have presented compelling arguments for how cycles are detrimental to specific software quality attributes, including understandability, testability, reusability, buildability and maintainability [Par79] [KGH95b] [Lak96] [Mar96b] [RBP+91] . Despite this purported detriment, in a recent empirical study we found that long dependency cycles are common among the classes of both open-source and commercial Java systems [MT07b]. This leaves us to wonder, with all the advice to the contrary, why and how cycles get created. If we could identify aspects of software development that increase the tendency of cycles being created, then we could better mitigate those factors and so reduce cycles in software.

Languages such as C++, Java, and C# have the concept of a *static member*. Such members are shared across all instances of a class. Programmatically we can access static members through the name of the class; there is no need to obtain a specific runtime instance of that class in order to access them. In a sense, this means it's easier to access a static member than a non-static one, from an arbitrary point in a system's source code. This "ease of access" is what makes us wonder if static members play a significant role in the creation of cyclic dependencies among classes. The study we present in this paper is a first attempt to investigate this hypothesis in the context of Java software.

The rest of the paper is organised as follows. In the next section, we expand upon the discussion above as to why there might be a relationship between cycles and static members. Section 3 gives the details of how we carried out our study, the results of which are presented in Section 4. We conclude with Section 5.

9.2 Background and Motivation

In our prior empirical study we looked at cycles among classes in several different dependency relations [MT07b]. The two dependency relations relevant here are the USES and USES-IN-THE-INTERFACE relations. The USES relation captures a class' entire compilation dependencies. The USES-IN-THE-INTERFACE relation captures the compilation dependencies a class has that are visible to its clients (i.e., the types that appear in the return types, formal parameters and throws clauses of its non-private methods, the declared types of its non-private fields, and its direct supertypes). From a conceptual perspective the USES-IN-THE-INTERFACE is meant to reflect cycles that are difficult to avoid, or cannot be sensibly broken (see [MT07b]). A finding of our study was that, for almost all of the applications examined, cycles in the USES-IN-THE-INTERFACE relation were smaller, and involved far fewer classes than cycles in the USES relation. We noted that this meant that types appearing *only* in the private part of a class, and not even transitively in the class' USES-IN-THE-INTERFACE relation, must contribute significantly to a class' participation in cycles.

```
public class A {                public class B {
  public m(B b) {                 public C makeC() {
    B bee = b;                      return new C;
    C c = b.makeC();              }
    D d = (D)b;                 }
    E.staticMethCall();
    F f = new F();              public class C {}
  }
}
```

USES-IN-THE-INTERFACE(A) = {A, B}
USES-IN-THE-INTERFACE*(A) = {A, B, C}

Figure 1. Java code to illustrate some ways that a type can appear only in the private part of a class and not even transitively in its USES-IN-THE-INTERFACE relation.

In order to see some ways in which a type may appear *only* in the private part of a class, and not in that class' USES-IN-THE-INTERFACE relation (even transitively) consider the Java code in Figure 1. Below this Java code we show the computation of the USES-IN-THE-INTERFACE and its transitive closure(*) for class `A`. We can see `A` depends on several types that do not appear in the transitive closure of its USES-IN-THE-INTERFACE relation: `D`, `E` and `F`. The dependency on `D` is ultimately due to a type-cast, the dependency on `E` is due to the access of a static method, and the dependency on `F` is ultimately due to instantiating this type via `new`.

Our primary interest is in determining the extent to which the access of static members (fields and methods) contribute to cycles but we will also briefly look, in later sections, at the extent to which object instantiation (i.e., the use of `new`) and type casts play a part in cycles. We concentrate on static members because there is a simple theory for why classes with non-private static members might be more likely to be involved in cycles, and also because of the negative connotations the `static` modifier has in the software design community. Static members are a convenient way of making things "global", and the widely-held belief is that "global is bad". Rather than make things global, conventional wisdom is that we should design the components of a software system around the principle of "information hiding" [Par72].

There's a growing feeling in object-oriented programming circles that static members are overused. Kerievsky's claim is that the Singleton pattern is overused, and since its typical implementation involves using a `static` field and/or method it can be argued that ultimately its static members that are overused. Kerievsky reports that Cunningham has said that while the notion of singularity is an important aspect of software design, the use of Singleton "seems to have grown out of proportion". Kerievsky reports that Beck too, has said of Singletons, "they give you a good excuse not to think carefully about the appropriate visibility of an object". Kerievsky himself has implied that Singletons are overused by coining the term "Singletonitis" to refer to the condition where a developer is addicted to using Singleton [Ker04]. Because cycles are thought to be detrimental to several specific software quality attributes [Par79][KGH95b][Lak96][Mar96b][RBP+91], a correlation between static members and cycles provides additional evidence that static members are also "bad".

9.3 Methodology

Given that static members are seemingly so easy to access from anywhere in a system's source code, and that long cycles among classes usually are not due to dependencies appearing in the public parts of a class alone (i.e., in the USES-IN-THE-INTERFACE relation), we would like to quantify the extent to which the access of these members contribute to dependency cycles among the classes of a system. Our hypotheses for this empirical study of static members and cycles are thus:

H1 Classes that are accessed statically are more likely to be involved in dependency cycles than classes that are not accessed statically.

H2 There are dependencies in the cycles that are due to access of static members.

The second hypothesis, H2, requires some further explanation. Figure 2 is the dependency graph of a small software system, comprising 9 classes. In this dependency graph 3 classes are accessed statically, and 6 are *not* accessed statically. Also, 6 classes appear in a dependency cycle; 3 are not involved in any dependency cycles. If there was an equal likelihood of classes that were accessed statically appearing in cycles as classes that were not accessed statically appearing in cycles then we would expect only 2 class that is accessed statically in this system to be involved in a cycle. This is because 6/9 classes are involved in cycles, and 3 classes are accessed statically, and 6/9 × 3 = 2. So in this system, classes that are accessed statically are overrepresented in cycles, i.e., the likelihood of a randomly selected statically accessed class appearing in a cycle in this system is higher than that of a randomly selected class that is not statically accessed. This supports H1. The problem is—and this is where H2 comes in—that none of the edges (dependencies) due to static access are actually contributing to the cycle, i.e., these edges do not appear on the cycle's path. This means our causal explanation for why classes that are accessed statically are more likely to become involved in cycles is not supported, thus illustrating the need for the second hypothesis.

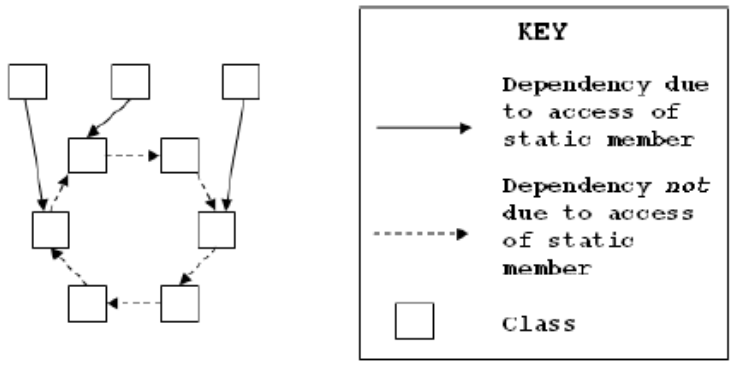

KEY
Dependency due to access of static member
Dependency *not* due to access of static member
Class

Figure 2. Dependency graph of a small software system.

In order to test our hypotheses, and as per our prior empirical study of cycles, we collected metrics from an (ever growing) corpus of Java software. For this paper we used 81 applications—a superset of those open source applications used in our prior study. Unfortunately we were not able to use the closed source applications from our prior study because of particulars in the intellectual property agreements we had for them. As we are continually adding to (and updating) our corpus of Java software, in this paper we used some more recent versions of the open source applications than in our prior study of cycles, however all the cycle data has been regenerated for the versions used in this paper. A list of the applications used in this study is shown in Tables 6 and 7. In the following subsections, we describe the specific metrics we collected, and the statistics and visualisations we used to show that access of static members contributes to cycles.

9.3.1 Metrics

As per our prior study of cycles we define a class to be involved in a cycle if it participates in a non-trivial Strongly Connected Component (SCC) in the program's dependency graph. The dependency graph to which we refer is the same as was used in our prior study—it involves only top-level classes defined in the application's source files (the dependencies of nested classes are merged with their top level counterpart's), and does not include edges due to redundant import statements. We refer the reader to our prior study [MT07b] for the rationale for this particular dependency graph and considering SCCs, and not say, *simple cycles*. In addition to whether or not a class participates in a non-trivial SCC, we can quantify the size of the cycle it participates in with the metric of number of nodes (classes) in the SCC to which it belongs.

In this paper, we say that a class is "accessed statically" if it declares a non-private (i.e., `public`, `protected` or default access) static method or field that is accessed from within the source code of a *different* class (i.e., a different node in the program's dependency graph). It is relatively easy to determine if a class is accessed via a call to a static method or reference to a static field through analysis of Java bytecode. We extended the tool we used in our prior study of cycles, Jepends-BCEL[33], to determine static access in this way. It was fairly easy to do because there are special instructions in byte code that pertain to static members: they are represented in Byte Code Engineering Library (BCEL)[34] by the classes: `INVOKESTATIC`, `PUTSTATIC` and `GETSTATIC`. Similarly we looked at dependencies due to instantiation (i.e., the use of `new`) which is represented in the BCEL model of byte code instructions with the `NEW` class.

We will see in Section 3.5 that we controlled for size, since class size could be a confounding factor in the participation of classes that are accessed statically in cycles. In order to do this we also collected the number of methods a class actually declares (cf. inherits). We counted constructors as methods, and only declared methods because this was most-closely aligned the BCEL object model of a Java class. We could have collected other measures of size such as number of byte code

[33]http://www.cs.auckland.ac.nz/~hayden/software.htm
[34]http://jakarta.apache.org/bcel/

instructions, number of lines of code (from source code), number of fields and so on, but it is unlikely that this would have affected our results much. A study by Bieman et al. across 5 Java applications found that number of lines of code, number of methods and number of fields were all strongly correlated with one another [BSW+03].

9.3.2 Statistics

In order to test H1, we made extensive use of the χ^2 test, which is useful for determining if an observed distribution is different from an expected one. In order to compute the *expected* distribution of values we used the null hypothesis. The null version of H1 implies it's equally likely that classes that are accessed statically appear in cycles as classes that are not accessed statically. Under the null hypothesis, the proportion of statically accessed classes in cycles would be the same as the proportion of classes involved in cycles from the total. Analogous calculations are done for all other combinations of static access and cycle participation. We used Excel's `CHITEST` function to calculate the probability that the difference between the expected and observed populations was due to chance. If the probability was significant at the 0.05 or 0.01 levels we had to look at the direction of the inequality between the expected and observed values to see if our hypothesis was supported or not. In some cases we could not apply the χ^2 test because statisticians have argued that it should only be applied when all values in the expected population are ≥ 5. This means we could not compute the probability of the difference being due to chance using the χ^2 test, so have some blank results.

9.3.3 Testing H1 at the Application-level

To initially test H1 at the application level we broke down an application's classes on the basis of static access, and further broke them down on the basis of cycle participation. This breakdown for Eclipse is shown in Table 1. The expected values come from the null hypothesis—that statically accessed classes have the same likelihood of appearing in cycles as classes that are not statically accessed. So, based on the null hypothesis, to compute the number of statically accessed classes that we would expect to appear in a cycle we multiply the proportion of an applications classes we observed to participate in cycles (= (1029+3971)/11415) by the number of classes that are accessed statically (=1029+778). This yields 791.5. The other expected values are similarly computed.

		observed	expected
static	in cycle	1029	791.5
	not in cycle	778	1015.5
not static	in cycle	3971	4208.5
	not in cycle	5637	5399.5
total		11415	11415

Table 1. Static access and cycle participation for Eclipse.

Applying the χ^2 test to data in Table 1 gives a probability of 1.90×10^{-32}. So we can reject the null hypothesis, at both the 0.05 and 0.01 levels of significance. Looking at the direction of the inequality, we see that more statically accessed classes *actually* appear in cycles than we would *expect*, so we can conclude that our data supports our hypothesis at these levels of significance.

9.3.4 Testing Class Size and Cycle Participation at the Application-level

El Emam et al. criticise a large body of prior work on metrics for not taking into account the potentially confounding effect of class size on a metrics validity [EEBGR01]. Particularly they argue that while metrics like CBO appear to be correlated with fault proneness, the effect disappears when class size is controlled for. We want to see if large classes are more likely to appear in cycles than small ones. It certainly seems that this would be possible, since large classes would likely have more distinct dependencies on other classes than other classes, so would be more likely to participate in a cycle than a small class.

In order to test the large class hypothesis at the application level, we split classes about the median number of methods they declare, into small and large classes. Our aim was to get the most even split of classes between large and small so as to have the best chance of getting a significant result with the Chi-Squares test. This meant that sometimes we split on strictly less than the median, other times it meant splitting on less than or equal to the median. To see why consider the following data sets: $\{1,2,2,2,2,3,3,4\}$ and $\{1,2,2,3,3,3,3,3,5\}$. In the former the most even split comes from less than or equal to the median (which is 2). In the latter, the most even split of values between large and small comes from strictly less than the median (which is 3).

		observed	expected
large	in cycle	3059	2422.7
	not in cycle	2472	3108.3
small	in cycle	1941	2577.3
	not in cycle	3943	3306.7
total		11415	11415

Table 2. Class size and cycle participation for Eclipse.

Table 2 shows the breakdown of classes by size, and subsequently by cycle participation for Eclipse. In order to compute the expected values once again we assume that large and small classes have the same probability of appearing in cycles, so to compute the expected value for a large class appearing in a cycle we multiply the observed total number of large classes (=3059+2472) by the observed proportion of Eclipse's classes appearing in cycles (= (3059+1941)/11415) to yield 2422.7. Applying the Chi-squared test to this data gives a probability of 1.01×10^{-124}, which is significant at both the 0.05 and 0.01 levels. Additionally the direction of the inequality means that, for Eclipse, large classes are more likely to appear in cycles, so it supports our class size hypothesis.

9.3.5 Testing H1 at the Application-level while Controlling for Size

The approach we take for controlling for size involves stratifying on class size as described by El Emam et al[EEBGR01]. Table 3 shows the participation of classes accessed statically in cycles for *only* the large classes in Eclipse. We can see in this table that 5531 classes were considered large when we divided them up in the way described in Section 3.4. To calculate the expected values in this table we calculated proportions based on the population of large classes only. So to calculate the expected number of statically accessed classes appearing in cycles for this dataset, we multiply the total number of large, statically accessed classes (=838+322) by the proportion of large classes appearing in cycles (= (838+2221)/5531) which yields 641.6. Applying the χ^2 test to this data gives a probability of 1.09×10^{-36} which is significant at the 0.01 level. Since the direction of the inequality supports our hypothesis, we can conclude that the effect of static access on cycle participation still holds for large classes in Eclipse.

		observed	expected
static	cycle	838	641.6
	not in cycle	322	518.4
not static	cycle	2221	2417.4
	not in cycle	2150	1953.6
total		5531	5531

Table 3. Static access and cycle participation considering only large classes for Eclipse.

We can perform a similar analysis for small classes in Eclipse. This is shown in Table 4. Applying the χ^2 test to this data gives a probability of 0.266 so we cannot reject the null hypothesis. Thus we cannot be certain that the difference between the expected and observed populations is not due to chance.

		observed	expected
static	in cycle	191	213.4
	not in cycle	456	433.6
not static	cycle	1750	1727.6
	not in cycle	3487	3509.4
total		5884	5884

Table 4. Static access and cycle participation considering only small classes for Eclipse.

9.3.6 Testing H2 at the Application-level

In order to test H2 we generated stacked bar graphs for each application of the form used in our prior study of cycles[MT07b]. In this type of graph a system's classes are shown on the basis of their participation of cycles (SCCs) of growing sizes. Figure 3 shows the involvement of the Java Runtime Environment's (JRE's) classes in cycles when various dependency relations are considered. The bar-stack marked "I" shows cycles when only edges due to the USES-IN-THE-INTERFACE relation are considered; that marked "I+T" shows cycles when only edges due to both the USES-IN-THE-INTERFACE and access of static members are considered; that marked "I+N" shows cycles when only edges due to the USES-IN-THE-INTERFACE and instantiation are considered; that marked "I+T+N" shows cycles when only edges due to the USES-IN-THE-INTERFACE and access of static members and instantiation (i.e., the use of `new`) are considered; finally the bar-stack marked "A" shows cycles when all edges in the dependency graph are considered (i.e., cycles in the USES relation).

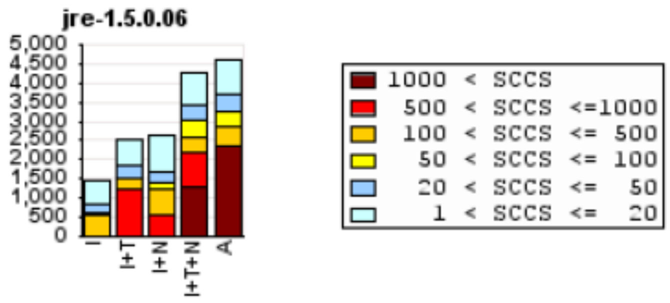


Figure 3. Cycles in JRE for different kinds of dependencies

From the graph of Figure 3 we can conclude that edges due to access of static members do contribute to both cycle size and number of classes participating in cycles in the JRE. This is because the bar-stack marked "I+T" is taller than that marked "I". The fact that the bar-stack marked "I+N" is shorter than that marked "I+T+N" also provides further evidence that edges due to static access are contribute to cycles in the JRE, because if we ignore these edges and only consider those due to `new` and USES-IN-THE-INTERFACE then cycles are smaller and fewer classes are involved in cycles. So for the JRE we can conclude that edges due to access of static members contribute to cycle size and cycle participation.

9.3.7 Testing Hypotheses at the Corpus-level

Besides testing the hypotheses at the application-level we can also test them at the corpus-level. This involves looking at the number of applications that have classes with a certain feature (e.g., static-access or large) being over or under-represented in cycles when compared to the entire population of an application's classes. Table 5 demonstrates how we can apply the χ^2 test to this data. If the null hypothesis were true we would expect that half of the applications would have the feature (marked "X" in the table) over-represented in cycles, and half of them not to have the feature over-represented in cycles. Since there are 81 applications in the corpus studied, this gives two expected values of 40.5. Observed values are shown in the table as *i* and *j*. We can then apply the χ^2 test to this table.

number of applications with X:	observed	expected
over-represented in cycles	i	40.5
not over-represented in cycles	j	40.5
total	81	81

Table 5. Cycle participation for classes statically accessed in cycles across all applications in the corpus.

9.4 Results

9.4.1 Application-level Results

The application-level results are shown in Table 7. The “size” column gives the size of the application in terms of number of top-level classes defined in its source files; the “in cycle” column shows the number of classes that participate in cycles; the “access%” column shows the percentage of the application’s classes that are accessed statically; the “static”, “large”, “large+static” and “small+static” columns show the probability of the difference between the expected and observed being due to chance as computing from using the χ^2 test for each of the application-level hypotheses described in Section 3. We marked up entries in the four rightmost columns with “*”, “**” if the data supported our hypothesis at the 0.05 and 0.01 levels, respectively; and with “ †” if we could reject the null hypothesis, but the direction of inequality went against our hypotheses. Blank entries indicated that we could not apply the χ^2 test because the expected values contained an entry less than 5.

For the hypothesis that statically accessed classes are more likely to be involved in cycles: 23 of the applications supported the it 0.01 level, 3 supported it at the 0.05 level, and for 1 application we could reject the null hypothesis at the 0.05 level but the direction of inequality went against our hypothesis. For the hypothesis that large classes are more likely to be involved in cycles: 34 of the applications supported it at the 0.01 level, 3 supported at the 0.05 level, none went against it at the 0.05 or 0.01 levels. For the hypothesis that small, statically accessed classes are more likely to be involved in cycles than small classes: 9 supported it at the 0.01 level, 1 went against it at the 0.05 level For the hypothesis that large, statically accessed classes are more likely to be involved in cycles than large classes: 17 supported it at the 0.01 level, 3 supported at the 0.05 level, 1 went against it at the 0.05 level.

The applications that had no significant results for any of the hypotheses, either because the χ^2 test could not be applied, or because the results were not significant at the either at the 0.05 level or 0.01 level are shown in Table 6. Most of the applications in this table are small in size (total number of classes) so it is unsurprising that their results were insignificant or that the χ^2 test could not be applied to them.

application	size	in cycle	access%
antlr-2.7.6	216	63	8%
axion-1.0-M2	237	48	5%
drawn-vC	17	2	0%
drawswf-1.2.9	183	69	12%
fitjava-1.1	37	2	19%
galleon-1.8.0	243	41	22%
ganttproject-1.11.1	310	153	9%
informa-0.6.5	151	9	13%
jaga-1.0.b	100	17	3%
james-2.2.0	259	8	5%
javacc-3.2	125	64	26%
jchempaint-2.0.12	612	18	10%
jeppers-20050607	20	0	5%
jfreechart-1.0.0-rc1	469	59	9%
jgraph-5.7.4.3	50	39	22%
jhotdraw-6.0.1	300	59	8%
joggplayer-1.1.4s	114	15	13%
jparse-0.96	69	66	6%
jrat-0.6	237	29	13%
jrefactory-2.9.19	211	159	6%
jspwiki-2.2.33	228	47	8%
jung-1.7.1	454	48	9%
junit-4.1	66	25	12%
oscache-2.3-full	63	10	10%
pmd-3.3	375	167	5%
poi-2.5.1	480	82	11%
proguard-3.6	363	108	15%
quartz-1.5.2	142	18	7%
quickserver-1.4.7	123	45	19%
sablecc-3.1	199	94	9%
sandmark-3.4	837	172	14%
spring-1.2.7	1158	31	8%
struts-1.2.9	286	41	7%
trove-1.1b5	262	96	1%

Table 6. Applications without any significant results

application	size	in cycle	access%	static	large	small+static	large+static
aglets-2.0.2	280	99	23%	0.000**	0.000**		0.121
ant-1.6.5	700	228	9%	0.059	0.577	0.919	0.019*
aoi-2.2	346	188	11%	0.332	0.001**	0.761	0.522
argouml-0.20	1402	780	13%	0.000**	0.001**	0.000**	0.940
aspectj-1.0.6	2116	1391	21%	0.000**	0.000**	0.162	0.000**
azureus-2.3.0.4	1650	1009	17%	0.000**	0.000**	0.000**	0.000**
bluej-2.1.0	396	204	23%	0.000**	0.003**	0.000**	0.067
colt-1.2.0	269	89	16%	0.942	0.000**		0.827
columba-1.0	1180	166	13%	0.923	0.000**	0.439	0.981
compiere-251e	1372	368	14%	0.000**	0.451	0.000**	0.000**
derby-10.1.1.0	1386	729	12%	0.471	0.000**	1.000	0.563
drjava-20050814	665	436	10%	0.943	0.000**	0.996	0.793
eclipse_SDK-3.1.2	11415	5000	16%	0.000**	0.000**	0.267	0.000**
findbugs-1.0.0-rc1	758	223	15%	0.000**	0.000**	0.997	0.000**
fitlibrary-20050923	124	15	15%		0.008**		
freecs-1.2.20060130	121	105	60%	0.004**	0.746		
geronimo-1.0-M5	1719	120	13%	0.000**	0.282	0.389	0.000**
glassfish-9.0-b15	582	148	11%	0.035†	0.000**		0.030†
gt2-2.2-rc3	1915	416	15%	0.000**	0.000**	0.000**	0.000**
heritrix-1.8.0	282	63	11%	0.036*	0.000**		0.173
hibernate-3.1-rc2	902	674	11%	0.932	0.020*	0.205	0.648
hsqldb-1.8.0.4	217	116	28%	0.523	0.000**	1.000	0.508
htmlunit-1.8	222	104	8%	0.897	0.001**		
ireport-0.5.2	347	193	7%	0.948	0.022*	0.473	0.939
itext-1.4	492	268	19%	0.000**	0.001**	0.652	0.000**
jag-5.0.1	121	31	15%		0.000**		0.681
jasperreports-1.1.0	633	283	16%	0.003**	0.000**	0.487	0.546
jboss-4.0.3-SP1	4143	1081	13%	0.116	0.000**	0.472	0.026*
jedit-4.2	234	153	21%	0.082	0.001**	0.758	0.317
jetty-5.1.8	327	82	13%	0.275	0.000**		0.925
jext-5.0	211	143	14%	0.020*	0.227		0.291
jmeter-2.1.1	560	125	10%	0.001**	0.000**		0.000**
jre-1.5.0.06	9820	4610	17%	0.000**	0.000**	0.001**	0.000**
jtopen-4.9	2857	907	8%	0.000**	0.000**	0.837	0.000**
log4j-1.2.13	171	46	16%	0.001**	0.658		0.000**
lucene-1.4.3	170	56	12%	0.785	0.019*		
luxor-1.0-b9	257	163	30%	0.000**	0.094	0.000**	0.008**
megamek-2005.10.11	455	152	25%	0.382	0.000**	0.069	0.007**
mvnforum-1.0-ga	219	67	26%	0.006**	0.000**		0.010*
netbeans-5.5-beta	10639	3986	19%	0.000**	0.000**	0.528	0.000**
openoffice-2.0.0	2925	227	6%	0.000**	0.000**		0.003**
rssowl-1.2	189	144	29%	0.179	0.000**		
scala-1.4.0.3	399	94	16%	0.000**	0.927	0.000**	0.005**
sequoiaerp-0.8.2-RC1	936	360	22%	0.029*	0.708	0.565	0.131
soot-2.2.3	1924	1513	11%	0.000**	0.000**	0.000**	0.061
squirrel_sql-2.4	882	405	8%	0.312	0.998	0.043†	0.956
tomcat-5.5.17	937	249	12%	0.929	0.000**	0.973	0.997

Table 7. Applications with at least one significant result.

