The CRSS Metric for Package Design Quality

Hayden Melton, Ewan Tempero
Department of Computer Science
University of Auckland
Auckland, New Zealand
{**hayden**|**ewan**}**@cs.auckland.ac.nz**

Abstract

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.

1 Introduction

The classes in an object-oriented software system can be partitioned into groups, or *subsystems* (Wirfs-Brock, Wilkerson & Wiener 1990, 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 (Booch 1991). In large-scale software systems, comprising thousands of classes, subsystems are absolutely essential (Lakos 1996, Booch 1991, Coad & Yourdon 1991, Martin 1996*b*). They help us to avoid information overload — a result of the limits on the human mind's information-processing capacity (Coad & Yourdon 1991). They facilitate a vocabulary that developers of a system can use in communication (Coad & Yourdon 1991, Booch 1991). They allow managers to determine a *partial ordering* of development activities with respect to time, that allows for parallelism in development effort (Meyer 1995, Lakos 1996, Martin 1996*b*). Finally, they have a significant impact on the system's quality particularly with respect to reusability and testability (Lakos 1996, Martin 1996*b*, Szyperski 1998).

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 (Martin 1996*b*). 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 (Martin 1996*b*). 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 consitutes 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 empiricial 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.

2 Background

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 (Lakos 1996, Wirfs-Brock et al. 1990, Coad & Yourdon 1991, Booch 1991). 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 (Lakos 1996).

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 (Lakos 1996, p.97). Martin uses the term *package design* (Martin 1996*b*). Earlier work in package design often does not give it an explicit name but uses terms such as *class category* (Booch 1991), *clusters* (Meyer 1995), *subject areas* (Coad & Yourdon 1991), *domains* (Shlaer & Mellor 1992) and *subsystems* (Booch 1987) 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.

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' (Miller 1956) as the basis for this principle (Coad & Yourdon 1991, 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) (Lakos 1996, 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 (Meyer 1995, 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 (Lakos 1996, 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 (Lakos 1996, p.149-202).

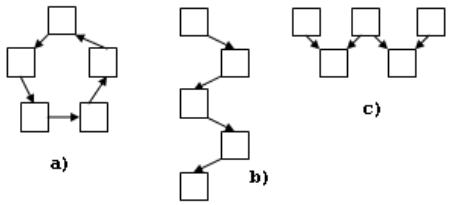

Figure 1: Cyclic, tall and flat PDGs

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.

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 (Lakos 1996,

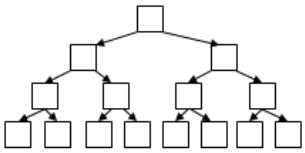

Figure 2: Balanced Binary Tree PDG

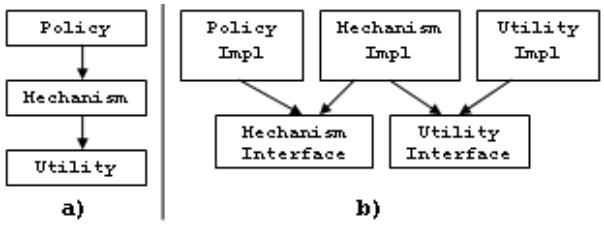

Figure 3: Flattening a tall PDG with the DIP

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.

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.

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.

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 (Szyperski 1998, p.3-4). 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 (Gaffney Jr & Cruickshank 1992).

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 enviroment 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 (Martin 1996*b*).

Packages are inherently related to code reuse because a class is not the fundamental unit of deployment (Martin 1996*b*, Lakos 1996) (Szyperski 1998, p.10). 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 (Martin 1996*b*), not individual source files each containing a single class.

In terms of package design, packages that are stand-alone (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 (Lakos 1996, 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.

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 (Martin 1996*a*). 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.

3.2 Testability

Testability is often defined as the ease at which software can be made to demonstrate faults through testing (Bass, Clements & Kazman 1998, 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) (Lakos 1996).

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 (Binder 1999, 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 (Thomas & Hunt 2002) 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' (Lakos 1996). 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 (Lakos 1996).

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 (Coad & Yourdon 1991, Booch 1991). 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 (Meyer 1995, Lakos 1996, Martin 1996*b*). 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 (Coad & Yourdon 1991), 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)).

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 $\mathcal{C}$, the relation DEPENDS-ON($\mathcal{C}$) is the set of classes that must be available in order to compile $\mathcal{C}$ (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 $\mathcal{C}$.java. Another way to think about it is that is the number of distinct types that are referred to by names that appear in $\mathcal{C}$.java. CRSS($\mathcal{C}$) is then the size of the set representing the *transitive closure* of the DEPENDS-ON relation as applied to $\mathcal{C}$.

Figure 4: Relationship between PDG structure and CRSS distributions

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 (Booch 1991, 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 (Lakos 1996, Martin 1996*b*).

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 ($\mathcal{L}$) have CRSS values of 99 or fewer, and the other 500 classes ($\mathcal{R}$) have CRSS values of 600–699. The classes in $\mathcal{L}$ 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 $\mathcal{R}$. 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 $\mathcal{L}$, so the package containing A must depend on at least 2 other packages involving classes in $\mathcal{R}$. Since this is true for every class in $\mathcal{R}$, every package involving classes from $\mathcal{R}$ must transitively depend on at least 2 other packages involving classes from $\mathcal{R}$. 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 Results

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 (Grothoff, Palsberg & Vitek 2001). 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.

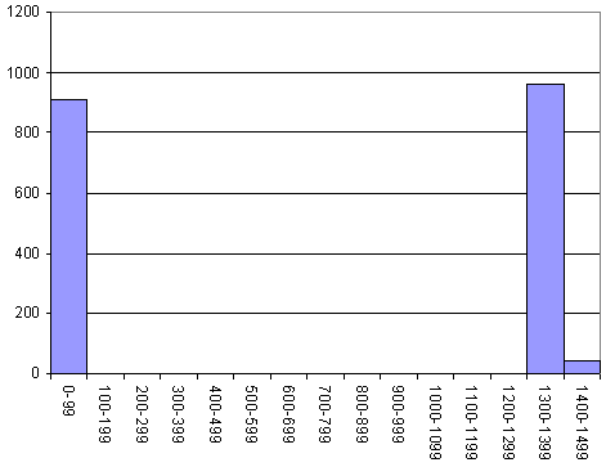

Figure 6: Azureus CRSS Distribution

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.

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.

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 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 'CRSS-refact' 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. Contin-