9.4.2 Corpus-level Results

Table 8 shows the number of applications in the corpus with classes that are accessed statically over-represented in cycles, regardless of whether the over-representation was significant or not at the application level as determined by the χ^2 test. The probability yielded by applying the χ^2 test to this corpus-level data is 1.77×10^{-6}. This means at the 0.01 level we can reject the null hypothesis, and looking at the direction of the inequality we see it supports our hypothesis, that at the corpus-level, a randomly selected application's classes that are accessed statically seem to be more likely to be involved in cycles than those not accessed statically.

#applications w/ static access	observed	expected
over-represented in cycles	62	40.5
not over-represented in cycles	19	40.5
total	81	81

Table 8. Cycle participation for classes statically accessed in cycles across all applications in the corpus.

Table 9 shows the number of applications in the corpus with classes that are large and over-represented in cycles. The probability yielded by applying the χ^2 test to this data is 5.54×10^{-11}. This means at the 0.01 level we can reject the null hypothesis, and looking at the direction of the inequality we see it supports our hypothesis, that at the corpus-level, a randomly selected application's classes that are large seem to be more likely to be involved in cycles than those that are small.

#applications w/ large classes	observed	expected
over-represented in cycles	70	40.5
not over-represented in cycles	11	40.5
total	81	81

Table 9. Cycle participation for large classes in cycles across all applications in the corpus.

Table 10 shows the number of applications in the corpus with classes that are both large and statically accessed overrepresented in cycles. The probability yielded by applying the χ^2 test to this data is 1.47×10^{-5}. This means at the 0.01 level we can reject the null hypothesis, and looking at the direction of the inequality we see it supports our hypothesis that, at the corpus-level, a randomly selected application's classes that are both large and statically accessed seem to be more likely to be involved in cycles than those that are just large.

#applications w/ static access	observed	expected
over-represented in cycles	60	40.5
not over-represented in cycles	21	40.5
total	81	81

Table 10. Cycle participation for only large classes statically accessed in cycles across all applications in the corpus.

Table 11 shows the number of applications in the corpus with classes that are both small and statically accessed overrepresented in cycles. The probability yielded by applying the χ^2 test to this data is 0.74. This means at we cannot reject the null hypothesis.

#applications w/ static access	observed	expected
over-represented in cycles	39	40.5
not over-represented in cycles	42	40.5
total	81	81

Table 11. Cycle participation for only small classes statically accessed in cycles across all applications in the corpus.

9.4.3 Edges Results

As shown in the stacked bar graphs of Figure 4, most of the applications showed an increase in the number of classes participating in cycles from when edges due to the USES-IN-THE-INTERFACE to when edges due to both the USES-IN-THE-INTERFACE and access of static members were considered. Applications that did not show a change were Antlr, Drawn(*), DrawSWF, FitJava, Informa, Jaga(*), James(*), Jeppers(*), JFreeChart, JHotDraw, JParse, JRefactory, JUnit, OSCache(*), SableCC, Trove(*). Those marked with (*) also indicate there was no difference in cycle participation when going considering the access of static members in addition to USES-IN-THE-INTERFACE and `new`. What this means is that with respect to cycle participation, edges due to static access contribute in some way to cycle participation in all but 6 small applications from the corpus. This is strong support for H2.

Also worth noting is the difference in heights between the bars marked "I+T+N" and "A". The difference in heights can be explained by the use of casts as illustrated earlier in Figure 1. In SableCC, for instance, there is a big difference in heights between the "I+T+N" and "A" bars. We have looked at the relevant code and found that this is indeed due to an unusual implementation of the visitor pattern that makes extensive use of type casts.

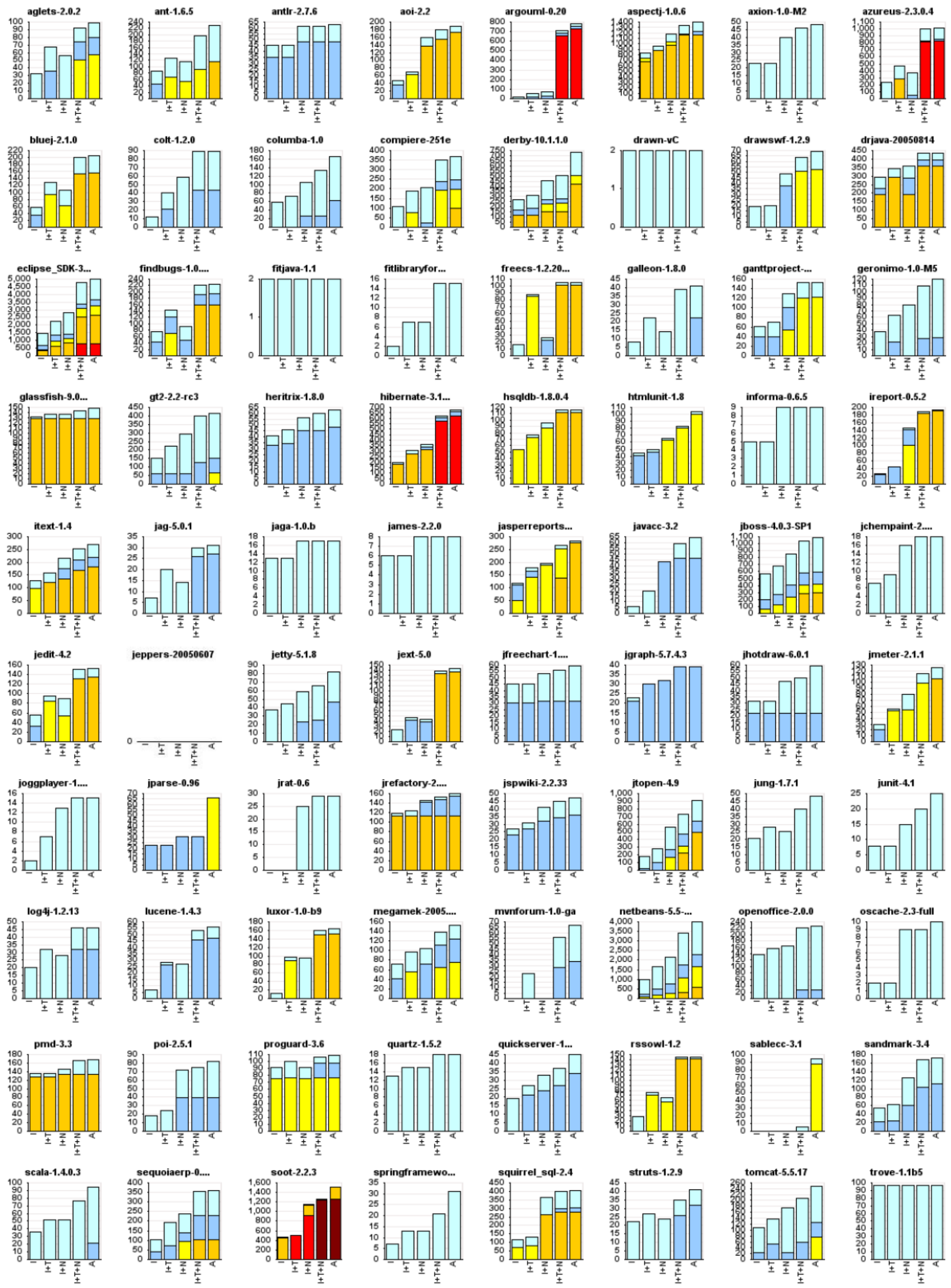


Figure 4. Participation of edges due to different forms of dependency in cycles.

9.5 Discussion and Conclusions

Both the application- and corpus-level results generally seem to support the contention that classes that are accessed statically are more likely to be involved in cycles than those that are not. For the four hypotheses we tested using the χ^2 test we obtained only 3 statistically significant negative results. For the hypothesis pertaining to edges due to access of static members appearing in cycles, only 6 applications of the 81 examined had a negative result.

One interesting finding from our attempt to control for size, was that, at the corpus-level, small classes seemed to have about the same chance of appearing in cycles as small classes that were accessed statically. Similarly, at the application-level, far fewer applications had statistically significant results that supported this hypothesis when only small classes were considered, than when only large classes were considered.

Another interesting thing we were able to do in this paper was to be able to include applications in an analysis that did not, on their own, have statistically significant results with the χ^2 test. We were able to look at the direction of inequality between the expected and observed populations, and incorporate that into a corpus-level analysis. Many empirical studies of code do not do this level of analysis, because they do not examine a large enough corpus of software for the results at this level to be significant. Rather they concentrate only on application-level analysis.

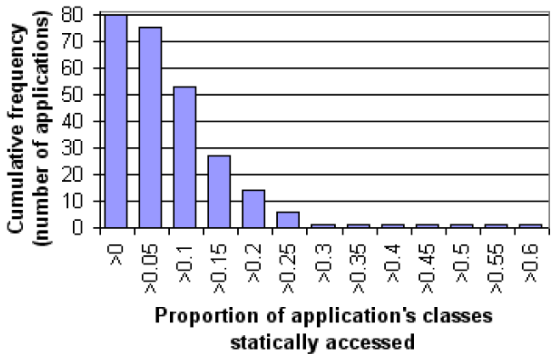


Figure 5. Distribution of proportion of an application's classes that are accessed statically across the corpus.

In terms of the proportion of classes accessed statically, Figure 5 shows a cumulative frequency distribution of this for all the applications in the corpus. 80 of the 81 applications in the corpus have >0% of their classes accessed statically, about 52 have >10% of their classes accessed statically, and so on as shown in this figure. We found this distribution surprising, given that the use of the static modifier is considered to be "bad" by many people in the object-oriented programming community. Kerievsky surveys some ways in which the static modifier can be eliminated from a design in his discussion of “Singletonitis”[Ker04]. The distribution quantifies the extent to which non-private static members are used in real designs.

In terms of future work, we’d like to know why the effect of statics on cycle participation only appears to be stronger for large classes than smaller ones. We’d also like to see if over time static members that are not involved tend to become involved in cycles. This could be done via a controlled experiment where programmers are asked to modify a design with many static members and one without, or by doing longitudinal program analysis on many subsequent releases of a program.

Coauthor Declaration Chapter 10 [YTM08]

AUTHORSHIP STATEMENT

1. Details of publication and executive author

Title of Publication		Publication details
Yang, Hong Yul, Ewan Tempero, and Hayden Melton. **An empirical study into use of dependency injection in Java.** *19th Australian Conference on Software Engineering (aswec 2008).* IEEE, 2008.		
Name of executive author	School/Institute/Division if based at Deakin; Organisation and address if non-Deakin	Email or phone
Hayden Melton	**School of Information Technology, Deakin University**	**hmelton@deakin.edu.au**

2. Inclusion of publication in a thesis

Is it intended to include this publication in a higher degree by research (HDR) thesis?	Yes / ~~No~~	If Yes, please complete Section 3 If No, go straight to Section 4.

3. HDR thesis author's declaration

Name of HDR thesis author if different from above. (If the same, write "as above")	School/Institute/Division if based at Deakin	Thesis title
As above	**School of Information Technology, Deakin University**	Empirical Studies of Structural Phenomena Using a Curated Corpus of Java Code
If there are multiple authors, give a full description of HDR thesis author's contribution to the publication (for example, how much did you contribute to the conception of the project, the design of methodology or experimental protocol, data collection, analysis, drafting the manuscript, revising it critically for important intellectual content, etc.)		
This work was conducted collaboratively by myself and Hong Yul Yang during our respective times as a PhD students in Computer Science at the University of Auckland, both sharing a PhD supervisor (Ewan Tempero). I wrote approximately half the paper, and the idea for the paper was mine. In particular, I was involved in the background, introduction, motivation and methodology insofar as identifying the specific concrete examples of dependency injection that were detected, and of course in the conception and development of the Qualitas Corpus a subset of which was analysed in this paper.		
I declare that the above is an accurate description of my contribution to this paper, and the contributions of other authors are as described below.	Signature and date	[illegible] 18th Dec 2016

4. Description of all author contributions

Name and affiliation of author	Contribution(s) (for example, conception of the project, design of methodology or experimental protocol, data collection, analysis, drafting the manuscript, revising it critically for important intellectual content, etc.)
A/Prof Ewan Tempero Dept of Computer Science University of Auckland New Zealand	Performed general tasks associated with supervising PhD students (Hayden Melton and Hong Yul Yang), including: identification of venue to submit paper to; helping with editing and typesetting of paper pre- and post-submission; critical review of early drafts of paper; feedback to student on general direction of research in weekly meetings.
Dr Hong Yul Yang (Formerly) Dept of Computer Science University of Auckland New Zealand	Used tool developed as part of own PhD research to detect the specific forms of dependency injection noted in paper, collated results, wrote approximately half the paper (results, conclusions, discussion).

5. Author Declarations

I agree to be named as one of the authors of this work, and confirm:

i. that I have met the authorship criteria set out in the Deakin University Research Conduct Policy,

ii. that there are no other authors according to these criteria,

iii. that the description in Section 4 of my contribution(s) to this publication is accurate,

iv. that the data on which these findings are based are stored as set out in Section 7 below.

If this work is to form part of an HDR thesis as described in Sections 2 and 3, I further

v. consent to the incorporation of the publication into the candidate's HDR thesis submitted to Deakin University and, if the higher degree is awarded, the subsequent publication of the thesis by the university (subject to relevant Copyright provisions).

Name of author	Signature*	Date
Ewan Tempero	[illegible]	16/12/2016
Hong Yul Yang	[illegible]	2/12/2016

6. Other contributor declarations

I agree to be named as a non-author contributor to this work.

Name and affiliation of contributor	Contribution	Signature* and date

* If an author or contributor is unavailable or otherwise unable to sign the statement of authorship, the Head of Academic Unit may sign on their behalf, noting the reason for their unavailability, provided there is no evidence to suggest that the person would object to being named as author

7. Data storage

The original data for this project are stored in the following locations. (The locations must be within an appropriate institutional setting. If the executive author is a Deakin staff member and data are stored outside Deakin University, permission for this must be given by the Head of Academic Unit within which the executive author is based.)

Data format	Storage Location	Date lodged	Name of custodian if other than the executive author
Qualitas Corpus of Java code, both binaries and source code with metadata (see http://qualitascorpus.com/)	University of Auckland, New Zealand	Starting 2006, onward	Ewan Tempero

This form must be retained by the executive author, within the school or institute in which they are based.

If the publication is to be included as part of an HDR thesis, a copy of this form must be included in the thesis with the publication.

Chapter 10 An Empirical Study into Use of Dependency Injection in Java

Over the years many guidelines have been offered as to how to achieve good quality designs. We would like to be able to determine to what degree these guidelines actually help. To do that, we need to be able to determine when the guidelines have been followed. This is often difficult as the guidelines are often presented as heuristics or otherwise not completely specified. Nevertheless, we believe it is important to gather quantitative data on the effectiveness of design guidelines wherever possible.

In this paper, we examine the use of "Dependency Injection", which is a design principle that is claimed to increase software design quality attributes such as extensibility, modifiability, testability, and reusability. We develop operational definitions for it and analysis techniques for detecting its use. We demonstrate these techniques by applying them to 34 open source Java applications.

10.1 Introduction

Design principles [Rie96][GHJV95][Mar96a][Mar96b][SMC74] influence the internal structure of a software system. Particularly, they guide the decisions we make as developers about the organization of the entities in a system's source code. These decisions are inherent to the activity of programming— for instance, in adding some particular functionality to a system should we write the code as a new method, generalise an existing method, create a whole new class, or some combination of the above? Design principles help us to choose the "best" option.

Design principles are important because we believe that the internal structure of a system, as reflected in its source code, affects its maintainability, understandability, testability, modifiability, performance and so on, that is, its *software quality attributes* [Par72][SMC74][BJ95]. Thus the "best" decision we can make in organising source code entities (i.e., methods, classes, packages etc) is the one that most improves the attributes of software quality that are important for a particular system. In order to determine which is "best", we need to be able to quantify the benefit due to the application of any given design principle. We need to understand what the trade-offs are and how different design principles interact.

We can determine the benefit achieved by applying a design principle by applying it and measuring the change to all the quality attributes. Measuring quality attributes is difficult enough but by itself does not tell us what the benefit is if we cannot be sure that the design principle has been applied correctly (or at all). Without reliable and objective means to determine when a design principle has been applied, we cannot be sure what caused the effects on quality attributes we observe.

A difficulty in reliably and objectively determining the use of most design principles is that they usually are not expressed in an *operational* manner. We believe that developing operational definitions of design principles is a necessary step in empirical validating their use. In this paper, we look at developing an operational definition for the the design principle sometimes known as *Dependency Inversion Principle (DIP)* [Mar96a] and carry out an empirical study of its use.

We think the DIP is worthy of further study because its proponents argue its application leads to systems that are more extensible [Mar96a][JF88][SGN04], testable [Mar96a][MFC01][TH02][Lak96, p.388],modifiable [Mar96a] [Lak96, p.330] and reusable [Mar96a][JF88]. In the work described in this paper we discuss a specific structural form of the DIP—what Fowler terms *Dependency Injection* (DI) [Fow04]. We have developed an operation definition for DI, developed a tool that measures the use of DI according to our definition, and have applied the tool to 34 open source Java applications.

The rest of the paper is organised as follows. In section 2, we summarise the arguments for using DI, in particular the anticipated benefits having classes designed by applying the DI principle. From this, in section 3, we determine the structural characteristics of code that result from such an application, which leads to the definitions of four structural forms representing possible DI use. From this we develop our analysis techniques. In section 4 we present the results of our study and discuss our interpretation of these results in section 5. Section 6 then presents our conclusions.

10.2 Background

The phrase *Dependency Inversion Principle* was first coined by Martin in 1996 [Mar96a] although the concept it represents has been discussed by many others under the guise of different names. Fowler [Fow05] dates the concept back to Johnson and Foote's discussion of *Inversion of Control (IOC)* [JF88]in 1988, and he notes that Sweet also alludes to it in 1985 with the more "colourful" phrase the *Hollywood's Law* [Swe85]. Lakos [Lak96, ch.6] also discusses the DIP under the guise of *insulation*.

While the DIP is easily stated at a conceptual level, defining it concretely, in terms of entities in Java source code, is less straightforward. A conceptual statement of the DIP is that by Martin: "High-level modules should not depend upon low-level modules. Both should depend upon abstractions" [Mar96a]. If we glean the examples given by Martin we might take this to mean that in Java a class should depend on interface or abstract types, not concrete types, although there are some benefits that accrue even with concrete types.

Besides the issue of whether we should depend on interface, abstract or concrete types we must deal with the "problem of instantiation" [MT07a], or as the Gang of Four state "you have to instantiate concrete classes (that is, specify a particular implementation) somewhere in your system" [GHJV95, p.18]. This is another challenge in concretely stating, and measuring the DIP — we need to know the mechanism by which concrete classes are instantiated and passed in to their DIP exhibiting clients.

There are actually many ways to instantiate and pass in concrete classes to those exhibiting the DIP. Fowler identifies the Dependency Injection and Service Locator approach [Fow04]. In this work, we concentrate on the *Dependency Injection* form of the DIP. In the *Dependency Injection* (DI) form of the DIP, as it is discussed by Fowler [Fow04], the object assigned to the field of a class is passed in through one of that class' constructors or methods. A simple illustration of dependency injection is as follows:

```
class A {
  B b;
  public A(B b) {
    this.b = b;
  }
  //...
}
```

In the above code example A is exhibiting dependency injection because the object that gets assigned to its field ‘b’ is passed in as a parameter in A’s constructor. We will, for the moment, avoid a discussion of whether B should be an interface type, abstract type or concrete type. The key observation is that A does not depend on a particular implementation of B. Particularly, when clients instantiate A, they get to specify the particular subtype of B to be assigned to ‘b’ at runtime. This can have beneficial consequences to several software design quality attributes.

10.2.1 Effects on Quality

It has been argued that Dependency Injection affects many quality attributes, in particular extensibility, testability, and reusability.

Extensibility can be defined as “the ease with which a system or component can be modified to increase its storage or functional capacity”[IEE90]. In the above snippet A is arguably more extensible because it can be used with different implementations of B without modifying the source code of A. Indeed this is why DI is used at the “plug-points” of application frameworks [JF88][SGN04].

Testability can be defined as “the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met”[IEE90]. Dependency Injection supports the use of *mock objects* to help unit test a class [MFC01][TH02]. Mock objects can be used to both provide control and observe the class under test.

Reusability can be defined as “the degree to which a software module or other work product can be used in more than one computer program or software system”[IEE90]. Dependency injection can improve reuse by (1) improving flexibility and (2) breaking transitive dependencies. DI can improve flexibility because different implementations can be used with the class we want to reuse, improving the degree to which that class can be used in multiple situations, as discussed above in extensibility. Dependency injection can reduce the number of classes we have to deploy in the context of a new system by breaking transitive dependencies. This is important because to effectively reuse a class it should not be tied to a large block of unnecessary code [Lak96, p.14]. If a class we reuse depends on a class type, from the perspective of reuse, it also transitively depends on any types that appear in the private part of that class type. If the type it depends on is an interface type then it does not transitively depend on any private parts of that interface's implementation.

10.3 Characterising Dependency Injection

10.3.1 Definitions

In the Dependency Injection form of the DIP the value assigned to a class' field is passed in through a setter or constructor, rather than created within the class. We can use this as the basis for an operational definition for DI. For each field in a class, we determine what values are assigned to the field and where those values came from. If they do not come from outside the class, then that is inconsistent with the intent of DI (although we identify one special case below). We have identified 4 forms of DI for fields in Java code, which we define below.

```
class CNDeg {               class MNDeg {
 B b;                        B b;
 public CNDeg(B b){          public void setB(B b){
   this.b = b;                 this.b = b;
 }                           }
}                           }

class CWDeg {               class MWDeg {
 B b;                        B b;
 public CWDeg() {            public MWDeg() {
   b = new BImpl();            b = new BImpl();
 }                           }
 public CWDeg(B b){          public void setB(B b){
   this.b = b;                 this.b = b;
 }                           }
}                           }
```

Figure 1. Examples of the forms of DI

10.3.1.1 Constructor No Default (CND)

The *only* object a field in a class can be assigned comes through the parameter of the class' constructors. That is, the only objects a field is assigned are passed in from *outside* the class, through the class' constructor(s). Class `CNDeg` in Figure 1 shows an example.

Rationale: In the above code the *only* way an object can be assigned to field 'b' is by passing that object through the constructor. This means `CNDeg` can be tested with a mock object of supertype `B`. Similarly, `CNDeg` is potentially more extensible because it can be used with different implementations of `B`. If `B` is an interface type it means that `CNDeg` can be reused in another system independently from any implementations of `B`.

10.3.1.2 Method No Default (MND)

The object a field in a class can be assigned comes through the parameter of one of either the constructor, or the class' non-private methods. That is, the only objects a field is assigned are passed in from *outside* the class, through the class' non-private methods(s). Class `MNDeg` in Figure 1 shows an example.

Rationale: The above code is similar to that given for CND, except the object 'b' is assigned is passed in through a setter method. The use of a setter method allows the object assigned to the field to change over the object of the lifetime purporting improved flexibility over CND. On the other hand it is also possible we forget to assign an object to field 'b' (not possible in CND), and this will likely cause a `NullPointerException` at runtime. Which is better is subject to some debate. Beck recommends the constructor based approach[Bec97], saying it is immediately clear what a class requires when it is instantiated, and furthermore it is impossible to instantiate the class without passing in the field's objects. However, a recent empirical study by Stylos and Clarke seems to contradict Beck's argument. Their study found that that programmers found it easier to pass references through setter methods rather than constructors [SC07]. Consequently we have chosen to measure both forms Apart from this the reusability, testability and extensibility are the same as CND.

10.3.1.3 ConstructorWith Default (CWD)

The object assigned to a field *can* be passed in through a constructor but this does not happen exclusively. The field is also assigned a "default" object from within the class. Class `CWDeg` in Figure 1 shows an example.

Rationale: The above code is similar to that given for CND, except there is a *default* implementation of `B` referenced in it. This potentially hinders reusability because we must now deploy `B` *and* `BImpl` in order to compile `CWDeg` in the context of a new system. We must also deploy anything `BImpl` depends on—we could end up copying a very large chunk of code in this transitive fashion. That said, CWD has no significant difference in flexibility, extensibility and testability than in CND, and furthermore users of such classes do not have the burden of having to provide an implementation for `B` for every use of `CWDeg`.

10.3.1.4 Method With Default (MWD)

The object assigned to a field *can* be passed in through a constructor or non-private method but this does not happen exclusively. The field is also assigned a "default" object from within the class. Class `MWDeg` in Figure 1 shows an example.

Rationale: This situation is analogous to CWD—reuse is inhibited because a concrete type (`BImpl`) is referred to in the body of `MWDeg`.

10.3.1.5 Completeness

There are a number of ideas that have been labelled "dependency injection" or something similar (as we will mention further below). As this is the first study of its kind, we have chosen not to try to capture all possible variations. Instead, we have limited our study to these relatively simple forms of DI. We believe these forms are representative of the presentations of DI in the literature, in particular, in the trade press and tutorials likely to be accessible to developers. As such, we believe that if there is widespread adoption of DI, then the forms we have identified should be prevalent.

10.3.2 Practical Considerations

The definitions above give the general structures that indicate the use of DI. There are, however, some consequences and practical issues that require further discussion. In this study, we require that types of fields be nonconcrete (that is, either interfaces or abstract classes). This means the use of any concrete type for a field rules out that class as using DI. As discussed earlier, from the point of view of, for example, testing, such fields might be considered an acceptable form of DI and so we intend to look at such forms in future work.

The creation of concrete values (that is, calls to a constructor) also rule out the class as using DI provided the creation occurs outside the class' constructor (since creation of concrete values within a constructor could indicate one of the "default" cases). It is also possible that concrete values can be assigned to fields, even though they are not created in the class. For example, a parameter of concrete type can be assigned to the field. Such definitions rule out the class from using DI.

The use of constructors of arrays depends on the base types of the arrays — if the result requires the use of concrete values then it rules out the use of DI, otherwise it is neutral.

A value assigned as a result of a method invocation on another class also rules out the use of DI, as in the general case we cannot be sure what the type of that value will be. The use of a service locator is a special case that we believe can be identified, and we will consider this in future work.

One last form of field definition that we must consider is the assignment of `null` to a field. This form of defining a value for a field does not impact use of DI and so is ignored.

We also ignore fields that are of primitive type, or of type from the Java Standard API ("built-ins"), in that their presence did not impact our classification of a class. We ignore fields of primitive types since there is no opportunity for a developer to allow alternative implementations to be provided for such fields. It could be argued that the object wrapper types, such as `java.lang.Boolean`
could have been used instead, however this choice has other issues, being both final classes and types supplied by the Standard API. In the case of types from the Standard API, there is the opportunity to use appropriate interfaces (e.g., `java.util.List`) or abstract classes (e.g., `java.io.Reader`). However there are also classes for which there is no convenient interface or abstract parent (e.g., `java.lang.String`) meaning, again, the developer has no alternative. This is also something we wish to consider further in future work, that is, whether these built-in types support are being used to provide DI.

Final types, that is types that cannot have subtypes (classes declared "final" in Java), cannot be used for DI, and so their presence disqualifies a class from using DI.We also note that, should we consider classes with fields of concrete types to be acceptable for using DI, final types would still be a problem. If we consider built-in types further, final types such as `String` might have to be treated specially.

10.3.3 Measurement

10.3.3.1 Analysis Procedure and Algorithm

The analysis of dependency injection in application code is performed by extending a part of our existing tool [YTB05], which operates on Jimple – a static single assignment typed 3-addressed intermediate representation of Java bytecode from the Soot framework [VRCG+99]. It also utilises static analysis features of the Indus project [Ind], which is based on Soot. The tool is limited to Java 1.4 source code.

The overall measurement actually comprises several steps. The first step is to analyse the source (represented in Jimple) for "use-def" information and generate a graph data-structure comprising the usage/definition sites and data flows among them. The algorithm for computing this employs standard inter-procedural data flow analysis techniques such as those found in the slicing literature (e.g. [AH03]), and in particular the data- and object-flow analyses that Indus provides, as described in [Ran02]. The precision of the analysis is thus dependent on the underlying techniques, which deal with well-known issues such as polymorphism and array aliasing to a certain extent.

The aim of the analysis for each field of a class is to determine all of its possible *definition* values. In the simplest case, the definition of a field is a direct assignment, e.g. `field := value`. But it is usually the case that the value is in turn defined through a preceding statement, and so on, which effectively results in a chain of definitions, thus referred to as *use-def chains*, that eventually affect the value of the original field.

```
public class A {
  protected B b;
  public A(B ba) {
    setB(ba);
  }
  public A() {
    setB(getDefault());
  }
  protected void setB(B b) {
    this.b = b;
  }
  protected B getDefault() {
    return new BImpl();
  }
}
```

Figure 2. Non-trivial assignment examples

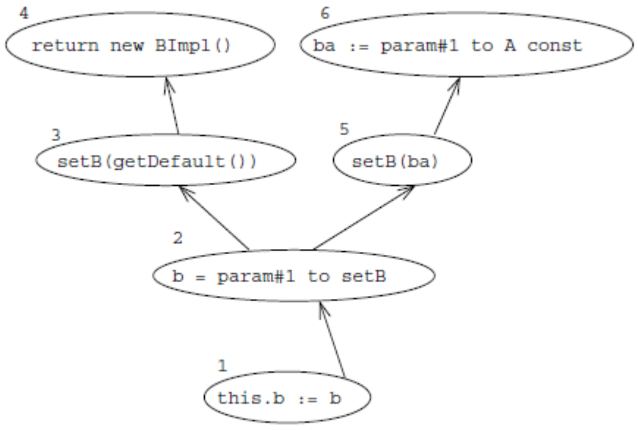


Figure 3. data-flow graph

Whereas in the examples we have shown so far, the definitions of fields have been of a simple nature (direct assignments), there are many instances where the value is defined through indirect means, as illustrated in figure 2, thereby requiring an analysis along use-def chains. Here, field `b` is defined through a parameter to the method `setB`, which is called locally by both of `A`'s constructors. The first constructor simply passes down its parameter `ba` to `setB`. The second constructor on the other hand additionally calls `getDefault`, which constructs and returns a concrete value that is then passed to `setB`.

To obtain the definitions of each field, the tool applies a depth-first traversal algorithm on a graph representing statements and data flows between them. This graph, for a given class `C`, is constructed such that:

•its vertices represent statements within the *"boundary"* of `C` (we define this as any program element contained in `C` or its superclasses)
•edge exists from v1 to v2 iff both data and control flow exist from a value used in v2 to value used in v1, i.e. the direction of the edge is opposite to that of data/control flow.

The algorithm begins traversing from the statements (vertices) that directly assign a value to any field of `C`, then follows the edges until either all vertices have been visited or no more vertices can be visited. During the traversal, the algorithm records each parameter value or concrete value it encounters. Figure 3 demonstrates this process being applied class `A` in figure 2 – note that parameter passing are shown as implicit statements to clarify the data flow. In this example, the algorithm begins from the source vertex labelled `1` (the assignment statement) and traces along the edges to eventually reach vertices numbers `4` (concrete value) and `6` (parameter to `A`'s constructor). These vertices represent the definition sites of `b` that are relevant to DI and hence are recorded.

For each definition site, the following are recorded:
• the name and type of the field that is being defined
• the class that it belongs to
• location (class, method and line number) of the definition
• the type of the value defining the field
• The nature of definition: 'is the value from a parameter? Or a concrete value? Or is it through some other means?'

The above details are aggregated for each class and used to determine the class's conformance to the DI definitions outlined previously.

10.3.3.2 Scope of Analysis

It is worth noting that we are deliberately limiting the use-def analysis to within the boundary of each given class. This effectively reduces the search space for the analysis, thereby improving performance. Also in the interest of the forms of dependency injection under investigation, analysing within classes is sufficient in obtaining the necessary results. However in the future we could extend the tool by tracking the use-def chain further to outside the boundary to cater for more system-wide forms of DIP such as service locators.

A consequence of our decision is that many of the issues that face other forms of data flow analysis, such as inheritance, polymorphism, aliasing, and the like do not apply to our analysis. For example, if a subclass of `A` from figure 2 directly assigns to the field `b`, then in our analysis that assignment will be treated as if `b` were a field of the subclass (as indeed it is).

An issue with polymorphic calls in data flow analysis is not being sure which code can actually be executed. However, since we regard values assigned to fields as the result of *any* method invocation to rule out the use of DI, the fact that the method invocation might be polymorphic is irrelevant.

Object aliasing is when two or more references (pointers) refer to the same runtime instance. Aliasing can present a problem because it allows the state of one object to be changed from multiple syntactic locations. We are only concerned as to where any object that is assigned to a field originated, specifically inside or outside of the class boundary. The state of that object, or how that state might change, is therefore not relevant to that determination.

10.4 Results

We have analysed a subset of a Java Corpus we have compiled [BFN+06] [MT07b] [TAD+10] consisting of open-source java applications, looking for evidence of the use of DI. Of the 34 applications in our study, 17 could be classified as true applications, in that they are intended to be deployed as is, whereas the other 17 were designed with the intent that they be embedded within other applications. For some it is difficult to draw the line, as some frameworks come with ready-made useful applications as examples (e.g., JMeter) and some applications provide APIs to allow programmatic customisation (e.g., JFreeChart). Our reason for classifying applications this way was the hypothesis that we would see more use of DI in systems intended to be embedded, in particular, frameworks.

Application	Type	Size	N	CND	MND	CWD	MWD	P	B	Not	%
ant-1.4.1	App	178	138	4	0	0	0	10	67	57	7%
antlr-2.7.5	Emb	209	142	2	1	1	0	23	35	80	5%
aoi-2.2	App	346	297	5	3	0	0	30	34	225	3%
argouml-0.18.1	App	1210	613	18	3	1	0	16	296	279	7%
axion-1.0-M2	App	237	148	13	13	0	0	8	40	74	26%
azureus-2.0.4.0	App	346	199	16	12	0	1	21	65	84	26%
colt-1.2.0	Emb	195	104	9	0	6	0	42	11	36	29%
fitjava-1.1	App	37	21	0	0	0	0	3	7	11	0%
fitlibraryforfitnesse-20051216	Emb	156	70	0	0	0	0	6	38	26	0%
ganttproject-1.11.1	App	310	197	28	4	0	0	9	50	106	23%
hibernate-3.1	Emb	895	529	62	22	0	0	24	232	189	31%
hsqldb-1.8.0.2	App	218	165	3	1	0	0	21	34	106	4%
ireport-0.5.2	App	347	289	0	0	0	0	41	131	117	0%
jag-5.0.1	Emb	121	99	1	0	0	0	6	51	41	2%
jaga-1.0.b	Emb	100	55	1	2	0	2	12	21	17	23%
james-2.2.0	App	259	150	4	4	0	0	9	79	54	13%
jchempaint-2.0.12	App	83	50	0	0	0	0	1	22	27	0%
jeppers-20050607	App	84	48	1	0	0	0	7	18	22	4%
jext-5.0	App	211	108	1	0	0	0	1	43	63	2%
jfreechart-1.0.0-rc1	App	469	313	10	8	2	1	33	141	118	15%
jgraph-5.7.4.3	Emb	50	32	2	1	0	0	1	9	19	14%
jhotdraw-6.0.1	Emb	300	171	15	11	0	1	20	59	65	29%
jmeter-1.8.1	Emb	216	153	3	3	0	0	4	81	62	9%
jparse-0.96	Emb	69	61	1	0	0	0	1	9	50	2%
jung-1.7.1	Emb	378	236	24	9	2	0	28	86	87	29%
junit-3.8.1	Emb	47	28	4	0	0	0	5	13	6	40%
lucene-1.4.3	Emb	170	134	18	2	0	0	19	38	57	26%
megamek-2005.10.11	App	455	314	12	4	0	0	52	94	152	10%
picocontainer-1.3	Emb	82	52	5	2	11	0	3	17	14	56%
poi-2.5.1	Emb	385	284	1	1	0	0	121	58	103	2%
rssowl-1.2	App	189	135	0	0	0	0	3	50	82	0%
sablecc-3.1	Emb	198	112	1	1	0	0	4	21	85	2%
scala-1.4.0.3	App	399	62	1	0	0	0	27	9	25	4%
spring_framework-1.1.5	Emb	905	508	22	30	4	16	22	295	119	38%

Table 1. Number of classes meeting each DI definition

We classified each class in an application according to the definitions given in section 3 and determined the totals for each category for each application. The results are shown in Table 1. The application name includes the version number we analysed. The **Type** column shows our classification of applications into those intended to be embedded (`Emb`) versus those that can be deployed stand-alone (`App`). The next column shows the size of the application in terms of number of top-level classes. The third column shows the number of top-level classes that appear in the analysis.

Classes that were not analysed include classes with no fields (e.g., interfaces, classes with only static methods), or subclasses whose only fields are those inherited from ancestors and do not directly assign to them. In some cases, the difference is quite surprising (ArgoUML for example), and is worth further study.

The columns **CND**, **MND**, **CWD**, **MWD** show the number of classes in the application obeying the different forms of DI as discussed in section 3.1. The **Not** column gives the number of classes analysed that have some form of field assignment that means they cannot be using DI as we have defined it. The **P** column shows classes that contain only fields that are of primitive type and the **B** column shows classes that contain only fields that are of a type from the Standard API or a primitive type. We report these separately as we cannot classify them in the 4 DI categories, and, since we ignore the effect of fields of primitive and built-in types, we felt it was mis-leading to classify such classes as not using DI. The last column shows the number of classes meeting one of sets of the DI criteria as a proportion of those classes that are "eligible" to meet the criteria, that is, the proportion of **N**− **P** − **B**.

Of the 34 applications we analysed, 5 have no classes that meet our criteria and a further 5 had only 1 class (classified as CND in all cases). All applications that had any class meeting the criteria had classes classified as CND, only two applications (`jaga` and `spring framework`) had fewer CND classes than some other category, and for only two further applications (`jfreechart` and `picocontainer`) where there fewer (and only just) CND classes that all other DI categories put together. The application with the most classes satisfying at least one DI form was `hibernate` (84) with the next largest number being that for `spring framework` (72).

The application with the highest proportion of eligible classes meeting one of the sets of DI criteria is `picocontainer`, with 12 applications overall having almost one quarter of their classes meeting the criteria.

10.5 Discussion

In this section we attempt to develop conclusions based on our data. Our goal is to determine to what degree DI is being used. The main issue in doing this, however, is determining intent. The structures we measure, while they may be consistent with the use of DI, may also be developed without intending to use DI or indeed without the knowledge of DI. Determining intent from source code is difficult. The rationale for decisions, particularly design decision, cannot be divined from code, and often is not provided in what documentation there may be available. Since this is the first study of its kind, we have no baseline on which to make some comparisons, and so some of our statements are necessarily speculative.

The overall sense is that DI is not being widely applied. There is no obvious difference between `Emb` and `App` type applications. Of the applications that had a small number of classes appearing to use DI, several of those classes appeared to meet the requirements only by accident, that is, a
false positive.

An example of a likely false positive is `JagBlockViewer` in `jag`. This is the only class that meets any of our criteria in this application, and it is documented as being a test class. It seems unlikely that DI was intended to be used in this case. Another example is `Node` in `sablecc`, which is classified as MND. This is an abstract class whose sole field is of its own type, that is, `Node`, which suggests it is unlikely to have been designed with DI in mind. This application also has a single class classified as CND, namely `ParserException`, an exception class.

For other applications with small numbers, it is more difficult to rule out the use of DI, but we think it unlikely that anyone familiar with DI would deliberately use it so few times. For example `ant` has four classes classified as CND. They are all in the same package (`org.apache.tools.ant.taskdefs`), a package that has more than 70 classes, including more than 25 that are eligible to meeting DI criteria but which do not. While it is conceivable that no other classes meet ID criteria due to the nature of the application, we think it is unlikely, and so suspect that the four classes that do, so do accidentally.

One possible explanation for small numbers is that the relevant classes were not developed with DI in mind, but in order to support a design pattern. Many common implementations of design patterns, especially the original set [GHJV95], use DI-like structures to achieve their goal. We speculate that someone not familiar with DI but implementing a design pattern would thus create classes that meet our definitions.

An example of this is JHotDraw. JHotDraw has a number of classes (about 26%) meeting the criteria, but looking at the classes we find many instances of classes with names involving "Command", "Handle", "Visitor", "Listener", and "Enumerator". These names suggest these classes were designed not so much with DI in mind, but a consequence of the respective design patterns.

In fact, given JHotDraw's history, DI was almost certainly intended, but it does illustrate the fact that since some design patterns implementations do mimic the DI patterns we study, it is conceivable that developers create classes without being aware of DI. This raises the issue of whether design patterns should be taught without reference to the underlying principles that make the patterns effective.

Determining whether the numbers we see are indicative of high or low use of DI is difficult without studying each and every class. The application `junit` provides a useful case study, being small enough for it to be feasible to do exactly that. On the surface, 4 of 28 classes seems a small number of classes to be using DI, and given Junit's development history we might expect to see DI used extensively. In fact, one class (`TestDecorator`) appears the consequence of a design pattern, and one class (`FailureRunView`) has a sufficiently complex "setter" method that we wonder whether DI was intended. That said, none of the classes classified as not involving use of DI could easily be changed to do so. We are left with the conclusion that either DI was not a significant consideration when designing Junit, or the level of use that we have measured is in fact indicative of good use of DI. Further study is needed in this regard.

The results for `spring framework` [Jos07], `hibernate` [Red06], and `picocontainer` [HT07] are of particular interest as all three have been described as being based around DI. That their results are three of the top four proportions (the 4th being the relatively small application `junit`) suggests that our methods for analysing software for use of DI are sound.

10.5.1 Threats to Validity

As already discussed, divining intent from code is problematic, and our conclusions must be interpreted in that light.

As we have indicated above, we believe there are false positives, meaning our results may overstate the actual usage of DI.

We have not considered service locators, mainly to limit the scope of the study to something that can be done in a reasonable amount of time. As we said earlier, our choice of DI structures was motivated by the material describing DI commonly available to developers. It would be very interesting indeed if the use of service locators was significantly higher than the structures we have studied.

Whether there are false negatives in our study is a matter of definition. There is some debate within the industry as to what is “proper” use of DI, or even whether DI is a concept separate from other concepts, such as design patterns.

Indeed, it has been suggested that what we are calling DI, is really just a particular Design Pattern. Rather than argue the point, we can only observe that Fowler, Martin, and others clearly consider DI as a distinct concept, and that alone makes it worthy of study.

Our decision to ignore fields with types from the Standard API needs to be revisited. It is conceivable that a number of classes containing just fields of such types may be a consequence of using DI.

Finally, how widely applicable our results are depends on the representativeness of our corpus. We do cover a variety of domains, although limitations of our analysis tools means we have been somewhat limited in the size of application we can consider. Nevertheless, we believe our results do indicate a significant trend, although it remains to be seen how widespread it is.

10.6 Conclusions

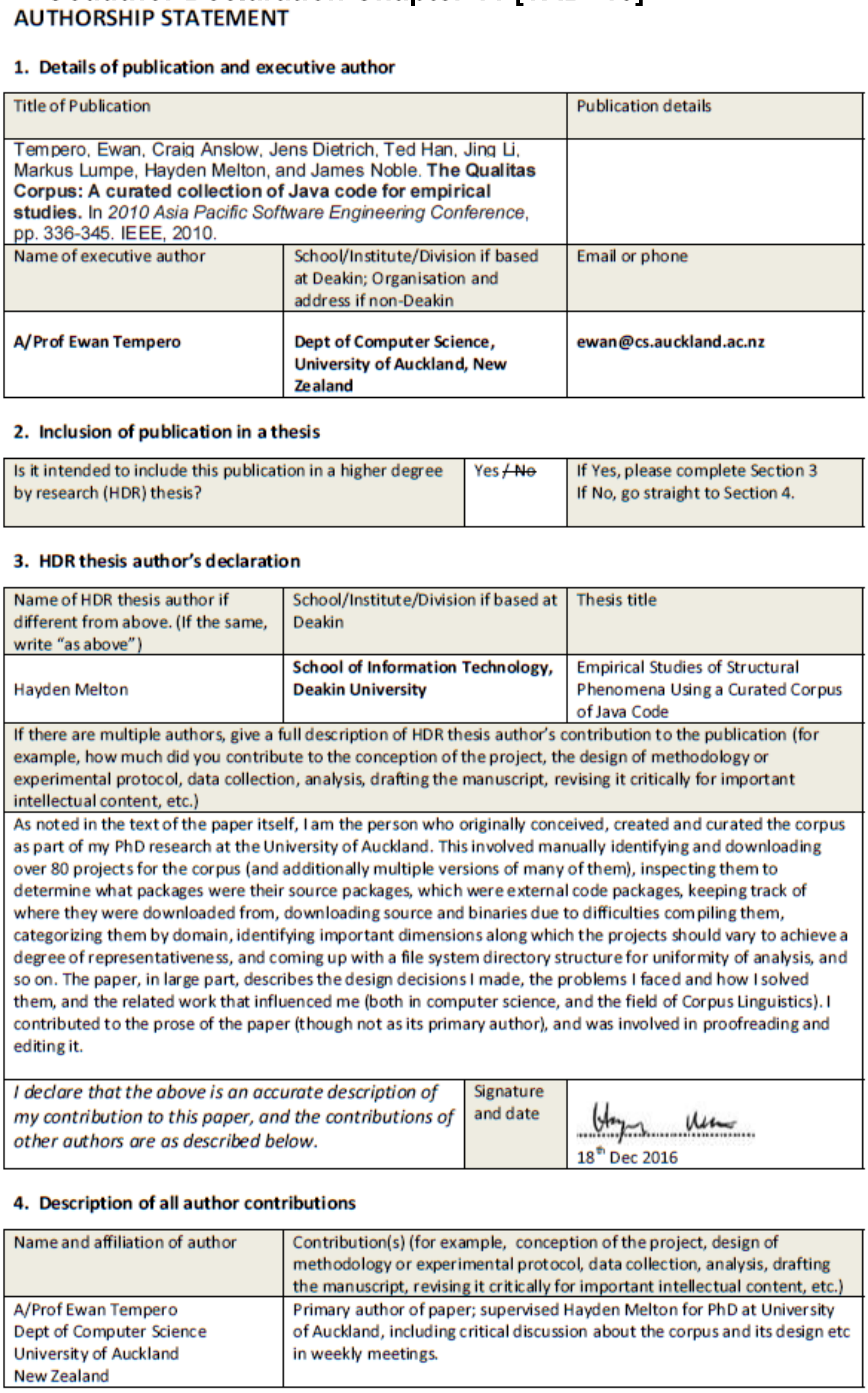
Dependency Injection is widely touted in the trade literature as a way to improve the structure of code. We have presented the analysis of 34 open-source Java applications for the evidence of the use of Dependency Injection. This represents the first study of this kind. To do so, we have identified four patterns of code structure that are consistent with DI use and developed analysis tools to recognise these structures.

Our conclusion is that, while there are individual pockets, there is not a great deal of evidence to suggest widespread use of DI. Why there is so little use of DI is a matter of conjecture. It may be that the benefits resulting from its use are not as good as claimed. It is possible that other mechanisms, such as service locators, are being used. The most likely explanation is that it is simply not taught as a matter of course in software design courses and so consequently is not that well known as a design principle.

The measurements we have obtained provide a useful starting point for developing a benchmark for DI use, however we see our main contribution as being the fact that we can make the measurements at all. Having an operational definition of DI means we can now more reliably do studies on the actual benefits of using this design principle.

There is ample future work to be done. As we have mentioned, we would like to determine how to measure the use of service locators. We would also like to make our tool more accessible. It grew out of other research we are doing and its current form is one that is sufficient to prove the concept but not useful for distribution. Ultimately we would like to carry out studies to quantify the benefits of using dependency injection.

Coauthor Declaration Chapter 11 [TAD+10]



AUTHORSHIP STATEMENT

1. Details of publication and executive author

Title of Publication		Publication details
Tempero, Ewan, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. **The Qualitas Corpus: A curated collection of Java code for empirical studies.** In *2010 Asia Pacific Software Engineering Conference*, pp. 336-345. IEEE, 2010.		
Name of executive author	School/Institute/Division if based at Deakin; Organisation and address if non-Deakin	Email or phone
A/Prof Ewan Tempero	**Dept of Computer Science, University of Auckland, New Zealand**	**ewan@cs.auckland.ac.nz**

2. Inclusion of publication in a thesis

Is it intended to include this publication in a higher degree by research (HDR) thesis?	Yes / ~~No~~	If Yes, please complete Section 3 If No, go straight to Section 4.

3. HDR thesis author's declaration

Name of HDR thesis author if different from above. (If the same, write "as above")	School/Institute/Division if based at Deakin	Thesis title
Hayden Melton	**School of Information Technology, Deakin University**	Empirical Studies of Structural Phenomena Using a Curated Corpus of Java Code
If there are multiple authors, give a full description of HDR thesis author's contribution to the publication (for example, how much did you contribute to the conception of the project, the design of methodology or experimental protocol, data collection, analysis, drafting the manuscript, revising it critically for important intellectual content, etc.)		
As noted in the text of the paper itself, I am the person who originally conceived, created and curated the corpus as part of my PhD research at the University of Auckland. This involved manually identifying and downloading over 80 projects for the corpus (and additionally multiple versions of many of them), inspecting them to determine what packages were their source packages, which were external code packages, keeping track of where they were downloaded from, downloading source and binaries due to difficulties compiling them, categorizing them by domain, identifying important dimensions along which the projects should vary to achieve a degree of representativeness, and coming up with a file system directory structure for uniformity of analysis, and so on. The paper, in large part, describes the design decisions I made, the problems I faced and how I solved them, and the related work that influenced me (both in computer science, and the field of Corpus Linguistics). I contributed to the prose of the paper (though not as its primary author), and was involved in proofreading and editing it.		
I declare that the above is an accurate description of my contribution to this paper, and the contributions of other authors are as described below.	Signature and date	[illegible] 18th Dec 2016

4. Description of all author contributions

Name and affiliation of author	Contribution(s) (for example, conception of the project, design of methodology or experimental protocol, data collection, analysis, drafting the manuscript, revising it critically for important intellectual content, etc.)
A/Prof Ewan Tempero Dept of Computer Science University of Auckland New Zealand	Primary author of paper; supervised Hayden Melton for PhD at University of Auckland, including critical discussion about the corpus and its design etc in weekly meetings.

Dr Craig Anslow Department of Computer Science Middlesex University London, UK	Discussion about the use of the corpus, promotion and dissemination of the corpus, and usage of the corpus in a number of empirical studies and tools. Co-writer of paper and provided substantial feedback.
A/Prof Jens Dietrich School of Engineering & Advanced Technology Massey University New Zealand	Discussion about the use of the corpus, provided feedback.
Ted Han	Explored and implemented various methods of metadata collection, verification and the automation of corpus distribution.
Jing Li	*(Unknown—did not respond to my email sent via LinkedIn)*
Markus Lumpe Department of Computer Science and Software Engineering (H39) Swinburne University of Technology Australia	Proposed domain classification of artefacts in the corpus. Co-writer of paper and provided editorial feedback.
Prof James Noble School of Engineering and Computer Science Victoria University of Wellington New Zealand	Suggested that the corpus was worth writing a paper about, and assisted in the preparation of the paper.

5. Author Declarations

I agree to be named as one of the authors of this work, and confirm:

i. that I have met the authorship criteria set out in the Deakin University Research Conduct Policy,

ii. that there are no other authors according to these criteria,

iii. that the description in Section 4 of my contribution(s) to this publication is accurate,

iv. that the data on which these findings are based are stored as set out in Section 7 below.

If this work is to form part of an HDR thesis as described in Sections 2 and 3, I further

v. consent to the incorporation of the publication into the candidate's HDR thesis submitted to Deakin University and, if the higher degree is awarded, the subsequent publication of the thesis by the university (subject to relevant Copyright provisions).

Name of author	Signature*	Date
Ewan Tempero	[illegible]	16/12/2016
Craig Anslow	[illegible]	01/12/2016
Jens Dietrich	[illegible]	2/12/2016
Ted Han	[illegible]	17/12/2016
Markus Lumpe	[illegible]	04/12/2016
James Noble	[illegible]	2/12/2016
Jing Li	*(Did not respond to my email sent via LinkedIn)*	

6. Other contributor declarations

I agree to be named as a non-author contributor to this work.

Name and affiliation of contributor	Contribution	Signature* and date

* If an author or contributor is unavailable or otherwise unable to sign the statement of authorship, the Head of Academic Unit may sign on their behalf, noting the reason for their unavailability, provided there is no evidence to suggest that the person would object to being named as author

7. Data storage

The original data for this project are stored in the following locations. (The locations must be within an appropriate institutional setting. If the executive author is a Deakin staff member and data are stored outside Deakin University, permission for this must be given by the Head of Academic Unit within which the executive author is based.)

Data format	Storage Location	Date lodged	Name of custodian if other than the executive author
Qualitas Corpus of Java code, both binaries and source code with metadata (see *http://qualitascorpus.com/*)	University of Auckland, New Zealand	Starting 2006, onward	Ewan Tempero

This form must be retained by the executive author, within the school or institute in which they are based.

If the publication is to be included as part of an HDR thesis, a copy of this form must be included in the thesis with the publication.

Chapter 11 The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies

In order to increase our ability to use measurement to support software development practise we need to do more analysis of code. However, empirical studies of code are expensive and their results are difficult to compare. We describe the Qualitas Corpus, a large curated collection of open source Java systems. The corpus reduces the cost of performing large empirical studies of code and supports comparison of measurements of the same artifacts. We discuss its design, organisation, and issues associated with its development.

11.1 Introduction

Measurement is fundamental to engineering, however its use in engineering software has been limited. While many software metrics have been proposed (e.g. [CK94]), few are regularly used in industry to support decision making. A key reason for this is that our understanding of the relationship between measurements we know how to make and quality attributes, such as modifiability, understandability, extensibility, reusability, and testability, that we care about is poor. This is particularly true with respect to theories regarding characteristics of software structure such as encapsulation, inheritance, coupling, and cohesion. Traditional engineering disciplines have had hundreds or thousands of years of experience of comparing measurements with quality outcomes, but central to this experience is the taking and sharing of measurements and outcomes. In contrast there have been few useful measurements of code. In this paper we describe the Qualitas Corpus, infrastructure that supports taking and sharing measurements of code artifacts.

Barriers to measuring code and understanding what the measurements mean include access to code to measure and the tools to do the measurement. The advent of open source software (OSS) has meant significantly more code is now accessible for measurement than in the past. This has led to an increase in interest in empirical studies of code. However, there is still a non-trivial cost to gathering the artifacts from enough OSS projects to make a study useful. One of the main goals of the

Qualitas Corpus is to substantially reduce the cost of performing large empirical studies of code.

However, just measuring code is not enough. We need models explaining the relationship between the measurements and the quality attributes, and we need experiments to validate those models. Validation does not come through a single experiment—experiments must be replicated. Replication requires at least understanding of the relationship between the artifacts used in the different experiments. In some forms of experiments, we want to use the same artifacts so as to be able to compare results in a meaningful way. This means we need to know in detail what artifacts are used in any experiment, meaning an ad hoc collection of code whose contents is unknown is not sufficient. What is needed is a *curated* collection of code artifacts. A second goal of the Qualitas Corpus is to support comparison of measurements of the same artifacts, that is, to provide a reference corpus for empirical studies of code.

The contributions of this paper are:

- We present arguments for the provision of a reference corpus of code for empirical studies of code.
- We identify the issues regarding performing replication of studies that analyse Java code.
- We describe the Qualitas Corpus, a curated collection of Java code that reduces the cost and increases the replicability of empirical studies.

The rest of the paper is organised as follows. In the next section we present the motivation for our work, which includes inspiration from the use of corpora in applied linguistics and the limited empirical studies of code that have been performed. We also discuss the use of reference collections in other areas of software engineering and in computer science, and discuss the need for a curated collection of code. In section III we discuss the challenges faced when doing empirical studies of code, and from that, determine the requirements of a curated corpus. Section IV presents the details of the Qualitas Corpus, its current organisation, immediate future plans, and rationale of the decisions we have taken. Section V evaluates the Qualitas Corpus. Finally we present our conclusions in section VI.

11.2 Motivation and Related Work

The use of a standard collection of artifacts to support study in an area is not new, neither in general nor in software engineering. One area is that of applied linguistics, where standard corpora are the basis for much of the research being done. Hunston [Hun02] opens her book with "*It is no exaggeration to say that corpora, and the study of corpora, have revolutionised the study of language, and of the applications of language, over the last few decades*." Ironically, it is the availability of software systems support for language corpora that has enabled this form of research, whereas researchers examining code artifacts have been slow to adopt this idea. While the goals of applied linguistics research are not exactly the same as ours, the similarities are close enough to warrant examining how corpora are used in that field. Their use of corpora is a major motivation for the Qualitas Corpus. We will discuss language corpora in more detail in section III.

11.2.1 Empirical studies of Code

To answer the question of whether a code corpus is necessary, we sample past empirical studies of code. By "empirical study of code" we mean a study in which the artifacts under investigation consist of source code, there are multiple, unrelated, artifacts, and the artifacts were developed independently of the study. This rules out, for example, studies that included the creation of the code artifacts, such as those by Briand et al. [BWDP00]or Lewis et al. [LHKS91], and studies of one system, such as that by Barry [Bar89].

Empirical studies of code have been performed for at least four decades. As with many other things, Knuth was one of the first to carry out empirical studies to understand what code that is actually written looks like [Knu71]. He presented a static analysis of over 400 FORTRAN programs, totaling about 250,000 cards, and dynamic analysis of about 25 programs. He chose programs that could "run to completion" from job submissions to Stanford's Computation Center, various subroutine libraries and scientific packages, contributions from IBM, and personal programs. His main motivation was compiler design, with the concern that compilers may not optimise for the typical case as no-one knew what the typical case was. The programs used were not identified.

In another early example, Chevance and Heidet studied 50 COBOL programs also looking at how language features are used [CH78]. The programs were also not identified and no details were given of size.

Open source software has existed for several decades, with systems such as Unix, emacs, and TEX. Their use in empirical studies is relatively recent. For example, Miller et al. [MFS90] studied about 90 Unix applications (including emacs, TEX, LATEX, yacc) to determine how they responded to input. Frakes and Pole [FP94] used Unix tools as the basis for a study on methods for searching for reusable components.

During the 1990s the number of accessible systems increased, particularly those written in C++, and consequently the number of studies increased. Chidamber and Kemerer applied their metrics to two systems, one had 634 C++ classes, the other had 1459 Smalltalk classes [CK94]. No further information on the systems was given.

Bieman and Zhao studied inheritance in 19 C++ systems, ranging from 7 classes to 922 classes in size, with 2744 classes in total [BZ95]. They identified the systems studied, but did not identify the versions for all systems.

Harrison et al. applied two coupling metrics to five collections of C++ code, consisting of 96, 197, 113, 61, and 12 classes respectively [HCN98]. They identified the systems involved but not the versions studied.

Chidamber et al. studied three systems, one with 45 C++ classes, one with 27 Objective C classes, and one identifying 25 classes in design documents [CDK98]. They were required to restrict information about the systems studied for commercial reasons.

By the end of the millennium, repositories supporting open source development such as sourceforge, as well as the increase in effectiveness of Internet search systems, meant a large number of systems were accessible. This affected both the number of studies done, and often their size. A representative set of examples include one with

3 fairly large Java systems [WC03], a study of 14 Java systems [GM05], and a study of 35 systems, from several languages including Java, C++, Self, and Smalltalk [PNFB05].

Two particularly large studies were by Succi et al. [SPD+05] and Collberg et al[CMS04]. Succi et al. studied 100 Java and 100 C++ applications. The Java applications ranged from 28 to 936 classes in size (median 83.5) and the C++ applications ranged from 30 to 2520 classes (median 59). The actual applications were not identified. Collberg et al. analysed 1132 Java jar files collected from the Internet. According to their statistics they analyse a total of 102,688 classes and 12,188 interfaces. No information was given as to what applications were analysed.

The studies described above suggest that there is interest in doing studies that involve analysing code and the ability to do such studies has significantly advanced our knowledge about the characteristics of code structure. There are several issues with these studies however. The first is that none of these studies use the same set of systems, making it difficult to compare or combine results. Another is that because full details of the systems analysed are not provided, we are limited in our ability to replicate them. A third issue is that it is not clear that even the authors are fully aware of what they have studied, which we discuss further below. Finally, while the authors have gone to some effort to gather the artifacts needed for their study, few others are able to benefit from that effort, meaning each new study requires duplicated effort. The Qualitas Corpus addresses these issues.

11.2.2 Infrastructure for empirical studies

Of course the use of standard collections of artifacts to support research in computer science and software engineering is not new. The use of benchmarks for various forms of performance testing and comparison is very mature. One recent example is the DaCapo benchmark suite by Blackburn et al. [BGH+06], which consists of a set of open source, real world Java applications with non-trivial memory loads. Another example of research infrastructure is the New Zealand Digital Library project, which provides the technology for the creation of digital libraries and is publicly available so that others can use it [WCA96].

There are also some examples in Software Engineering. One is the Software-artifact Infrastructure Repository (SIR) [DER05]. The explicit goal of SIR is to support controlled experimentation in software testing techniques. SIR provides a curated set of artifacts, including the code, test suites, and fault data. SIR represents the kind of support the Qualitas Corpus is intended to provide. We discuss SIR's motivation in the section III.

Bajracharya et al. describe Sourcerer, which provides infrastructure to support code search [BNL+06]. At the time of publication, the Sourcerer database held 1500 real-world open source projects, a total of 254,049 Java classes, gathered from Sourceforge. Their goals are different to ours, but it does give an indication as to what is available. Finally, we must mention the Purdue Benchmark Suite. This was described by Grothoff et al. in support of their work on confined types [GPV01]. It consisted of 33 Java systems, 5 with more than 200 classes, and a total of 46,165 classes. At the time it was probably the largest organised collection of Java code, and was the starting point for our work.

11.2.3 The need for curation

If two studies that analyse code give conflicting reports of some phenomena, one obvious possible explanation is that the studies were applied to different samples. If the two studies claimed to be analysing the same set of systems, we might suspect error somewhere, although it could just be that the specific versions analysed were different. In fact, even if we limit our sample to be from open source Java systems, there is still room for variation even within specific versions, as we will now discuss.

In an ideal world, it would be sufficient for a researcher to just analyse what was provided on the system's download website. However, it is not that simple. Open source Java systems come in both deployable ("binary") and source versions of the code. While we are interested in analyzing the source code, in some cases it is easier to analyse the binary version. However, it is frequently the case that what is distributed in the source version is not the same as what is in the binary version. The source often includes "infrastructure" code, such as that used for testing, code demonstrating aspects of the system, and code that supports the installation, building, or other management tasks of the code. Such code may not be representative of the deployed code, and so could bias the results of the study. In some cases, this extra code can be a significant proportion of what is available. For example, jFin_DateMath version R1-0.0 has 109 top-level non-test classes and 38 JUnit test classes. If the goal of a study is to characterize how inheritance is used, then the JUnit classes (which extend TestCase) could bias the result. Another example is fitjava version 1.1, which has 37 top level classes, and, in addition, 22 example classes. If there are many example classes, which are typically quite simple, then they would bias the results in a study to characterise some aspect of the complexity of the system design.

Another issue is identifying the infrastructure code. Different systems organise their source code in different ways. In many cases, the source code is organised as different source directories, one for the system source, one for the test infrastructure, one for examples, and so on. However there are many other organisations. For example, gt2 version 2.2-rc3 has nearly 90 different source directories, of which only about 40 contain source code that is distributed in binary form.

The presence of infrastructure code means that a decision has to be made as to what exactly to analyse. Without careful investigation, researchers may not even be aware that the infrastructure code exists and that a decision needs to be made. If this decision is not reported, then it impacts other researchers' ability to replicate the study. It may be possible to avoid this problem by just analysing the binary form of the system, as this can be expected to represent how the system was built. Unfortunately, some systems do include infrastructure code in the deployed form.

Another complication is third-party libraries. Since such software is usually not under the control of the developers of the system, including it in the analysis would be misleading in terms of understanding what decisions have been made by developers. Some systems include these libraries in their distribution and some do not. Also, different systems can use the same libraries. This means that third-party library use must be identified, and where appropriate, excluded from the analysis, to avoid bias due to double counting.

Identifying third-party libraries is not easy. Some systems are deployed as many archive (jar) files, meaning it is quite time-consuming to determine which are third-party libraries and which are not. For example, compiere version 250d has 114 archive files in its distribution. Complicating the identification of third-party libraries is the fact that some systems have such libraries packaged along with the system code, that is, the library binary code has been unpacked and then repacked with the binary system code. This means excluding library code is not just a matter of leaving out the relevant archive file.

Some systems are careful to identify what third-party systems are included in the distribution (eclipse for example). However usually this is in simple text document that must be processed by a human, and so some judgement is needed.

Another means to determine what to analyse might be to look at the code that appears in both source and binary form. Since there is no need for third-party source to be distributed, we might reasonably expect it would only appear in binary form. However, this is not the case. Some systems do in fact distribute what appears to be original source of third-party libraries (for example compiere version 250d has a copy of the Apache Element Construction Set[35] that differs only in one class and that only by a few lines). Also, some systems provide their own implementations of some third-party libraries, further complicating what is system code and what is not.

In conclusion, to study the code from a collection of systems it is not sufficient to just analysis the downloaded code, whether it is binary or the original source. Decisions need to be made regarding exactly what is going to be analysed. If these decisions are not reported, then the results may be difficult to analyse (or even fully evaluate). If the decisions are reported, then anyone wanting to replicate the study has, as well as having to recreate the collection, the additional burden of accurately recreating the decisions.

If the collection is *curated*, that is, the contents are organised and clearly identified, then the issues described above can be more easily managed. This is the purpose of the Qualitas Corpus.

[35] http://jakarta.apache.org/ecs

11.3 Designing a Corpus

In discussing the need for the Software-artifact Infrastructure Repository (SIR), Do et al. identified five challenges that need to be addressed to support controlled experimentation: supporting replicability across experiments; supporting aggregation of findings; reducing the cost of controlled experiments; obtaining sample representativeness; and isolating the effects of individual factors[DER05]. Their conclusion was that these challenges could be addressed to one degree or other by creating a collection of relevant artifacts.

When collecting artifacts, the target of those artifacts must be kept in mind. Researchers use the artifacts in SIR to determine the effectiveness of techniques and tools for testing software, that is, the artifacts themselves are not the objects of study. Similarly, benchmarks are also a collection of artifacts where they are not the object of study, but provide input to systems whose performance is the object of study. While any collection of code may be used for a variety of purposes, our interest is in the code itself, and so we refer to our collection as a corpus.

Corpora are now commonly used in linguistics and there are many used in that area, such as the International Corpus of English[Eng10]. The development of standard corpora for various kinds of linguistics work is an area of research in itself. Hunston says the main argument for using a corpus is that it provides a reliable guide to what language is like, more reliable than the intuition of native speakers [Hun02, p20]. This applies to programming languages as well. While both research and trade literature contain many claims about use of programming language features, code corpora could be used to provide evidence for such claims.

Hunston lists four aspects that should be considered when designing a corpus: *size*, *content*, *representativeness*, and *permanence*. Regarding size, she makes the point that it is possible to have too much information, making it difficult to process it in any useful way, but that generally linguistics researchers will take as much data as is available. For the Qualitas Corpus, our intent is to make it as big as is practical, given our goal of supporting replication.

According to Hunston, the content of a corpus primarily depends on the purpose it used for, and there are usually questions specific to a purpose that must be addressed in the design of the corpus. However, the design of a corpus is also impacted by what is available, and pragmatic issues such as whether the corpus creators have permission from the authors and publishers to make the contents available. The primary purpose that has guided the design of the Qualita Corpus has been to support studies involving static analysis of code. The choice of contents is due to the large number of open source Java systems that are available.

The representativeness of a corpus is important for making statements about the population it is a sample of, that is, the generalisability of any conclusions based on its study. Hunston describes a number of issues that impact the design of the corpus, but notes that the real question is how the representativeness of the corpus should be taken into account when interpreting results. The Qualitas Corpus supports this assessment by providing full details of where its entries came from, as well as metadata on such things as the domain of an entry.

Finally, Hunston notes that a corpus needs to be regularly updated in order to remain representative of the current usage, and so its design must support that.

11.4 The Qualitas Corpus

The current release is 20100719. It has 100 systems, 23 systems with multiple versions, with 495 versions total. The full distribution is 9.42GiB in size, which is 32.8GiB once installed. It contains the source and binary forms of each system version as distributed by the developers (section IV-B). The 100 systems had to meet certain criteria (section IV-C). These criteria were developed for the first external release, one consequence of which is that some systems that were considered part of the corpus previously now are not as they do not meet the criteria (section IV-I). There are questions regarding what things are in the corpus (section IV-E). The next release is scheduled for the end of October 2010 (section IV-J).

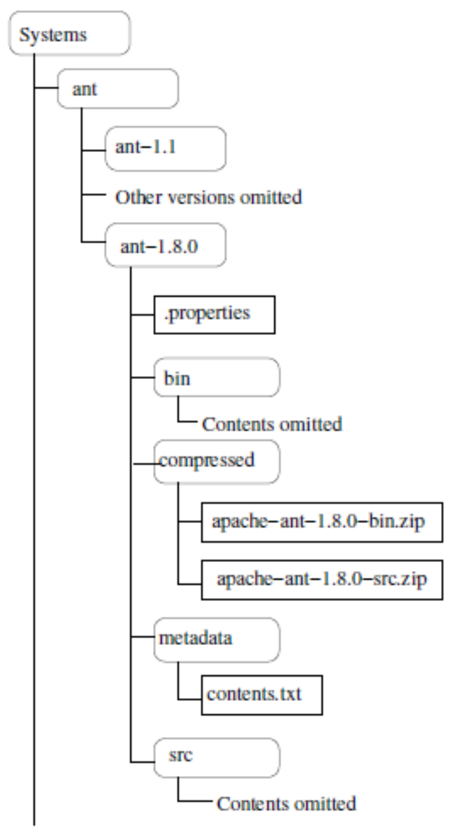


Figure 1. Organisation of Qualitas Corpus.

As discussed previously, the main goals for the corpus are that it reduces the costs of studies and supports replication of studies. These goals have impacted the criteria for inclusion and the corpus organisation.

11.4.1 Organisation

The corpus contains of a collection of **systems**, each of which consists of a set of **versions**. Each version consists of the original distribution (**compressed**) and two "unpacked" forms, **bin** and **src**. The unpacked forms are provided in order to reduce the costs of performing studies. The **bin** form contains the binary system as it was intended to be used, that is, Java bytecode. The **src** form contains everything in the source distribution. If the binary and source forms are distributed as a single archive file, then it is unpacked in **src** and the relevant files are copied into **bin**. There is also a **metadata** directory that contains detailed information about the contents of the version and a file .properties that contains information on specific attributes of the version (section IV-D).

ant antlr aoi argouml aspectJ axion azureus c_jdbc checkstyle cobertura colt columba compiere derby displaytag drawswf drjava eclipse_SDK emma exoportal findbugs fitjava fitlibraryforfitnesse freecol freecs galleon ganttproject gt2 heritrix hibernate hsqldb htm-lunit informa ireport itext ivatagroupware jFin_DateMath jag james jasml jasperreports javacc jchempaint jedit jena jext jfreechart jgraph jgraphpad jgrapht jgroupsn jhotdraw jmeter jmoney joggplayer jparse jpf jrat jre jrefactory jruby jsXe jspwiki jtopen jung junit log4j lucene marauroa megamek mvnforum myfaces_core nakedobjects nekohtml openjms oscache picocontainer pmd poi pooka proguard quartz quickserver quilt roller rssowl sablecc sandmark springframework squirrel_sql struts sunflow tomcat trove velocity webmail weka xalan xerces xmojo

Figure 2. Systems in the Qualitas Corpus.

The original distribution is provided exactly as downloaded from the system's download site. This serves several purposes. First, it means we can distribute the corpus without creating the **bin** and **src** forms, as they can be automatically created from the distributed forms, thus reducing the size of the corpus distribution. Second, it allows any user of the corpus to verify that the **bin** and **src** forms match what was distributed, or even create their own form of the corpus. Third, many distributions contain artifacts other than the code in the system, such as test and build infrastructure and so we want to keep these in case someone wishes to analyse them as well.

We use a standard naming convention to identify systems and versions. A system is identified by a string that cannot contain any occurrence of "-". A version is identified by <system>-<versionid>, where <system> is the system name, and <versionid> is some system-specific version identifier. Where possible, we use the names used by the original distribution. So far, the only case where we have not been able to do this is when the system name contains "-", which we typically replace with "_".

Figure 1 shows an example of the distribution for ant. There are 19 versions of ant, from ant-1.1 to ant-1.8.0. The original distribution of ant-1.8.0 consists of apache-ant-1.8.0-bin.zip, containing the deployable form of ant, which is unpacked in **bin**, and apache-ant-1.8.0-src.zip containing the source code, unpacked in **src**.

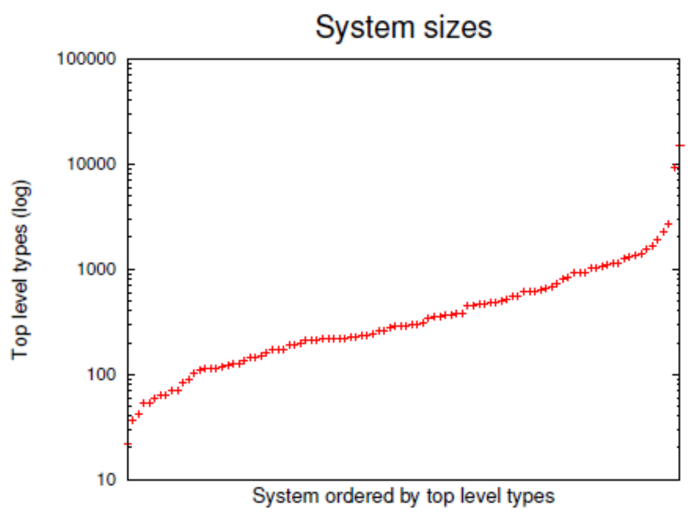


Figure 3. Distribution of sizes of systems (y is log scale).

Domain	No.
3D/graphics/media	6
IDE	4
SDK	7
database	7
diagram/visualisation	9
games	3
middleware	15
parsers/generators/make	8
programming language	2
testing	12
tool	27

Table 1: Domains Represented in the Corpus

11.4.2 Contents

Figure 2 lists the systems that are current represented in the corpus. Figure 3 gives an idea of how big the systems are, when listing the latest version of each system in the current release in order of number of top-level types (that is, classes, interfaces, enums, and annotations). Note that the y-axis is on a log scale. Table I shows the representativeness of the corpus in terms of domains represented and number of systems in each domain.

For the most part, the systems in the corpus are open source and so the corpus can contain their distributions, especially as what is in the corpus is exactly what was downloaded from the system download site. One exception to this is jre. The license agreements for the binary and source distributions appear to not allow their inclusion in the corpus. Since jre is an interesting system to analyse, we consider it part of the corpus however corpus users must download what they need from the Java distribution site. What is provided by the corpus for jre is the metadata similar to that for other systems.

11.4.3 Criteria for inclusion

Currently, the criteria for a system to be included in a release of the corpus are as follows:

1) **In the previous release** We do not want to remove things from a release that was in a previous release. This allows people to have the latest release and yet still be able to reproduce studies based on previous releases. While we intend to continue to distributed previous releases, we assume most people would prefer not to have to juggle multiple versions of the corpus.

2) **Written in Java** The choice of Java is due to both the amount of open source code available (far more than C# at the moment, although perhaps not as much as C++) and the relative ease with which it can be analysed (unlike, for example, C++). Should the opportunity arise, other languages will be added, but doing so is not a priority at the moment.

3) **Distributes both source and binary forms** One advantage with Java is that its “compiled” form is also fairly easy to analyse, easier than for the source code in fact (section IV-E), however there are slight differences between the source and binary forms. Having both forms means that analysis results from the binary form can be manually checked against the source.

In order for it to make sense to have both source and binary forms, the binary form must really be the binary form of the source. It is expensive (in time) to download source and then compile it as every project has a different build technology (e.g. ant, bat files, uses eclipse infrastructure) that takes significant effort to understand. We have made the decision to simply take what is distributed by the developers, and assume that the binary form is from the source that is distributed. For this reason, we only include systems that do actually distribute both forms in a clearly identifiable way.

This rules out, for example, systems whose source are only available through a source control system. While in theory it should be possible to extract the source relevant to a given binary release, being confident that we can extract exactly the right versions of each file is sufficiently hard that we just avoid the problem at the moment. In the future we hope to relax this, at least for systems where the relevant source version is clearly labelled.

4) **Distribute binary forms as a set of jar files** The binary form of systems included in the corpus must be bundled as .jar files, that is, not .war, .ear, etc, and not unbundled .class files. This is solely due to the expectations of our tools for managing the corpus and doing analysis using the corpus. This criterion will probably be the first to completely go away.

5) **Available to anyone independent of the corpus** This criterion is intended to avoid ephemeral systems that crop up from time to time, or systems that are only known to us that cannot be acquired by other researchers. This allows the possibility of others to independently check the decisions we have made. This is the hardest one to meet, as we cannot be sure when development will stop on some system. Some systems we used (and analysed) before the first external release of the corpus have suffered this fate, and so are not in the corpus. In fact we already have the situation where the version of a system we have in the corpus is now apparently no longer available, as the developers only appear to keep (or make available at least) the most recent versions. Due to criterion 1, we have chosen to keep these, even though they may not now be available to everyone.

6) **Identifiable contents** As discussed in section II-C, it is not always easy to determine what the contents of a system are. If there is uncertainty regarding the contents of a system, we do not include it. For example, the binary form of netbeans has 400+ jar files. Trying to determine what is relevant and what is not has proven to be a challenge that we are still struggling with, and so it is not in the corpus (yet). These criteria were developed to simplify the management of the corpus. Eventually we hope some of them will be relaxed (e.g. 2 and 4) or will have less impact (e.g. 6).

11.4.4 Metadata

As part of the curation process we gather metadata about each system version, and we will continue to improve what metadata is provided (section IV-J). The corpus provides this metadata in part to resolve the issues discussed in section II-C. Ideally we would like have the exact specification as to what the developers consider to be "in" the system however it is a very time consuming process to get such information and it is not clear that even the developers would necessarily agree amongst themselves. Instead, we follow these two principles:

• Do not include something in a given system if it could also appear in some other system in the corpus. This will avoid (or at least reduce) double-counting of code measurements that are done over the entire corpus.

• Make some decision about what is in a system and *document it*. This means that even if the decision is not necessarily the best, others trying to reproduce a given analysis will know what actually was analysed. One place where metadata is kept is in a .properties file (see Figure 1). This file is formatted so that it can be easily managed using java.util.Properties.

For example, the decision we have made regarding what is identified as being in a given version of a system is recorded in the sourcepackages field of the .properties file. This is a space-separated list of prefixes of packages of Java types. Any type whose fully-qualified name has one of the listed package prefixes as a prefix of the name is considered a type that was developed for the system, and everything else is considered as being a library type. For example, for azureus-3.0.3.4, its sourcepackages value is "org.gudy.com.aelitis", indicating that types such as com. aelitis.azureus.core.AzureusCore and org.gudy.azureus2.core3.util.FileUtil are considered part of that version of azureus, whereas org.pf.file.FileUtil (which is distributed in with azureus) would not.

Other metadata we keep in .properties includes the release date of the version, notes regarding the system and individual versions, domain information, and where the system distribution came from. The latter allows users of the corpus to check corpus contents for themselves.

The most significant development in the latest release has been the addition of significantly more metadata. We have improved the domain identification to use a more rigorous classification system (as shown in table I). We now also list, for every .java file in **src** and every .class file found in an archive in **bin**, the actual location of the file, plus information regarding how the Java type these files corresponds to is classified in the corpus.

Figure 4 shows an example of the data provided. It shows three entries for ant-1.8.0 (out of 2786). The first and third entries show that there are both .class (column 2) and .java files (column 3) corresponding to the Java types org.apache.tools.zip.ZipEntry and org. apache.tools.zip.ZipExtraField. The middle entry, for org.apache.tools.zip.ZipEntry, does not have data in column 2 indicating that while there is source code for it, it is not part of the ant deployment. Column 4 indicates whether the entry corresponds to a type identified as being in the system (that is, matches the sourcepackages value), with 0 indicating it does. Column 5 provides a summary of what forms the type exists in the corpus (0 meaning it is in both **src** and **bin**, 1 for **bin** only, and 2 for **src** only). The next column indicates whether or not the entry is for a type that is considered "distributed". Such types should also occur in **bin**, so this information can be used to identify non-public types—types that are declared in files with different names. Such types would be recorded as being not distributed but in **bin**. The remaining columns show whether types are public or non-public, number of physical lines of code, and the number of non-commented non-blank lines.

The information shown in Figure 4 is provided in a tab separated file, along with scripts that do basic analysis and which can be extended by users of the corpus.

```
...
org.[..].ZipEntry       apache-[..]/ant.jar apache-ant-1.8.0/[..]/ZipEntry.java       0 0 0 0 435 195
org.[..].ZipEntryTest                       apache-ant-1.8.0/[..]/ZipEntryTest.java   0 2 0 0 208 144
org.[..].ZipExtraField  apache-[..]/ant.jar apache-ant-1.8.0/[..]/ZipExtraField.java  0 0 0 0  85  11
...
```

Figure 4. Metadata for system version content details for ant-1.7.1. Some names have been elided for space.

11.4.5 Issues

Given the goal of replication of studies, the biggest challenge we have faced is clearly identifying the entities, as discussed in section II-C. There are, however, other issues we face. One is that systems change their name, such as the system that used to be called azureus now being called vuze. This creates the problem of whether the corpus entry should also change its name, meaning corpus users would have to be aware of this change when comparing studies done on different releases of the corpus, or maintaining the old name in the corpus. We have chosen the latter approach.

Another issue is what to do when systems stop being supported or otherwise become unavailable. One example of this issue is jgraph, which is no longer open source. Since we keep the original distribution as part of the corpus, there should be no problem with simply keeping such systems in the corpus. While we target systems we hope will be long-lived for inclusion in the corpus, we cannot guarantee that the systems will in fact continue to exist. Already there are a number of systems in the corpus that no longer appear to be actively developed (e.g., fitjava, jasml, jparse—see section IV-J). For now we will just note the status of such systems.

11.4.6 Content Management

Following criterion 1, a new release of the corpus contains all the versions of systems in the previous release. There are however some changes between releases. If there are errors in a previous release (e.g. missing or wrong metadata, misnamed systems or versions, problems with installation) then we will fix them, while providing enough information to allow people to determine how much the changes may affect attempts to reproduce previous studies.

We have developed processes over time to support the management of the corpus. The two main processes are for making a new entry of a version of a system into the corpus, and creating a distribution for release. In the early days, these were all manual, but now, with each new release, scripts are being developed to automate more parts of the process.

11.4.7 Distributing the Corpus

To install the copy one acquires a *distribution* for a particular *release*. The release indicates the decision point as to what is in the corpus and so is used for identification in studies (section IV-H). A given distribution of a release provides support for particular kinds of studies. For example, one distribution contains just the most recent version of each system in the corpus. For those interested in just "breadth" studies, this distribution is simpler to deal with (and much smaller to download). As the corpus grows in size we anticipate other distributions will be provided.

Releases are identified by their date of release (in ISO 8601 format). The full distribution uses the release date, whereas any other distribution will use the release date annotated to indicate which distribution it is. For example, the current release is 20100719 and the distribution containing only the most recent versions of systems is 20100719r.

11.4.8 Using the Corpus

The corpus is designed to be used in a specific way. A properly-installed distribution has the structure described in section IV-A. If every study is performed on the complete contents of a given release, using the metadata provided in the corpus to identify the contents of a system (in particular sourcepackages, section IV-D), then the results of those studies can be compared with good confidence that comparison is meaningful. Furthermore, what is actually studied can be described succinctly by just by indicating the release (and if necessary, particular distribution) used.

There is, however, no restriction on how the corpus can be used. It has been quite common, for example, to use a subset of its contents in studies. In such cases, in addition to identifying the release, we recommend that either what has been included be identified by listing the system versions used, or what has been left out be similarly identified. If systems not in the corpus are also used in a study, then not only do the system versions need to be identified, but some discussion regarding how the issues described in section II-C have been resolved, and, ideally, some indication as to how others can acquire the same system code distributions.

11.4.9 History

The Qualitas Corpus was initially conceived and developed by one of us (Melton) for Ph.D. research during 2005. Many of the systems were chosen because they have been used in other studies (e.g., [GPV01][GM05][PNFB05]) although not all were still available. In its first published use (the work was done in 2005 but published later) there were 21 systems in the corpus [MT07a].

The original corpus was used and added to by members of the University of Auckland group over the next three years, growing from 21 systems initially. It was made available for external release in January of 2008, containing 88 systems, 21 systems with multiple versions, a total of 214 entries. As noted earlier, some of the systems that were originally in the corpus and used in studies before its release did not meet the criteria used for the external distributions. By the end of 2008, there were 100 systems in the corpus. Since then, development of the corpus has focused on improving the quality of the corpus, in particular the metadata.

As the corpus has developed it has undergone some changes. The main changes have been in terms of the metadata that is maintained, however there has also been a change in terminology. Initially, the terminology used was that the corpus contained "versions" of "applications", however "application" implied something that functioned independently. This created confusion for such things as jgraph or springframework, which are not useful by themselves. We now use "versions" of "systems".

11.4.10 Future Plans

Our plans for the future of the corpus include growing it in size and representativeness (section V), making it easier to use for studies, and providing more "value add" in terms of metadata. As noted earlier, the next release is planned for late October 2010. The main goals for this release are to add new systems and to add the latest version of each of the existing systems.

One consequence of those outside the University of Auckland group using the corpus has been suggestions for systems to add. These will be the main candidates for new systems to be added. We will mainly consider large systems for this release. In the past such systems have typically been very expensive to process, however the scripts that produce the metadata described above will reduce that cost, making it easier to grow the corpus this way. This should allow us to, for example, include systems with complex structures such as netbeans.

Another consequence of people using the corpus is the need to perform studies different to what we originally envisaged. One example of this is that some studies need to have a complete deployable version of a system (e.g. for dynamic analysis). As we originally were only thinking of doing static analysis, we did not by default include third-party libraries in the corpus. We have now begun developing the infrastructure to provide versions that are deployable. As there are more users of the corpus, more information (such as measurements from metrics) about the systems in the corpus is being gathered. We would like to include some of these measurements as part of the metadata in the future.

11.5 Discussion

The Qualitas Corpus has been in use now for 5 years, and has been made externally available for just over 2 years. There have been over 30 publications describing studies based on its use (see http://www.cs.auckland.ac.nz/~ewan/corpus for details). Increasingly, the publications are by researchers not connected to the original development group. It is in use by about 15 research groups spread across 9 countries. It is being used for Ph.D., Masters, and undergraduate research. Some of the users have started contributing to the development of the corpus, as evidenced by the author list of this paper.

Looking at how the corpus has been used, primarily it has been used to reduce the cost for developing experiments. It is difficult to determine the cost of the development of the corpus since early on it was done as an adjunct to research, rather than the main goal. However it is certainly more than 1000 hours and could easily be double that. Any user of the corpus directly benefits from this effort. Some users have in fact used the corpus merely as a starting point and added other systems of interest to them. In some cases, those other systems have been commercial systems, allowing relatively cheap comparison between commercial and open source code.

There has been less use of the ability to replicate experiments or compare results across experiments. Given that the corpus has only been available relatively recently, this is perhaps not surprising. Once other measurements and metadata become part of the corpus itself, we hope this will change.

As Do et al. note, use of infrastructure such as the Qualitas Corpus can be both of benefit and can introduce problems [DER05]. They note that misuse by users who have not followed directions carefully can be a problem, as we have also experienced. An example of where that can be a problem with the corpus is not using the sourcepackages metadata to identify system contents, meaning it is not clear which entities have being studied.

The main issue with the corpus is its representativeness. For now, it contains only open source Java systems. This issue is faced by any empirical study, but any users of the corpus must address it when discussing their results. Hunston observes that there are limitations on the use of corpora [Hun02]. While the points she raises (other than representativeness) do not directly relate to the Qualitas Corpus, they do raise an issue that does apply. The code in the corpus shows us what a software developer *wrote*, but what it cannot tell us is the *intent* of the developer.

11.6 Conclusions

In order to increase our ability to use measurement of code to support software development practise we need to do more measurement of code in research. We have argued that this requires large, curated, corpora with which to conduct code analysis empirical studies. We have discussed the issues associated with developing such corpora and how these might impact their design.

In this paper we have presented the Qualitas Corpus, a curated collection of open-source Java systems. This corpus significantly reduces the cost of empirical studies of code by reducing the time needed to find, collect, and organize the necessary code sets to the time needed to download the corpus. The metadata provided with the corpus provides an explicit record of decisions regarding what is being studied. This means that studies conducted with the corpus are easily replicated, and the results from different kinds of studies are more likely to be able to be sensibly compared.

The Qualitas Corpus is the largest curated corpus for code analysis studies, with the current version having 495 code sets, representing 100 unique systems. The next release will significantly increase that. The corpus has been successful, in that it is now being used by groups outside its original creators, and the number and size of code analysis studies has significantly increased since it has become available. We hope that it will further encourage replication and sharing of experimental results. The corpus will continue to be expanded in content and in provision of metadata, in particular its representativeness.

Chapter 12 Conclusions and Future Work

In this chapter I describe the claimed contributions of this work, I evaluate its significance, I identify some possible criticisms of it, and finally I identify some directions for future work.

12.1 Contributions of this Work

The contributions of this work fall into roughly two categories: thematic and concrete.[36] The thematic contributions provide evidence to support or refute various general themes in the field of software engineering; the concrete relate to specific empirical findings in the papers, and to the delivery of specific novel artifacts, and whether the stated goals (per Chapter 4 [Mel06]) of this work were achieved.

Thematically, by virtue of the fact these works have been accepted in refereed venues and have become quite widely-cited, it is my claim that my thesis statement—that carefully conducted empirical studies of *just* internal attributes can help to advance knowledge in the field of software structure—has to a large extent been shown to be true. This is contrary to the views of many in the empirical software engineering community who seem to think that the only useful studies are those that seek to establish an empirical relationship between internal and external software attributes [Par03].

Also thematically, it is my claim that without the use of a well thought-out, curated corpus of Java software, the results in these works would have been far less compelling and likely would not have been accepted for publication in the quality of venues they ultimately were. My approach to evolving the corpus over the course of this research was largely influenced by Hunston's book on Corpora in Linguistics [Hun02]. There is something compelling about results being collected from curated corpus that is deliberately constructed so the software in it varies along a number of dimensions e.g., domain, size, whether library or application, and that also

[36] Other authors have categorized the nature of contributions in a manner similar to that I have here [BM85].

deliberately varies longitudinally—so for some projects in it, multiple versions are present. What is also unique about the curated corpus is it contains both source code and compiled binaries so specific causes of a structural phenomenon can be examined in the former, and so that tools to analyze the phenomenon can more easily be constructed using the latter. What is further useful about the corpus is that the actual source code written for a project is distinguished from the external code e.g., libraries on which it depends to avoid double counting. Other corpora used prior to this work, as referenced in Chapter 11 [TAD+10], generally do not possess these properties, and therefore may lead to less compelling results when used as the sample of study.

A final theme, relevant to the empirical software engineering community, is that measurement is what forces us to formalize what might otherwise be only our fuzzy intuition of things [FP96]. It is my claim that this body of work provides evidence to support this theme. I do not believe I would have come to the insights on the nature of coupling described in the introduction (and on the nature of modularity as in Chapter 8 [MT07e]) without going through this process of measuring forms of it. I also do not believe I would have had the insight on the formal (lower) limits of coupling briefly discussed in future work section below without it. Since, as described in the introduction the results of one publication in the body of work naturally led to questions that were answered in the next, it is not clear how the body of work might otherwise have progressed without the use of measurement.

12.1.1 Retrospectives on the Stated Goals

The stated goals of this work, as described in Chapter 4 [Mel06], were largely achieved, though some perhaps not in the exact way I had originally conceived. To reiterate, those goals were as follows, and a short retrospective on each is provided:

- *To better align research in software engineering with problems actually faced by practitioners.* Since all of the studies in this work were performed on real-world software systems, and cycles and long transitive dependencies were found to be quite prolific among these, if one believes cycles are "bad" (more on this later in this section), then it does seem to follow that this research is highly relevant to practitioners. The remodularization effort in Java 9 and the US Patent Grant to IBM for breaking cyclic dependencies that are both referenced later in this chapter are evidence of this. The body of PhD research conducted subsequently by Oyetoyan and described later in this chapter further supports this goal [Oye15].

- *To better study the effect of design principles on software quality.* In Chapter 4 [Mel06] I noted that studies such as the ones I ultimately performed can help us to "get the best bang for our buck" by helping us to focus our efforts on problems that actually exist in real-world software. Cycles were found to be quite prolific in Java software and are often characterized as "bad" in the instructional literature on software design [MT07b], therefore it follows that my work should help us to focus our research efforts on understanding the effects of these cycles (as it has). Again, Oyetoyan's PhD research which was largely motivated by this work confirms this [Oye15].
- *To be more scientific in our research.* It has been said that measurement and empiricism are key aspects of science [FP96], and this work certainly embodies both of these principles, with a curated corpus of Java software, and extensive measurement of the software in that corpus for the purposes of identifying and categorizing cycles.
- *To empirically establish a relationship between these [Lakos' design] principles and understandability.* Although I was unable to perform empirical studies linking cycles to external quality attributes such as understandability (largely due to time constraints), the insight that there may be an intermediate step involving an *activity* in performing such a validation is novel. In Chapter 1 I argued to this end, that sometimes—as with compilation dependencies among source files—there are strong theoretical links to activities such as verbatim reuse of source code, incremental recompilation of Java source code, and as we shall see in a later in this chapter relating to the modularization of a system by packaging of classes into jar files—and that it may make more sense to try and relate these specific activities to external quality attributes. Further, the PhD research of Oyetoyan, which largely takes over where my own PhD research left off, does perform some studies that link these cycles (an internal attribute) to external attributes such as defect density and change proneness [Oye15].
- *To evaluate ways of disseminating my results to practitioners.* I did not perform a formal evaluation of any of the approaches I took to disseminating my work to practitioners, but I did post to the mailing lists of several projects in the corpus my findings on transitive dependencies in them. Occasionally these postings led

to constructive conversations on those mailing lists (see e.g., that on ArgoUML where the developers speculate on the cause of cycles with reference to specific classes in ArgoUML, and consider deploying my Jepends tool in their nightly build process)[37]. I also setup a webpage describing my work that was not behind a "paywall" so practitioners could read it[38]. Over the years I have had a few people contact me over those postings and that webpage—and I am encouraged to see Oyetoyan (described below) actually cited that webpage in his PhD thesis [Oye15]—but it is hard to very hard to quantify the effects these things had.

- *To make the Java corpus widely-accessible*. Before I suspended my PhD at the University of Auckland, I copied the latest version of the corpus I had created to a shared disk, and explained both verbally and on an internal wikipage how I had structured it and the rationale for the various decisions I made in constructing it (e.g., why to have meta data on the download location, the source packages, why to include both binaries and source code etc) to my supervisor there (Ewan Tempero). Various modifications were made to the corpus since then—mostly what I would consider to be superficial changes—and ultimately (and pleasingly from a personal perspective for me) it was made available to the wider community by my supervisor.[39]

12.1.2 Concrete Contributions

In terms of the concrete contributions of this work, several tools were produced for extracting, quantifying and avoiding compilation dependencies among source files in Java. One was *Jepends*, which uses a modified form of Lagorio's [Lag04] algorithm to quickly infer dependencies among a Java project's source files, even if that project is not a compilable state [MT06]. This tool proved very useful for collecting the data in a number of the studies in this paper. Another tool that was produced was *Jepends-BCEL*, which examines Java byte-code of a compiled project to infer different forms of compilation dependencies among the classes in that project's source (such as Lakos' uses-in-size, uses-in-the-interface, and uses relations, adapted for Java) [MT07b]. That same tool implements Eade's mEFS algorithm, Tarjan's

[37] See e.g., ArgoUML: http://dev.axion.tigris.narkive.com/71EOIDE8/argouml-dev-structure-of-argouml-oo-design, JMeter: http://www.jmeter-archive.org/Structure-of-JMeter-OO-Design-td538956.html, JEdit: http://thread.gmane.org/gmane.editors.jedit.devel/9913 Soot: https://mailman.cs.mcgill.ca/pipermail/soot-list/2006-June/000706.html , and so on.
[38] https://www.cs.auckland.ac.nz/~hayden/research.htm
[39] http://qualitascorpus.com/

Strongly Connected Component finding algorithm, and was used to collect various other metrics for the paper of Chapter 3 [BFN+06]. Most of the source code for the Jepends-BCEL tool was made publicly available via my personal webpage at the University of Auckland[40], and this source code is what was extended by Oyetoyan in his own cycle breaking refactoring tool [OCTN15].

The final tool that was built, and that I claim to be a contribution of this work is *JooJ* the plugin for Eclipse (which I did not make available on the web, but have made available to researchers such as Oyetoyan [Oye15] at their request). The aim of this tool was largely to show that cycles can be detected in real-time and that feedback can be given to a software engineer who may have inadvertently created that cycle, in the same way that the Eclipse IDE gives immediate feedback that a line of code written by a software engineer contains a compilation error.

Contributions in terms of results are as follows:

Large cycles are common among the classes of Java software, regardless of their domain, size and nature (i.e., whether framework or application) [MT07b]. These cycles often require removal of many dependencies to break such cycles, as shown by the sizes of the approximated minimum edge feedback sets in these cycles. Oftentimes as a project evolves over time from one release to the next, cycles grow in size and connectedness. In a suite of commercial applications, developed by the same company, designs perceived to be "better" by the software engineers at that company were the ones without large cycles (perceptions of the design were solicited prior to sharing results on cycles with those software engineers). Metrics are proposed to distinguish intrinsic dependencies among classes in a domain from "unnecessary" ones, and in calculating edge feedback sets certain types of dependencies (e.g., inheritance) are excluded due to arguments that they are harder to break than others.

Non-private static members may cause cycles, in that it is possible to access such members from anywhere in a project's codebase. Even after controlling for the potentially confounding effect of size—larger classes which elsewhere have been

[40] https://www.cs.auckland.ac.nz/~hayden/software.htm

shown to have higher coupling—classes defining non-private static members are more likely to participate in cycles than those without static members [MT07d]. The use of default implementations in dependency injection, and the general lack in the extent to which it is used may also be a cause of large transitive dependencies.

Various refactoring techniques were proposed for breaking cycles, based on the examination of real code. They include extracting an interface and either passing the implementation into the clients of that interface via dependency injection or via a reference to a registry of singletons. Large values for the CRSS metric and large numbers of simple cycles through a class may indicate it is a good candidate for refactoring [MT07a]. Extract interface forms of refactoring may prove very useful in reducing large transitive dependencies because (1) it was found that dependency injection is not widely used in Java software [YTM08] and (2) because the cycles in the public parts of classes are generally much smaller than those that occur when the private parts (implementation) of a class are also considered [MT07b]

A new theory on relating internal attributes to external attributes by introducing an intermediary step was proposed. Insights in the "meaning of coupling" and the "meaning of modularity" are put forward in Chapter 1 and in Chapter 8 [MT07e] casting doubt on them in their long held classifications as *internal attributes*. It is argued that strong connections can be drawn between specific activities and these internal attributes, and that a new approach to correlating an internal attribute of code with an external quality attribute may be achieved by instead determining the extent to which the specific activity (e.g., verbatim reuse of source code) is an effective approach to reuse (or other quality attribute).

The curated corpus of Java software I conceived and developed as part of this PhD research has been made available to and become widely used by researchers around the world.[41] It has lowered their barriers to entry in performing empirical studies of structural attributes and has improved the reproducibility of their studies. Much like in the field of Corpus Linguistics, the Java corpus has become a thing of study unto itself (see e.g., [TMVB13] and [DSST17], as discussed in Chapter 1). It is my position that many of the works in this thesis would not have been accepted for

[41] http://qualitascorpus.com/

publication in such prestigious venues, nor would those results and the conclusions drawn from them have been as convincing, if the results in those papers were presented without the use of this sizeable, carefully curated corpus.

The CRSS metric and corresponding theory about why large values of it imply packages can't be both stand-alone and of manageable size is put forward. Interestingly, calculation of the metric does not involve knowledge of what packages a class belongs to at all [MT07a]. Distributions of other metrics are examined with a view to determining which are invariant between projects and which vary-by-project [BFN+06]. Among those that appear invariant in their distribution though they might initially appear to follow a power law, statistically speaking, other types of distributions might model them just as well.

12.1.3 Revisiting the Research Questions

In this subsection I *very* briefly—so as to avoid belaboring that which has already been said in the introduction and middle chapters, and at times in other sections of this conclusions chapter—provide answers to the research questions that were, in the introduction, ascribed to each published paper (chapter) in this dissertation.

RQ1: Can compilation dependencies among a Java project's source files (only) be quickly and accurately computed without external libraries, build scripts and so on, and if so what observations can one make about those compilation dependencies in real-software?
In Chapter 2 [MT06], I described a tool Jepends that implemented an adaptation of Lagorio's algorithm for inferring dependencies among a Java project's source files even when that project was missing external libraries, build scripts etc. As noted in the work the tool ran very quickly, and the computed dependencies were consistent with those appearing in the byte code of the corresponding classes, after known effects of the compilation process in Java were taken into account. What was found relating to transitive and direct compilation dependencies when the tool was run on a small number of real Java projects, was that the distribution of *direct* dependencies among classes seemed to be "power-law-ish", yet the distribution of *transitive* dependencies varied greatly apparently due to the existence of many dependency cycles among some projects' class files.

RQ2: In real Java software, which structural metrics seemingly have distributions that are invariant from project-to-project, and among those with invariant distributions are they really powerlaws?
In Chapter 3 [BFN+06], a selection of metrics were collected from the corpus of Java software and what was found was that although the distribution of many metrics (e.g., fan-in, fan-out, number of methods per class, size of methods etc) appeared similar in shape regardless of what project they were computed on, the statistics showed they may not accurately be described as power laws, but rather just as 'truncated-curve' distributions. The work concludes with the speculation that metrics capturing direct (cf. transitive) relationships may follow truncated-curve distributions because programmers, working on single source file at a time, are inherently more aware of what these metrics capture.

RQ3: What is the intended approach, goals, and outcomes of this PhD research?
In Chapter 4 [Mel06], the approach, goals and outcomes of this PhD research were set forth in this work which appeared in a doctoral symposium venue. The extent to which these things were (and were not) achieved is described in the two sections preceding this one.

RQ4: In a corpus of real Java software what do the distribution of transitive dependencies among source files look like, and what are the implications in terms of software design quality of these distributions?
In Chapter 5 [MT07a], transitive compilation dependencies were calculated over the source files in corpus of real Java software. What was found was that in many (but not all) Java projects "large" transitive dependencies existed. An argument was put forth for why these large transitive dependencies imply poor package structure—particularly, in the presence of large transitive dependencies `packages` (or units of organization "above" that of classes such as jar files) imply that those packages cannot both be of manageable size and exhibit low coupling to one another. Specific examples of refactorings in real-software and their ability to reduce transitive dependencies were demonstrated.

RQ5: In a corpus of real Java software to what extent do cyclic dependencies exist and evolve over time, and in terms of software design quality what are reasonable metrics for measuring this?

In Chapter 6 [MT07b], a major study on a large corpus of Java software comprising both commercial and open-source projects was performed and what was found—contrary to the advice reviewed in the software design literature over the past 50 years—was that long cyclic dependencies are common among the source files of real Java software. Metrics were proposed and collected for distinguishing "necessary" cyclic dependencies from "unnecessary" or "bad" ones, and for estimating the cost of breaking cycles. Cycles were found to grow over time in many projects, consistent with anecdotal observations by others on the degradation of a design over time. "Necessary" cycles, i.e., those likely expressing intrinsic interdependencies between things in the domain model, were found to be much smaller in size than "unnecessary" ones.

RQ6: Is it computationally feasible to perform whole-program analysis to identify cyclic dependencies in Java code, as that code is being written, in a manner that is tightly integrated with existing Integrated Development Environment (IDE) features?

In Chapter 7 [MT07c], a tool *JooJ* was prototyped to demonstrate that feedback could be provided in the IDE *Eclipse* to alert programmers to lines of code that induced cyclic dependencies in *real-time* as those lines of code were being written. Consistent with the real-time feedback Eclipse gives for compilation errors and other stylistic errors, "squiggles" were used to identify such lines of code. The corpus of Java software was used to demonstrate the scalability of the tool—in particular that, even on large projects, the algorithms implemented by the tool could actually provide that feedback in real-time. Various techniques such as equality-by-reference, and `WeakReference`s were used in the implementation of the tool's various algorithms to ensure adequate performance.

RQ7: Does it make sense to reason about modularity without a clear definition of it, and even with such does it make sense to do so in isolation without reference to a specific activity?

In Chapter 8 [MT07e], it was argued that one cannot reason about modularity without reference to the specific activity of one's interest. Put another way, it is nonsensical to say “modularity is improved” without reference to a specific activity such as integration testing, verbatim reuse, and so on. What was shown is that to provide a *convincing* argument about modularity one must identify the things that are *parts* in that activity, what makes them *independent* of one another in that activity, and subsequently the argument must explain why the number of parts *and* their independence from one another has increased in that context. Merely saying, for example., modularity has improved, conveys no useful information at all and reduces the term modularity to a mere platitude.

RQ8: Is the use of non-private static members in Java projects a probable cause of dependency cycles among classes in those projects?
In Chapter 9 [MT07d], a theory is put forth that non-private static members (i.e., methods and fields) cause dependency cycles among source files, because they make those members “global”. What was found was that, even after controlling for the potentially confounding effect class size, classes defining non-private static members are more likely to be involved in cycles than classes without those members, hence providing support to the theory.

RQ9: Is dependency injection widely-used in real Java projects, and if so is it used in a manner that would reduce transitive compilation dependencies?
In Chapter 10 [YTM08], noting previously from Chapter 6 [MT07b] that cycles in the public interfaces of classes were much smaller than those appearing in the totality of the class’ implementation, the question was asked if dependency injection was widely-used in Java projects. Analysis was performed on the corpus of Java software and results indicate the answer to this question was “no”, even when a weaker form of it was considered that would *not* break transitive dependencies causing cycles. That weaker form involved checking to see if, within the class receiving the injection, a default implementation of the interface being injected was instantiated by way of the `new` keyword. The implication of this would seem to be that if dependency injection were used more in real Java software, cycles may not be as prevalent as they currently are.

RQ10: What were the specific considerations, issues and limitations encountered when designing the Qualitas Corpus and what is the case for other researchers making future use of it in their empirical studies?

In Chapter 11 [TAD+10], the manner in which the work of Hunston in the field of Corpus Linguistics influenced the high-level design of what became known as the Qualitas Corpus was described. Low-level details such as the arrangement of projects into directories, separation of binary and source code, and how versions of the same project were stored, along with other metadata such as a project's source (cf. external library) packages were distinguished was also described. Challenges in gathering the corpus, making it available to others, and continuing to maintain it were identified. As noted in this chapter, and the paper itself, the corpus has become quite widely-used by researchers at other institutions around the world, so one conclusion of this might be that this paper does a good job of describing it and making the case for its relevance.

12.2 Significance and Relevance of this Work

As shown in the table below, the number of citations of the works appearing as chapters in this PhD thesis may help to support its significance and contemporaneous relevance. While the passage of time has likely contributed to these citations counts, and while some of the publications have appeared in more prestigious venues than others, this nevertheless provides evidence of the impact of this PhD research.

What may further be seen as supporting the significance and relevance of this work is the number of PhD and Masters theses to which it seems to have either directly or indirectly influenced. Perhaps the most significant and connected of these works is the PhD thesis of Oyetoyan [Oye15], which itself has resulted in a number of very high quality publications at top venues.

Bibliogra-phic Key/ Chapter	**Publication**	**Citation Count Per *Google Scholar* as at December 2016**
[MT06] / Ch.2	Hayden Melton and Ewan Tempero. Identifying refactoring opportunities by identifying dependency cycles. In Proceedings of the 29th Australasian Computer Science Conference-Volume 48, pages 35–41. Australian Computer Society, Inc., 2006.	33
[BFN+06] / Ch.3	Gareth Baxter, Marcus Frean, James Noble, Mark Rickerby, Hayden Smith, Matt Visser, Hayden Melton, and Ewan Tempero. Understanding the shape of java software. In ACM Sigplan Notices, volume 41, pages 397–412. ACM, 2006.	172
[Mel06] / Ch.4	Hayden Melton. On the usage and usefulness of OO design principles. In Companion to the 21st ACM SIGPLAN symposium on Object Oriented programming systems, languages, and applications, pages 770– 771. ACM, 2006.	7
[MT07a] / Ch.5	Hayden Melton and Ewan Tempero. The CRSS metric for package design quality. In Proceedings of the thirtieth Australasian conference on Computer science-Volume 62, pages 201–210. Australian Computer Society, Inc., 2007.	42
[MT07b] / Ch.6	Hayden Melton and Ewan Tempero. An empirical study of cycles among classes in Java. Empirical Software Engineering, 12(4):389–415, 2007.	88
[MT07c] / Ch.7	Hayden Melton and Ewan Tempero. JooJ: Real-time support for avoiding cyclic dependencies. In Proceedings of the thirtieth Australasian conference on Computer science-Volume 62, pages 87–95. Australian Computer Society, Inc., 2007.	24
[MT07e] / Ch.8	Hayden Melton and Ewan Tempero. Towards assessing modularity. In Assessment of Contemporary Modularization Techniques, 2007. ICSE Workshops ACoM'07. First International Workshop on, pages 3–3. IEEE, 2007.	6
[MT07d] / Ch.9	Hayden Melton and Ewan Tempero. Static members and cycles in Java software. In First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007), pages 136–145. IEEE, 2007.	8
[YTM08] / Ch.10	Hong Yul Yang, Ewan Tempero, and Hayden Melton. An empirical study into use of dependency injection in Java. In 19th Australian Conference on Software Engineering (aswec 2008), pages 239–247. IEEE, 2008.	29
[TAD+10] / Ch.11	Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. The Qualitas corpus: A curated collection of Java code for empirical studies. In 2010 Asia Pacific Software Engineering Conference, pages 336–345. IEEE, 2010.	186

12.2.1 Oyetoyan's PhD on Cycles in Java

Oyetoyan, in his PhD research, essentially picks up where my own research off. In my research it was found that cycles and large transitive were prevalent in real-world Java software, and that it was non-trivial to remove them (non-trivial because it was found that these cycles were not due to say, e.g., redundant *import* statements in Java classes, or the existence of a single *simple cycle*). In the introductory chapter of Oyetoyan's thesis, he concurs exactly with a major premise of my research that our study of cycles should be among the source files comprising the project only (and *not* among classes appearing in external libraries on which that application depends), and with the reasons for which I provided thereto. Oyetoyan's stated goals, again from his introductory chapter, are then: "Firstly, to collect empirical evidence of the effect of dependency cycles among internally declared types on defects and change rate. This can consequently motivate for refactoring of defect-prone cyclic components. Secondly, to realize a cycle-breaking decision support system that could assist developers and maintenance engineers to refactor dependency cycles and improve the structure of the software".

What Oyetoyan finds in his thesis and the many publications that resulted from the research in it is as follows. In a study performed on six "non-trivial" systems that defective components are more likely to be "near" cycles; that the majority of defects in the systems studied are in or near cycles; that participation in cycles may be a predictor of defect-proneness; and that defect density may be correlated with cycles [OCC13c]. In the next study, he argues that it is not the number of defects that matter per se, but rather the severity (or "criticality") of those defects [OCC13b]. In this "criticality" study he examines two applications and finds that almost all critical defects exist in components in or near cycles. In summary, and as articulately summarized by Oyetoyan himself, these two studies "indicate that components with cyclic relationships are responsible for the largest number and severity of defects and defect-prone components."

The focus of Oyetoyan's research then begins to move towards refactoring to break cycles—something that is also addressed in my own research. In his paper on refactoring to reduce defect prone components using information about cycles, he proposes measures based on diameter, radius, density, edges and nodes in cycles in attempt to "zoom in" on specific components participating in cycles that may be more defect prone than others also participating in cycles [OCC13b]. His conclusion from this work is that increasing dependencies (either with new components or new dependency relationships among existing components) seems to increase defect proneness, and that if the reverse is also true (it may or may not be, he just in his own words “hypothesizes”), then refactoring to break cycles may lead to less defect prone software.

In his next work Oyetoyan conducts a longitudinal study of four applications and makes a number of very interesting and significant findings [OCC14]. He finds in the applications under study that there is no evidence of deliberate “cycle-breaking” by developers on those systems, as they have evolved, providing evidence to support the position better tools and techniques are required for such (a topic in my own research). He also finds, related to the CRSS metric proposed in my own research, that “The results of this study support pivotal metrics such as CRSS as a metric that can be focused for optimization during cycle-breaking refactoring of defect-prone components. By minimizing the CRSS values of problematic (defect-prone) components that are in cycles, it might be possible to effectively reduce the probability of defect propagation to other components”. He additionally finds that components that “transition” into cycles become more defect-prone than those that transition out of them.

In the next work Oyetoyan (and coauthors) [OFDJ15] attempt to distinguish between “bad” and “harmless” cycles and corresponding change proneness, the former being much in the style of my own work where in my own empirical study of cycles I attempted to distinguish between such using the “uses-in-the-interface” relation, described by Lakos [Lak96]. What is found by Oyetoyan in this study of 12 Java applications is that classes in cycles are no more change prone than others, but that the former have higher change probability. The distinction here is to do with approach to measurement: one counting the frequency of changes, the other categorizing a component as changed or not changed. What is also found, for the way in which “bad” cycles are distinguished using subtype knowledge [Rie96] and containment of cycles within a package is that there is no (strong) correlation between the change proneness of components in bad cycles versus those in harmless ones.

In the final publication resulting from his PhD research Oyetoyan introduces a tool for breaking cycles using his prior observations about the correlation of my CRSS metric with defect proneness [OCTN15]. The paper indicates Oyetoyan’s tool is built on top of my own *Jepends-BCEL* tool I publicly released during the course of my own research. The paper is also largely focuses on evaluating the tool by performing refactorings on the Azuerus Java project suggested by the tool, the same project used in my CRSS paper where I also proposed refactorings. The paper notes a "Significant improvement on the strategy employed in Melton and Tempero ... by introducing a new metric IRCRSS, to identify CRSS reduction between an interface and its implementation. In this way, it is possible to improve the structural quality of the code and reduce the refactoring efforts".

On a personal note, Oyetoyan sent me a very kind email in October of 2013 (I was not even aware of his work at the time) making me aware of a number of his publications and stating that ***"[my own] work in this field of dependency cycle[s] has given [him] quite [a] lot of motivation and background to most of [his] work"***. Having now read Oyetoyan's work and he having been awarded his PhD, it is clear he has done an excellent job of building on top of that which my own PhD research started.

12.2.2 Other Closely-Related PhD and Masters Research

The other major works that seem to have benefitted to varying extents from my own research are summarized as follows. Shah's PhD thesis is on automating the breaking of dependencies among classes in Java applications [Sha13]. It uses my work in two ways (1) to justify the need for research into dependency breaking noting cyclic and large transitive dependencies are commonplace in real-world Java applications among classes and packages, and (2) as prior work in the area of identifying candidate classes for dependency breaking, and (3) the curated corpus I developed as part of my research to validate the forms of refactoring proposed. Shah notes with reference to several other datasets for empirical studies of Java code (all previously cited by my work): "Among these datasets the Qualitas Corpus turns out to be the most comprehensive and widely used dataset". Shah's research led to five publications in refereed venues between the years 2010 and 2013.

Laval's PhD thesis is identifying, avoiding and correcting unwanted package level dependencies [Lav11]. It also resulted in a number of publications in refereed venues. Laval cites my work to explain that the semantics of the software must be taken into account when breaking dependencies, to avoid breaking intrinsic dependencies that may exist among objects in the domain. My work on JooJ—the tool I created primarily for avoiding the creation of cyclic dependencies in the first place—is also cited as related work, and an evaluation is performed between the tool built by Laval and my own, to show the improvement of Laval's algorithms for identifying cycles to break, over my own.

Al-Mutawa's Master's thesis [AM13] is on classifying cyclic dependencies in Java. It makes use of the curated corpus I created as part of my research, and builds heavily on top of my research on cycles (in fact, it cites *all* of my publications). It builds on top of my research by using additional concepts from graph theory to provide new metrics on the "shape" of connections among classes in cycles. It also builds on my work on the CRSS metric and its effect on package design quality by studying the extent to which packages contain cycles among classes (it may be better for cyclically dependent classes to all belong to the same package; if they are not then the package structure will of course be cyclically dependent, as my own research notes). Al-Mutawa goes even further with this and looks at the parent-child relation among packages—something that was not considered in my own work.

Gonzalez's Master's thesis does not cite my work but has an entire chapter entitled "Addressing Cycles" [Gon13]. The broader topic of his thesis relates to a change propagation metric. Miloš' PhD thesis investigates networks in different domains—among those considered are software networks [Mil15]. Miloš investigates in degrees and out degrees per the "shape" paper on which I am coauthor [BFN+06], and find results consistent with my own on cycles [MT07b]. Schmidt's PhD thesis is on recovering and reestablishing architecture of systems whose designs have deteriorated over time [Sch14]. His work cites mine as evidence architectural deterioration is commonplace, especially the form where an otherwise layered system has become cyclically dependent.

Taube-Schock's PhD thesis in large part seems to depend on and have been inspired by some of the work in my thesis [TS12]. For starters, it uses the Qualitas Corpus extensively to test the hypothesis that in real software, high coupling is unavoidable. Further, among the studies cited as the starting point for the work is that of Chapter 3 [BFN+06]. As described earlier in the introduction chapter of this thesis, I noted that based off my first paper, the distribution of some metrics by software systems seem to be invariant across them (e.g., in-degrees and out-degrees); others seem to vary by system. The former may be unavoidable (from an empirical perspective at least), and this is perhaps a key insight that inspired Taube-Schock's work.

12.2.3 Other Related Works

In terms of individual papers citing the work in this thesis (there are too many to review each in detail), some representative and recent ones to further underscore the contemporaneity of the research topic are:

- That of IBM researchers Goldstein and Moshkovich [GM14] on automatically breaking cyclic dependencies. IBM was granted a United States Patent protection in 2016, serial number US9348583B2, for this work.
- That of Constantinou *et al.*, noting from my work especially that cycles are widespread and that we cannot expect to extract individual components if classes long cycles exist among classes [CNKS15]
- That of Caracciolo *et al.* noting from my work that cycles are prevalent in real-software, thereby justifying the tools existence, and describing the improvements of their Marea tool on my tools Jepends and JooJ [CALN16]. This paper seems to be expanded upon in Aga's Master's thesis which describes the tool's purpose as breaking dependency cycles among packages [AN15].
- The PhD thesis of Caracciolo where a whole chapter is devoted to breaking dependency cycles, and where my own study is cited as proof such cycles are prevalent in medium and large scale software projects, and the tools I build to detect and prevent cycles (Jepends and JooJ) [Car16].
- Callaú *et al.* study the extent to which developers use dynamic features of Smalltalk in real code [CRTR13]. In their related work section they seem to intimate that my paper of Chapter 6 [MT07b] has led to quite a number of other studies of *just* internal attributes that have interesting findings. For instance, the rate at which programmers transitioned to using generics, the findings seemingly indicating their adoption was dependent on just one or two programmers per project. The only earlier paper they cite in the related work is one by Knuth from 1971, where his interest was largely in compiler optimization based off features actually used by programmers.

- Assunção *et al.* are interested in breaking dependency cycles, to minimize potential stubbing costs in when conducting *integration testing*. They cite my work as evidence theirs is important, because cycles are widespread in real software. [ACVP14]
- Clarke *et al.* use a subset of the projects curated corpus in this work, and data from my paper of Chapter 6 [MT07b] to investigate a strategy for achieving an integration test ordering of classes [CPBK12].

Schiaffonati and Verdicchio identified the 50 most cited papers in the journal *Empirical Software Engineering* from 2003 to 2012 in an attempt to study trends in experimentation in the field of software engineering [SV15]. In the top 50, despite only having been published in approximately the middle of that period, my paper of Chapter 6 appears [MT07b].

A final word on the contemporaneity of my work: the in-progress PhD research of Xiao at Drexel University has led to a number of publications, but one of particular interest opens with the following sentence: “Despite decades of research on software metrics, we still cannot reliably measure if one design is more maintainable than another” [MCK+16]. The same publication collects a “decoupling level metric from 108 open source and 21 industrial projects (across multiple versions of each project) and finds long cycles among files in some of those projects, and makes observations about what changes in the code caused changes in the collected metric’s values. The approach and even subject matter is highly reminiscence of that in Chapter 6 [MT07b], though it does not cite it. The point is, that the specific topic, the general area of measuring design quality by way of structural attributes of code, and my specific approach to all of this remains highly contemporaneous and an active research area.

12.2.4 Summary of Impact on Academic Works

To summarize, based on the citations described above, the impact and relevance of this work: this work has justified the existence of many of new works, by carefully and thoroughly identifying the problems they are attempting to solve (cyclic and transitive dependencies, noting some cycles may be “unavoidable” and some may be "bad", and some cycles are more strongly connected than others), and by showing the problem is widespread in real software systems. Works in which this problem is relevant span testing, remodularization of otherwise tangled software systems via tools and algorithms, and studies of cycles themselves seeking to empirically establish connection between them and external software quality attributes, and attempting to identify "hotspots" for cycle breaking refactoring using metrics proposed in my work. Tangentially, too, this work seems to have spawned a number of other papers (e.g., those cited by Callaú et al [CRTR13]) that are similarly carefully constructed studies of internal attributes only, and that in many cases were performed on the Qualitas Corpus that was developed for my own research.

12.2.5 Potential Impact on Java itself

Besides the aforementioned academic works that cite the publications in this work, the industrial relevance of this work also warrant discussion. In the paper of Chapter 6 [MT07b] I found that the Java Runtime Environment version 1.4 contained classes involved in very big strongly connected components: the largest of which contained over top-level 900 classes, the second largest of which contained over 700 such classes. Oracle—the company that now "owns" Java—seems to have determined themselves that these large transitive dependencies are a problem, and is attempting to fix them with the new modularity constructs shipping with Java 9, due for release in 2017 [BH16].

The extent to which my work informed Oracle's decision to modularize Java is unclear, but certainly my work was published and in the public domain long before any publicly announced initiatives began on this at Oracle or in the Open JDK. Indeed on the webpage for Project Jigsaw (the codename for the modularization effort in Java) on the OpenJDK website says "The JDK is big and deeply interconnected with many undesirable dependencies between APIs and different areas of the implementation. We started the JDK modularization effort in *mid 2009* during the development of JDK 7 (emphasis added)" [42].

[42] http://openjdk.java.net/projects/jigsaw/doc/jdk-modularization.html

Interestingly, another of my works is particularly relevant to this modularization effort in Java too: the one describing the CRSS metric for package design quality [MT07a]. In this paper I argued that large transitive dependencies involving many classes would preclude a package structures that both are acyclic *and* reasonably sized in terms of the number of classes each package contains. The exact same argument applies for Jar files, which is an additional and actually important way classes can be organized in Java. One of the goals of the modularization project in Java was apparently to reduce the footprint of the JRE, especially for smaller, Internet-of-Things (IoT) devices running Java. These devices may have to fetch Jar files (cf. `packages`) from the Internet, as they run, so minimizing the size (in bytes) of the things they have to fetch is a goal. Further, these IoT devices tend to be memory constrained so that is another reason to minimize the size of the Jars they need to fetch, and required at runtime. Just like in package design, larges cycles among classes result either in large jars (to contain the cycle) or cycles among Jar files themselves, meaning all of them need to be downloaded.

On this topic of the interdependencies among the classes in the Java API Shah, in the introductory chapter of PhD thesis, provides his own visualization of dependencies among classes and packages in the Java API, further noting that the modularization effort of Java given these deep dependencies was so complicated (citing Mark Reinhold, chief Java architect) that it was delayed from a release in Java 8 to a forthcoming release in Java 9 [Sha13]. This is entirely consistent with my results in Chapter 6 [MT07b] published several years earlier on the JRE, showing cycles in the uses-in-the-interface relation, the uses-relation, and the minimum edge feedback set size.

12.3 Possible Criticisms of this Work

It would be at worst arrogant and at best shortsighted not to self-identify possible criticisms of this work of which there are, despite the contributions and significance of the work discussed earlier, potentially quite a number.

In the introductory chapter for this thesis, I made the claim that it is indisputable that software structure affects external software quality attributes, using code obfuscation as an example to support that claim. A possible criticism of this is that code obfuscators result in code that no reasonable human being would actually write. Since our interest is in maintaining, understanding, testing (and so on) code that is written by actual human beings, as part of the ongoing software development process, one might argue that it only the extent to which the structure of code—as written by human beings—varies that is relevant, to the extent that variation affects external software quality attributes.

The response to that argument lies in works like that of Arisholm and Sjoberg [AS04], which was also previously discussed in the introductory chapter. Given though, that Arisholm and Sjoberg find that the situation is more complicated than *just* structure affecting software quality attributes—recall that they find the experience (i.e., whether expert or novice) of the software engineer performing the change determines whether the centralized or delegated control structure is easier to maintain—does structure *really* matter that much? With particular reference to this work, do cyclic and large transitive compilation dependencies among source files really matter that much when it comes to external quality attributes?

An Economist might provide an argument where the answer to that question is “no”. If cycles were so utterly detrimental to software quality, then the software would not be able to be modified, maintained, understood and so on, and competition would eventually lead to its replacement by another system without cycles, the latter being of higher quality (e.g., being able to evolve more quickly to meet the new needs of users, without introducing regression faults, and so on). The works contained herein that have longitudinal analysis show that some systems with cycles continued to be released and developed, and sometimes that the cycles grew in size between releases. The conclusion seems to be that cycles, while “bad”, have not proven fatal for many real software systems, so in the larger scale of things they might not matter *that* much. This would seem to be consistent with Raccoon’s position that we have come far, overall are doing well and continue to make good progress in the field of software engineering [Rac97]. Further supporting this would be the view of technology venture capitalist Marc Andreesson that “software is eating the world”—particularly that it has and continues successfully displace the current ways of doing things in many industries (e.g., like how Amazon displaced physical bookstores, how

Netflix displaced video rental stores and so on)[43]. In spite of seemingly much software with “bad” structure, it continues to proliferate into and displace traditional ways of doing things.

Further—and related to this—it was put to me by an academic who attended a seminar on this work, that the prevalence of cycles in real-software may actually support the view that the principle of information hiding is successfully applied in wide-use by professional software engineers. The argument goes like this: if software engineers are able to modify code, without having to be concerned about what code it indirectly (transitively) depends on, then the code it does directly depend upon is doing a “good job” of hiding the details of its implementation. To this end, an empirical study may be required to determine the extent to which transitive (cf. direct) dependencies actually influence understandability of code.

Another possible criticism may pertain to transitive dependencies and API usability. In a very recent work Fontana et al. [FDW+16] also point out of Azureus (the first system downloaded in for research, as described in Chapter 1, and Chapter 2 [MT06]) that some of the cyclic dependencies in it are due to a reference between an abstract type and its subtype, the former providing a reference to its “default implementation”. They imply that, in terms of software quality, this might not be so bad. Their observation is reminiscent of the discussion in Chapter 10 [YTM08] that providing such a default implementation and inducing a cycle might actually improve API usability. Further investigation into transitive dependencies (including cycles) and their relationship to API usability—which is a very active and ongoing field of research [MS16]—may be warranted.

Another possible criticism of this work, that has not been discussed as a threat to validity in any of the publications, nor was it ever flagged as such by any of the referees of these publications is to do with what uniquely identifies a class in Java. I am embarrassed to admit that I only found this out myself in an industry job I had subsequent to the work in these papers. In Java—seemingly contrary to what may be a widely held belief—a class is *not* uniquely identified by its fully qualified class name. It is instead uniquely identified by the pair of its `ClassLoader` *and* its fully

[43] http://www.wsj.com/articles/SB10001424053111903480904576512250915629460

qualified name [LB98]. It is entirely possible, as I found doing some integration work of my employers software on a client site, that one can have a `ClassCastException` where the class being cast has the exact same fully qualified name as the type of the class it is being cast too (of course, the `ClassLoaders` for those two classes must be different for this exception to occur).

In all of the analysis done in this work, the implicit (but strictly incorrect) assumption is that a class is uniquely identified by its fully qualified name alone. In my subsequent and recent review of a sampling of projects in the corpus, thankfully, it does not appear that many applications in it implement custom `ClassLoader`s, which are needed to achieve the effect I describe above (in the sampling I reviewed, I only saw Eclipse using a custom `ClassLoader`). Therefore I do not believe this unaccounted for effect materially affects the results in any of the publications.

Some further criticisms of specific aspects of this work that were recently put to me by academic readers are that the JooJ tool of Chapter 7 lacks a usability analysis, does not discuss how the tool might work in a collaborative work development environment, provides only a fairly superficial discussion of which dependencies should be "broken" and which should stay in place e.g., through an "exclusion set" specified by the user, and how cycles might be retrospectively broken if the tool was used on an existing code base, and so on. These are all valid criticisms in that the tool does not address these things. My response to them is two-fold. First, each of these issues are major problems unto themselves, and indeed the primary focus of several PhD theses has been on how to automatically break cycles and large transitive dependencies [Sha13][Lav11]. Some, among the catalogue of ten techniques for breaking cycles identified by Lakos [ch.5,Lak96] could perhaps even be automated in a future work. Second, the main goal of the paper was to demonstrate that cycles could be detected in real-time by way of integrated tool support, because as noted in the paper (chapter) it is widely-known in both software engineering (and in engineering disciplines in general) that fixing problems earlier in a process easier, cheaper and better than fixing them later in that same process. My experiences from the works of Chapters 2 and 6 had previously shown that computing cycles, resolving compilation dependencies, computing approximate minimum edge feedback sets among class files and so on could be computationally expensive on real

Java projects, so much so that it was unclear if providing *real-time* feedback by way of “squigglies” in Eclipse would be feasible.

12.4 Future Work

Much of what might be considered future work has been undertaken in the works that cite this. Relative to the published papers in this work, these citing works are future works. Additional directions to take this work remain (and indeed this in itself is an additional indicator of contemporaneity of a body of work—that additional directions remain open from the body of work for future pursuit) and I shall describe some below.

In the introductory text for this work, I noted that with each answer came a new question, and this is how the publications in it came to be, and how they are quite concretely (cf. thematically) connected. Among the last publications in this work was that looking at static members and their relationship to cycles, and that characterizing and measuring various forms of dependency injection in Java. These two publications, especially when considered with the works of Fowler [Fow01], Stevens *et al*. [SMC74], that of Lakos on the likely shapes of the dependency graphs among components [Lak96] and the even recent PhD thesis of Taube-Schock [TB12] lead me to ask the question: what are the *theoretical* lower limits on compilation dependency coupling in Java, or any other programming language for that matter?

Both Fowler and Stevens *et al.* note that the modules of a program must all be coupled to one another in some way in order to interact with one another and ultimately to be part of the same program, but do not provide much more commentary than this. Taube-Schock takes a very *empirical* approach to answering the question in his PhD thesis. Why do my works, especially when considered in the light of these ones raise this “theoretical limits” question though?

In the lead up to the “statics” publication [MT07d] I theorized non-private static members were one way that a programmer could get hold of a reference to a “distant” class and induce a cycle. In the lead up to the “dependency injection” publication I realized that by referencing a default implementation of an interface whose implementation was only intended to be "injected" into a class might cause a vastly bigger transitive dependency for that class. These two things combined led me to realize the *fundamental* ways (with respect to transitive compilation dependencies at least) that a class can transitively depend on others: either it *instantiates* the class (and subsequently calls methods on it/accesses its variables), or it references the class *statically*, or it has a reference to the class passed in through a *formal parameter* to its constructor or other non-private method. If an object is not passed in one of these three manners then we cannot invoke its methods or reference its fields within our given class, because we would not have a valid reference to it, and therefore it would be null. (There is another case too, to do with *casting* from one type to another, but let's for now ignore this and assume our programs are type safe through the use of parameterized types, and so on.)

It would seem to follow then, that a theoretical more formal lower limit on the minimum coupling with respect to compilation dependencies, is that from the *main* method of a type-safe Java program, there should exist a directed path to every other class if the only compilation dependencies followed are those that are (1) *static references*, (2) *appear as types in the declarations of the class' non-private interface* and (3) *instantiations* of objects with the `new` keyword. Obviously it would be possible to empirical validate this with the corpus, and the said future work could spend more space explaining all of this.

There may be two important practical implications of this proposed future work on formalizing lower limits on coupling. One is that the algorithms used to automatically identify dependencies to break in works such as [Lav11][Sha13] might focus on the so-called fundamental dependencies (e.g., instantiating an object) over the more secondary ones (calling a method on an object, noting the reference to that object must have come from somewhere else, first). Another is that properties of this lower-limit model may help explain limits that have been empirically observed in other forms of run-time coupling because it is largely compile-time dependencies that determine (the approximate superset of) run-time ones (see e.g., [DHS15]).

Another future work might be to construct a corpus of say software written in C, and determine if cycles are as prevalent in procedure programming languages as what they are in object-oriented ones like Java. Szyperski has said that the features object oriented languages make them much more susceptible to dependency cycles compared to procedural languages [SGM02], yet I am unaware of any empirical studies to confirm this. To date, at least one such inter-language study of metrics exists, but it does not consider cycles or transitive dependencies [DOP+16]. Such studies may help associate certain programming language features with cycles, in a manner similar to how statics were associated with cycles in this work. That, in turn, may help inform decisions about what features to include or exclude in the programming languages of the future.

Yet another future work, might be to figure out how to express the *uses-in-name-only* technique proposed by Lakos [Lak96] in the context of C++, as a way to "break" intrinsic dependency cycles in Java. This technique in C++ involves making a forward declaration of a type's name, rather than importing its header file, so the type cannot be used in any substantive way (i.e., no methods called on it, no fields accessed on it) in C++. In more modern languages like Java one might attempt to use a generic to do the same thing, but with the so-called intrinsic dependency between an `Edge` type (e.g., `getSourceNode()`) and `Node` type (e.g., `getInboundEdges()`) it is impossible to instantiate an `Edge<T1>` with `T1` as `Node`, and `Node<T2>` with `T2` as the parameterized `Edge` type because it results in a recursive type declaration (for works dealing with problems highly reminiscent of this one see e.g., [SMPN13] [EL16]). Languages with more implicit type inference like Scala might have a type system that supports breaking even of *intrinsic* cyclic dependencies as Lakos terms them. It is also worth noting, on this specific topic, although a lot of work has been done in tool support for breaking cycles or avoiding them, there seems to have been little to none done in programming language support for avoiding them (e.g., by forbidding them or generating compiler warnings). While Java has good backwards compatibility, there is nothing to stop Oracle adding a new *optional* feature in say the Java compiler, turned on by default, that refuses to compile cyclically dependent source files. Such an approach is reminiscent of what Hatton has described as "Language Subsetting", where the features available in a language are narrowed for the purposes of improving quality attributes [Hat07].

Finally, we now live in a time where "big data" is all the rage. How might things associated with the "big data" movement be applied to this work? Consider the works of El-Emam *et al.* [EEBGR01] and my own work empirically linking static members to cycles. In both those cases a human being postulated that class size might be correlated with coupling, and attempted to control for its confounding effect with respect to another structural attribute. It is not far-fetched to imagine that machine learning techniques could be used to more automatically identify correlated and confounding effects, which in turn might lead us to new insights and theories on which structural attributes cause others. Indeed, this would be entirely consistent with Hunston's statement in the field of Linguistics, that besides learning from studies of corpora how language is actually used, such studies can also lead us to entirely new theories on it [Hun02].

.

Bibliography

[ACVP14] Wesley Klewerton Guez Assunção, Thelma Elita Colanzi, Silvia Regina Vergilio, and Aurora Trinidad Ramirez Pozo. Evaluating different strategies for integration testing of aspect-oriented programs. Journal of the Brazilian Computer Society, 20(1):1, 2014.

[AH03] Matthew Allen and Susan Horwitz. Slicing java programs that throw and catch exceptions. ACM SIGPLAN Notices, 38(10):44–54, 2003.

[AL04] John Arthorne and Chris Laffra. Official Eclipse 3.0 Faq (Eclipse Series). Addison-Wesley Professional, 2004.

[AM13] Hussain Abdullah A. Al-Mutawa. On the Classification of Cyclic Dependencies in Java Programs. Masters Thesis, Massey University, 2013.

[AN15] Bledar Aga and Oscar Nierstrasz. A tool for breaking dependency cycles between packages. Master's Thesis, University of Bern, 2015.

[ANMT08] Craig Anslow, James Noble, Stuart Marshall, and Ewan Tempero. Visualizing the word structure of java class names. In Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, pages 777–778. ACM, 2008.

[AS04] Erik Arisholm and Dag IK Sjoberg. Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software. IEEE Transactions on software engineering, 30(8):521–534, 2004.

[AYZ94] Noga Alon, Raphael Yuster, and Uri Zwick. Finding and counting given length cycles. In European Symposium on Algorithms, pages 354–364. Springer, 1994.

[Azu05] Azureus project page. http://azureus.sourceforge.net, 2005.

[BA99] Albert-László Barabási and Réka Albert.. Emergence of scaling in random networks. Science, 286(5439):509–512, 1999.

[Bar89] Brian M Barry. Prototyping a real-time embedded system in smalltalk. ACM SIGPLAN Notices, 24(10):255–265, 1989.

[Bar02] A. Barabasi. Linked: the New Science of Networks. Perseus Press, New York, 2002.

[BC00] Carliss Young Baldwin and Kim B Clark. Design rules: The power of modularity, volume 1. MIT press, 2000.

[BCK98] Len Bass, Paul Clements, and Rick Kazman. Software Architecture in Practice. Addison Wesley, Reading, USA, 1998.

[BDW98] Lionel C Briand, John W Daly, and Jurgen Wust. A unified framework for cohesion measurement in object-oriented systems. Empirical Software Engineering, 3(1):65–117, 1998.

[BDW99] Lionel C. Briand, John W. Daly, and Jurgen K Wust. A unified framework for coupling measurement in object-oriented systems. IEEE Transactions on software Engineering, 25(1):91–121, 1999.

[BE96] Grady Booch and Edward M Eykholt. Best of Booch: Designing Strategies for Object Technology, volume 7. Cambridge University Press, 1996.

[Bec97] Kent Beck. Smalltalk Best Practice Patterns. Volume 1: Coding. Prentice Hall, Englewood Cliffs, NJ, 1997.

[Ber93] Edward V Berard. Essays on object-oriented software engineering (vol. 1). Prentice-Hall, Inc., 1993.

[BFN+06] Gareth Baxter, Marcus Frean, James Noble, Mark Rickerby, Hayden Smith, Matt Visser, Hayden Melton, and Ewan Tempero. Understanding the shape of Java software. In ACM Sigplan Notices, volume 41, pages 397–412. ACM, 2006.

[BGH+06] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. In ACM Sigplan Notices, volume 41, pages 169–190. ACM, 2006.

[BH16] Neil Bartlett and Kai Hackbarth. Java 9, OSGI and the future of modularity (part 1). https://www.infoq.com/articles/ java9-osgi-future-modularity, 2016.

[Bin99] Robert Binder. Testing object-oriented systems: models, patterns, and tools. Addison-Wesley Professional, 1999.

[BJ95] Frederick P Brooks Jr. The mythical man-month (anniversary ed.). 1995.

[BJ03] Frederick P Brooks Jr. Three great challenges for half-century-old computer science. Journal of the ACM (JACM), 50(1):25–26, 2003.

[BK12] VS Bidve and Akhil Khare. A survey of coupling measurement in object oriented systems. International Journal of Advances in Engineering & Technology, 2(1):43, 2012.

[Bla01] Sue Black. Computing ripple effect for software maintenance. Journal of Software Maintenance and Evolution: Research and Practice, 13(4):263–279, 2001.

[Blo01] Joshua Bloch. Effective Java programming language guide. Addison Wesley, 2001.

[BLW03] Lionel C Briand, Yvan Labiche, and Yihong Wang. An investigation of graph-based class integration test order strategies. IEEE Transactions on Software Engineering, 29(7):594–607, 2003.

[BM85] D Brinberg and J E McGrath. Validity and the Research Process. SAGE Publications, 1985.

[BNL+06] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. Sourcerer: a search engine for open source code supporting structure-based search. In Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, pages 681–682. ACM, 2006.

[Boo87] Grady Booch. Software Component with ADA. Benjamin-Cummings Publishing Co., Inc., 1987.

[Boo91] Grady Booch. Object oriented design with applications. Redwood City. Benjamin/Cummings Publishing, 1991.

[Boo95] Grady Booch. Object solutions: managing the object-oriented project. Menlo Park, Ca.: Addison-Wesley Pub. Co. xii, 1995.

[BSW+03] James M Bieman, Greg Straw, Huxia Wang, P Willard Munger, and Roger T Alexander. Design patterns and change proneness: An examination of five evolving systems. In Software metrics symposium, 2003. Proceedings. Ninth international, pages 40–49. IEEE, 2003.

[BWDP00] Lionel C Briand, J¨urgen W¨ust, John W Daly, and D Victor Porter. Exploring the relationships between design measures and software quality in object-oriented systems. Journal of systems and software, 51(3):245– 273, 2000.

[BZ95] James M Bieman and Josephine Xia Zhao. Reuse through inheritance: A quantitative study of C++ software. ACM SIGSOFT Software Engineering Notes, 20(SI):47–52, 1995.

[CALN16] Andrea Caracciolo, Bledar Aga, Mircea Lungu, and Oscar Nierstrasz. Marea: A semi-automatic decision support system for breaking dependency cycles. In 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), volume 1, pages 482– 492. IEEE, 2016.

[Car16] Andrea Caracciolo. A Unified Approach to Architecture Conformance Checking. PhD thesis, University of Bern, March 2016

[CDK98] Shyam R Chidamber, David P Darcy, and Chris F Kemerer. Managerial use of metrics for object-oriented software: An exploratory analysis. IEEE Transactions on software Engineering, 24(8):629–639, 1998.

[CG77] Douglas W Clark and C Cordell Green. An empirical study of list structure in lisp. Communications of the ACM, 20(2):78–87, 1977.

[CH78] RJ Chevance and T Heidet. Static profile and dynamic behavior of cobol programs. ACM SIGPLAN Notices, 13(4):44–57, 1978.

[CK91] Shyam R Chidamber and Chris F Kemerer. Towards a metrics suite for object oriented design, volume 26. ACM, 1991.

[CK94] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. IEEE Transactions on software engineering, 20(6):476– 493, 1994.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. Introduction to Algorithms. The MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, 1990.

[CMS04] Christian Collberg, Ginger Myles, and Michael Stepp. An empirical study of java bytecode programs. Technical Report TR04-11, 2004.

[CN09] Christian Collberg and Jasvir Nagra. Surreptitious software: obfuscation, watermarking, and tamper-proofing for software protection. Addison-Wesley Professional, 2009.

[CNKS15] Eleni Constantinou, Athanasios Naskos, George Kakarontzas, and Ioannis Stamelos. Extracting reusable components: A semi-automated approach for complex structures. Information Processing Letters, 115(3):414–417, 2015.

[Coc89] Beth Cockerham. Parallel compilation of Ada units. In Proceedings of the conference on TRI-Ada'88, pages 147–164. ACM, 1989.

[CPBK12] Peter J Clarke, James F Power, Djuradj Babich, and Tariq M King. A testing strategy for abstract classes. Software Testing, Verification and Reliability, 22(3):147–169, 2012.

[CRK16] Wai Ting Cheung, Sukyoung Ryu, and Sunghun Kim. Development nature matters: An empirical study of code clones in javascript applications. Empirical Software Engineering, 21(2):517–564, 2016.

[CRTR13] Oscar Callaú , Romain Robbes, Éric Tanter, and David Röthlisberger.. How (and why) developers use the dynamic features of programming languages: the case of Smalltalk. Empirical Software Engineering, 18(6):1156–1194, 2013.

[CY91] Peter Coad and Edward Yourdon. Object oriented analysis. Yourdon Press, Upper Saddle River, NJ, USA, 1991.

[DER05] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Engineering, 10(4):405–435, 2005.

[DH99] Sylvia Dieckmann and Urs Hölzle. A study of the allocation behavior of the specjvm98 Java benchmarks. In European Conference on Object- Oriented Programming, pages 92–115. Springer, 1999.

[DHS15] Jens Dietrich, Nicholas Hollingum, and Bernhard Scholz. Giga-scale exhaustive points-to analysis for Java in under a minute. In ACM SIGPLAN Notices, volume 50, pages 535–551. ACM, 2015

[Dij01] Edsger W Dijkstra. The structure of the the multiprogramming system. In Classic operating systems, pages 223–236. Springer, 2001.

[DLP05] St´ephane Ducasse, Michele Lanza, and Laura Ponisio. Butterflies: A visual approach to characterize packages. In 11th IEEE International Software Metrics Symposium (METRICS'05), pages 10–pp. IEEE, 2005.

[DMTS10] Jens Dietrich, Catherine McCartin, Ewan Tempero, and Syed M Ali Shah. Barriers to modularity-an empirical study to assess the potential for modularisation of Java programs. In International Conference on the Quality of Software Architectures, pages 135–150. Springer, 2010.

[DOP+16] Giuseppe Destefanis, Marco Ortu, Simone Porru, Stephen Swift, and Michele Marchesi. A statistical comparison of Java and python software metric properties. In Proceedings of the 7th International Workshop on Emerging Trends in Software Metrics, pages 22–28. ACM, 2016.

[DSST17] Jens Dietrich, Henrik Schole, Li Sui, and Ewan Tempero. XCorpus: An executable Corpus of Java Programs. Unpublished manuscript available at https://goo.gl/ZR5QRX, 2017.

[EEBGR01] Khaled El Emam, Saida Benlarbi, Nishith Goel, and Shesh N. Rai. The confounding effect of class size on the validity of object-oriented metrics. IEEE Transactions on Software Engineering, 27(7):630–650, 2001.

[Egy06] Alexander Egyed. Instant consistency checking for the UML. In Proceedings of the 28th International conference on software engineering, pages 381–390. ACM, 2006.

[EK03] Amnon H Eden and Rick Kazman. Architecture, design, implementation. In proceedings of the 25th International Conference on Software Engineering, pages 149–159. IEEE Computer Society, 2003.

[EL16] Michael D Ekstrand and Michael Ludwig. Dependency injection with static analysis and context-aware policy. Journal of Object Technology, 15(1), 2016.

[ELS93] Peter Eades, Xuemin Lin, and William F Smyth. A fast and effective heuristic for the feedback arc set problem. Information Processing Letters, 47(6):319–323, 1993.

[Eme62] James C Emery. Modular data processing systems written in cobol. Communications of the ACM, 5(5):263–268, 1962.

[Eng10] International Corpus of English. http://ice-corpora.net/ice, June 2010.

[ES84] Kate Ehrlich and Elliot Soloway. An empirical investigation of the tacit plan knowledge in programming. In Human factors in computer systems, pages 113–133. Norwood, NJ: Ablex Publishing Co, 1984.

[FBB99] Martin Fowler, Kent Beck, and John Brant. Refactoring: improving the design of existing code. Addison-Wesley, 1999.

[FDW+16] Francesca Arcelli Fontana, Jens Dietrich, Bartosz Walter, Aiko Yamashita, and Marco Zanoni. Antipattern and code smell false positives: Preliminary conceptualization and classification. In 2016 IEEE 23rd International Conference on Software Analysis, Evolution,2016.

[Fen94] Norman Fenton. Software measurement: A necessary scientific basis. IEEE Transactions on software engineering, 20(3):199–206, 1994.

[FLW00] Mohamed E Fayad, Mauri Laitinen, and Robert P Ward. Thinking objectively: software engineering in the small. Communications of the ACM, 43(3):115–118, 2000.

[FM96] Norman Fenton and Austin Melton. Measurement theory and software measurement. Software Measurement, pages 27–38, 1996.

[Fow01] Martin Fowler. Reducing coupling. IEEE Software, 18(4):102, 2001.

[Fow04] Martin Fowler. Inversion of control containers and the dependency injection pattern. 2004.

[Fow05] Martin Fowler. Inversion of control. Martin Fowler's Bliki, 2005.

[FP94] William B. Frakes and Thomas P. Pole. An empirical study of representation methods for reusable software components. IEEE Transactions on Software Engineering, 20(8):617–630, 1994.

[FP96] N.E. Fenton and S.L. Pfleeger. Software metrics: a rigorous and practical approach. International Thomson Computer Press, 1996.

[FY97] Brian Foote and Joseph Yoder. Big ball of mud. Pattern languages of program design, 4:654–692, 1997.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: elements of reusable object-oriented software. Addison-Wesley, 1995.

[GJC92] John E Gaffney Jr and RD Cruickshank. A general economics model of software reuse. In Proceedings of the 14th international conference on Software engineering, pages 327–337. ACM, 1992.

[GM05] Joseph Yossi Gil and Itay Maman. Micro patterns in Java code. ACM SIGPLAN Notices, 40(10):97–116, 2005.

[GM14] Maayan Goldstein and Dany Moshkovich. Improving software through automatic untangling of cyclic dependencies. In Companion Proceedings of the 36th International Conference on Software Engineering, pages 155–164. ACM, 2014.

[Gon13] Marco A Gonzalez. A new change propagation metric to assess software evolvability. PhD thesis, University of British Columbia, 2013.

[Gos00] James Gosling. The Java language specification. Addison-Wesley Professional, 2000.

[GPV01] Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with confined types. ACM SIGPLAN Notices, 36(11):241–255, 2001.

[GY04] Jonathan L Gross and Jay Yellen. Handbook of graph theory. CRC press, 2004.

[Hat98] Les Hatton. Does OO sync with how we think? IEEE software, 15(3):46– 54, 1998.

[Hat07] Les Hatton. Language subsetting in an industrial context: A comparison of MISRA C 1998 and MISRA C 2004. Information and Software Technology, 49(5), 475-482, 2007.

[Hat09] Les Hatton. Power-law distributions of component size in general software systems. IEEE Transactions on Software Engineering, 35(4):566– 572, 2009.

[Hau02] Edwin Hautus. Improving Java software through package structure analysis. In The 6th IASTED International Conference Software Engineering and Applications, 2002.

[HBS+12] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In 2012 34th International Conference on Software Engineering (ICSE), pages 837–847. IEEE, 2012.

[HCN98] Rachel Harrison, Steve Counsell, and Reuben Nithi. Coupling metrics for object-oriented design. In Software Metrics Symposium, 1998.

Metrics 1998. Proceedings. Fifth International, pages 150–157. IEEE, 1998.

[Hee03] Jan Heering. Quantification of structural information: on a question raised by brooks. ACM SIGSOFT Software Engineering Notes, 28(3):6– 6, 2003.

[HSR05] Nor Laily Hashim, Heinz W Schmidt, and Sita Ramakrishnan. Test order for class-based integration testing of Java applications. In Fifth International Conference on Quality Software (QSIC'05), pages 11–18. IEEE, 2005.

[HT07] A. Hellesoy and J. Tirsen. Picocontainer introduction. http://www.picocontainer.org/, 2007.

[Hun02] Susan Hunston. Corpora in applied linguistics. Cambridge University Press, 2002.

[IEE90] IEEE Standards Committee. Ieee standard glossary of software engineering terminology ieee std 610.12-1990, 1990.

[Ind] Indus project site. http://indus.projects.cis.ksu.edu/.

[JF88] Ralph E Johnson and Brian Foote. Designing reusable classes. Journal of object-oriented programming, 1(2):22–35, 1988.

[Jon86] C. Jones. Programming Productivity. McGraw-Hill, 1986.

[Jos07] B. Jose. The spring framework. http://javaboutique.internet.com/tutorials/springframe/article.html, 2007.

[Jun02] Stefan Jungmayr. Identifying test-critical dependencies. In Software Maintenance, 2002. Proceedings. International Conference on, pages 404–413. IEEE, 2002.

[KCM07] Huzefa Kagdi, Michael L Collard, and Jonathan I Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. Journal of software maintenance and evolution: Research and practice, 19(2):77–131, 2007.

[KDJ04] Barbara A Kitchenham, Tore Dyba, and Magne Jorgensen. Evidencebased software engineering. In Proceedings of the 26th international conference on software engineering, pages 273–281. IEEE Computer Society, 2004.

[Ker04] Joshua Kerievsky. Refactoring to patterns. Pearson Higher Education, 2004.

[KGH+93] CH Kung, Jerry Gao, Pei Hsia, Jeremy Lin, and Y Yoyoshima. Design recovery for software testing of object-oriented programs. In Reverse Engineering, 1993., Proceedings of Working Conference on, pages 202– 211. IEEE, 1993.

[KGH+95a] David Kung, Jerry Gao, Pei Hsia, Yasufumi Toyoshima, Chris Chen, Young-Si Kim, and Young-Kee Song. Developing an object-oriented software testing and maintenance environment. Communications of the ACM, 38(10):75–87, 1995.

[KGH+95b] David Chenho Kung, Jerry Gao, Pei Hsia, Jeremy Lin, and Yasufumi Toyoshima. Class firewall, test order, and regression testing of objectoriented programs. JOOP, 8(2):51–65, 1995.

[KMC72] David J Kuck, Yoichi Muraoka, and Shyh-Ching Chen. On the number of operations simultaneously executable in fortran-like programs and their resulting speedup. IEEE Transactions on Computers, 100(12):1293– 1310, 1972.

[Knu71] Donald E Knuth. An empirical study of fortran programs. Software: Practice and Experience, 1(2):105–133, 1971.

[Kru00] Philippe Kruchten. The Rational Unified Process: An Introduction, Second Edition. Addison-Wesley Professional, 2000.

[Lag04] Giovanni Lagorio. Capturing ghost dependencies in Java sources. Journal of Object Technology, 3(11):77–95, December 2004. OOPS Track at the 19th ACM Symposium on Applied Computing, SAC 2004.

[Lak96] John Lakos. Large-scale C++ software design. Addison-Wesley, 1996.

[Lav11] Jannik Laval. Package Dependencies Analysis and Remediation in Object-Oriented Systems. PhD thesis, INRIA Lille, 2011.

[LB98] Sheng Liang and Gilad Bracha. Dynamic class loading in the java virtual machine. Acm sigplan notices, 33(10):36–44, 1998.

[LHKS91] John A Lewis, Sallie M Henry, Dennis G Kafura, and Robert S Schulman. An empirical study of the object-oriented paradigm and software reuse. In ACM SigPlan Notices, volume 26, pages 184–196. ACM, 1991.

[LLLB+98] Bruno Lague, Charles Leduc, Andre Le Bon, Ettore Merlo, and Michel Dagenais. An analysis framework for understanding layered software architectures. In Program Comprehension, 1998. IWPC'98. Proceedings., 6th International Workshop on, pages 37–44. IEEE, 1998.

[LRW+97] Meir M Lehman, Juan F Ramil, Paul D Wernick, Dewayne E Perry, and Wladyslaw M Turski. Metrics and laws of software evolution-the nineties view. In Software Metrics Symposium, 1997. Proceedings., Fourth International, pages 20–32. IEEE, 1997.

[LS98] Jean Laherrere and Didier Sornette. Stretched exponential distributions in nature and economy:fat tails with characteristic scales.

The European Physical Journal B-Condensed Matter and Complex Systems, 2(4):525– 539, 1998.

[LY99] T. Lindholm and F. Yellin. The Java Virtual Machine Specification. Java (Addison-Wesley). Addison-Wesley, 1999. 8

[Mar96a] Robert C Martin. The dependency inversion principle. C++ Report, 8(6):61–66, 1996.

[Mar96b] Robert C Martin. Granularity. C++ Report, 8(10):57–62, 1996.

[MCK+16] Ran Mo, Yuanfang Cai, Rick Kazman, Lu Xiao, and Qiong Feng. Decoupling level: a new metric for architectural maintenance complexity. In Proceedings of the 38th International Conference on Software Engineering, pages 499–510. ACM, 2016.

[Mel06] Hayden Melton. On the usage and usefulness of oo design principles. In Companion to the 21st ACM SIGPLAN symposium on Objectoriented programming systems, languages, and applications, pages 770– 771. ACM, 2006.

[Mey95] Bertrand Meyer. Object success: a manager's guide to object orientation, its impact on the corporation, and its use for reengineering the software process. Prentice-Hall, Inc., 1995.

[Mey97] B. Meyer. Object-oriented Software Construction. Object-oriented programming. Prentice Hall PTR, 1997.

[MFC01] Tim Mackinnon, Steve Freeman, and Philip Craig. Endo-testing: unit testing with mock objects. Extreme programming examined, pages 287– 301, 2001.

[MFS90] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. Communications of the ACM, 33(12):32– 44, 1990.

[Mil56] George A Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. Psychological review, 63(2):81, 1956.

[Mil15] Savi´c Miloˇs. Extraction and analysis of complex networks from different domains. PhD thesis, 2015.

[MPST04] Michele Marchesi, Sandro Pinna, Nicola Serra, and Stefano Tuveri. Power laws in smalltalk. In Proc. of the ESUG Conf, pages 27–44, 2004.

[MS16] Brad A Myers and Jeffrey Stylos. Improving API usability. Communications of the ACM, 59(6):62–69, 2016.

[MT06] Hayden Melton and Ewan Tempero. Identifying refactoring opportunities by identifying dependency cycles. In Proceedings of the

29th Australasian Computer Science Conference-Volume 48, pages 35–41. Australian Computer Society, Inc., 2006.

[MT07a] Hayden Melton and Ewan Tempero. The crss metric for package design quality. In Proceedings of the thirtieth Australasian conference on Computer science-Volume 62, pages 201–210. Australian Computer Society, Inc., 2007.

[MT07b] Hayden Melton and Ewan Tempero. An empirical study of cycles among classes in Java. Empirical Software Engineering, 12(4):389–415, 2007.

[MT07c] Hayden Melton and Ewan Tempero. Jooj: Real-time support for avoiding cyclic dependencies. In Proceedings of the thirtieth Australasian conference on Computer science-Volume 62, pages 87–95. Australian Computer Society, Inc., 2007.

[MT07d] Hayden Melton and Ewan Tempero. Static members and cycles in Java software. In First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007), pages 136–145. IEEE, 2007.

[MT07e] Hayden Melton and Ewan Tempero. Towards assessing modularity. In Assessment of Contemporary Modularization Techniques, 2007. ICSE Workshops ACoM'07. First International Workshop on, pages 3–3. IEEE, 2007.

[NB01] James Noble and Robert Biddle. Visualising 1,051 visual programs module choice and layout in the nord modular patch language. In Proceedings of the 2001 Asia-Pacific symposium on Information visualisation- Volume 9, pages 121–127. Australian Computer Society, Inc., 2001.

[NB03] James Noble and Robert Biddle. Software visualization, chapter visual program visualisation, 2003.

[New05] Mark EJ Newman. Power laws, pareto distributions and zipf's law. Contemporary physics, 46(5):323–351, 2005.

[Obj04] Object Management Group. Unified modeling language (uml) 1.5 specification, 2004.

[OCC13a] Tosin Daniel Oyetoyan, Reidar Conradi, and Daniela Soares Cruzes. Criticality of defects in cyclic dependent components. In Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on, pages 21–30. IEEE, 2013.

[OCC13b] Tosin Daniel Oyetoyan, Daniela S Cruzes, and Reidar Conradi. Can refactoring cyclic dependent components reduce defect-proneness? In ICSM, pages 420–423, 2013.

[OCC13c] Tosin Daniel Oyetoyan, Daniela S Cruzes, and Reidar Conradi. A study of cyclic dependencies on defect profile of software components. Journal of Systems and Software, 86(12):3162–3182, 2013.

[OCC14] Tosin Daniel Oyetoyan, Daniela Soares Cruzes, and Reidar Conradi. Transition and defect patterns of components in dependency cycles during software evolution. In Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week- IEEE Conference on, pages 283–292. IEEE, 2014.

[OCTN15] Tosin Daniel Oyetoyan, Daniela Soares Cruzes, and Christian Thurmann-Nielsen. A decision support system to refactor class cycles. In Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on, pages 231–240. IEEE, 2015.

[OFDJ15] Tosin Daniel Oyetoyan, Jean-R´emy Falleri, Jens Dietrich, and Kamil Jezek. Circular dependencies and change-proneness: An empirical study. In 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pages 241–250. IEEE, 2015.

[OSV16] Martin Odersky, Lex Spoon, and Bill Venners. Programming in Scala (3rd ed). Artima Inc, 2016.

[Oye15] Tosin Daniel Oyetoyan. Dependency cycles in software systems: quality issues and opportunities for refactoring. PhD thesis, NTNU, 2015.

[Par72] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. Communications of the ACM, 15(12):1053–1058, 1972.

[Par79] David Lorge Parnas. Designing software for ease of extension and contraction. IEEE transactions on software engineering, (2):128–138, 1979.

[Par94] David Lorge Parnas. Software aging. In Proceedings of the 16th international conference on Software engineering, pages 279–287. IEEE Computer Society Press, 1994.

[Par96] David Lorge Parnas. Why software jewels are rare. Computer, 29(2):57– 60, 1996.

[Par03] David Lorge Parnas. The limits of empirical studies of software engineering. In Empirical Software Engineering, 2003. ISESE 2003. Proceedings. 2003 International Symposium on, pages 2–5. IEEE, 2003.

[PNB04] Alex Potanin, James Noble, and Robert Biddle. Checking ownership and confinement. Concurrency and Computation: Practice and Experience, 16(7):671–687, 2004.

[PNFB05] Alex Potanin, James Noble, Marcus Frean, and Robert Biddle. Scale-free geometry in OO programs. Communications of the ACM, 48(5):99–103, 2005.

[Pre01] R.S. Pressman. Software Engineering: A Practitioner's Approach (5th Ed). McGraw-Hill, New York, NY, USA., 2001.

[PV03] Sandeep Purao and Vijay Vaishnavi. Product metrics for object-oriented systems. ACM Computing Surveys (CSUR), 35(2):191–221, 2003.

[Rac97] LBS Raccoon. Fifty years of progress in software engineering. ACM SIGSOFT Software Engineering Notes, 22(1):88–104, 1997.

[Ran02] Venkatesh Prasad Ranganath. Object-flow analysis for optimizing finite-state models of java software. PhD thesis, Kansas State University, 2002.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William E. Lorensen. Object-oriented modeling and design. Prentice-hall Englewood Cliffs, NJ, 1991.

[Red06] Red Hat Middleware, LLC. Preparing daos with manual dependency injection. http://www.hibernate.org/328.html#A5, 2006.

[RHR98] Jason E Robbins, David M Hilbert, and David F Redmiles. Software architecture critics in argo. In Proceedings of the 3rd international conference on Intelligent user interfaces, pages 141–144. ACM, 1998.

[Rie96] Arthur J Riel. Object-oriented design heuristics. Addison-Wesley Longman Publishing Co., Inc., 1996.

[SC07] Jeffrey Stylos and Steven Clarke. Usability implications of requiring parameters in objects' constructors. In Proceedings of the 29th international conference on Software Engineering, pages 529–539. IEEE Computer Society, 2007.

[Sch14] Frederik Schmidt. A multi-objective architecture reconstruction approach. PhD thesis, Auckland University of Technology, 2014.

[SEG+06] Cara Stein, Letha Etzkorn, Sampson Gholston, Phillip Farrington, and Julie Fortune. A knowledge-based cohesion metric for object-oriented software. INFOCOMP Journal of Computer Science, 5(4):44–53, 2006.

[SFCMV02] Ricard V Sole, Ramon Ferrer-Cancho, Jose M Montoya, and Sergi Valverde. Selection, tinkering, and emergence in complex networks. Complexity, 8(1):20–33, 2002.

[SGM02] C. Szyperski, D. Gruntz, and S. Murer. Component Software: Beyond Object-oriented Programming. ACM Press Series. ACM Press, 2002.

[SGN04] Douglas C Schmidt, Aniruddha Gokhale, and Balachandran Natarajan. Leveraging application frameworks. Queue, 2(5):66, 2004.

[Sha13] Syed Muhammad Ali Shah. On the automation of dependency-breaking refactorings in Java. PhD thesis, Massey University, 2013.

[SJSJ05] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In ACM Sigplan Notices, volume 40, pages 167–176. ACM, 2005.

[Ski98] Steven S Skiena. The algorithm design manual: Text, volume 1. Springer Science & Business Media, 1998.

[SM92] Sally Shlaer and Stephen J Mellor. Object lifecycles: modelingthe world in states. Yourdon Press, Upper Saddle River, NJ, USA, 1992.

[SMC74] Wayne P. Stevens, Glenford J. Myers, and Larry L. Constantine. Structured design. IBM Systems Journal, 13(2):115–139, 1974.

[SMPN13] Marco Servetto, Julian Mackay, Alex Potanin, and James Noble. The billion-dollar fix: Safe modular circular initialisation with placeholders and placeholder types. In ECOOP 2013–Object-Oriented Programming: 27th European Conference, Montpellier, France, July 1-5, 2013, Proceedings, volume 7920, page 205. Springer, 2013.

[SPD+05] Giancarlo Succi, Witold Pedrycz, Snezana Djokic, Paolo Zuliani, and Barbara Russo. An empirical exploration of the distributions of the chidamber and kemerer object-oriented metrics suite. Empirical Software Engineering, 10(1):81–104, 2005.

[Sun99] Sun. Code conventions for the java programming. http://java.sun.com/docs/codeconv/, June 1999.

[SV15] Viola Schiaffonati and Mario Verdicchio. Rethinking experiments in a socio-technical perspective: The case of software engineering. Philosophies, 1(1):87–101, 2015.

[Swe85] Richard E Sweet. The mesa programming environment. In ACM SIGPLAN Notices, volume 20, pages 216–229. ACM, 1985.

[TAD+10] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. The qualitas corpus: A curated collection of java code for empirical studies. In 2010 Asia Pacific Software Engineering Conference, pages 336–345. IEEE, 2010.

[Tem13] Ewan Tempero. Towards a curated collection of code clones. In Proceedings of the 7th International Workshop on Software Clones, pages 53–59. IEEE Press, 2013.

[TH02] Dave Thomas and Andy Hunt. Mock objects. IEEE Software, 19(3):22– 24, 2002.

[Tic98] Walter F Tichy. Should computer scientists experiment. more?, IEEE Computer, 1998.

[Tip95] Frank Tip. A survey of program slicing techniques. Journal of programming languages, 3(3):121–189, 1995.

[TMVB13] Ricardo Terra, Luis Fernando Miranda, Marco Tulio Valente, and Roberto S Bigonha. Qualitas. class corpus: A compiled version of the Qualitas corpus. ACM SIGSOFT Software Engineering Notes, 38(5):1–4, 2013.

[TS12] Craig Taube-Schock. Patterns of Change: Can modifiable software have high coupling? PhD thesis, University of Waikato, 2012.

[USH+16] Phillip Merlin Uesbeck, Andreas Stefik, Stefan Hanenberg, Jan Pedersen, and Patrick Daleiden. An empirical study on the impact of C++ lambdas and programmer experience. In Proceedings of the 38th International Conference on Software Engineering, pages 760–771. ACM, 2016.

[VCS02] Sergi Valverde, R Ferrer Cancho, and Richard V Sole. Scale-free networks from optimal design. EPL (Europhysics Letters), 60(4):512, 2002.

[VRCG+99] Raja Vall´ee-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot-a Java bytecode optimization framework. In Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research, page 13. IBM Press, 1999.

[VS03] Sergi Valverde and Ricard V Sole. Hierarchical small worlds in software architecture. arXiv preprint cond-mat/0307278, 2003.

[WBWW90] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. Designing object-oriented software. 1990.

[WC03] Richard Wheeldon and Steve Counsell. Power law distributions in class relationships. In Source Code Analysis and Manipulation, 2003. Proceedings. Third IEEE International Workshop on, pages 45–54. IEEE, 2003.

[WCA96] Ian H Witten, Sally Jo Cunningham, and Mark D Apperley. The New Zealand digital library project. D-Lib magazine, 2(11), 1996.

[Wei51] Waloddi Weibull. A statistical distribution function of wide applicability. Journal of applied mechanics, 103:293–297, 1951.

[Wei85] Gerald M Weinberg. The psychology of computer programming,. Van Nostrand Reinhold New York, 1985.

[Win99] Mario Winter. Managing object-oriented integration and regression testing. arXiv preprint cs/9902008, 1999.

[Wir95] Niklaus Wirth. A plea for lean software. Computer, 28(2):64–68, 1995.

[WW90] Yair Wand and Ron Weber. An ontological model of an information system. IEEE transactions on software engineering, 16(11):1282–1292, 1990.

[YDFM03] Yijun Yu, Homy Dayani-Fard, and John Mylopoulos. Removing false code dependencies to speedup software build processes. In Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research, pages 343–352. IBM Press, 2003.

[YTB05] Hong Yul Yang, Ewan Tempero, and Rebecca Berrigan. Detecting indirect coupling. In 2005 Australian Software Engineering Conference, pages 212–221. IEEE, 2005.

[YTM08] Hong Yul Yang, Ewan Tempero, and Hayden Melton. An empirical study into use of dependency injection in Java. In 19th Australian Conference on Software Engineering (aswec 2008), pages 239–247. IEEE, 2008.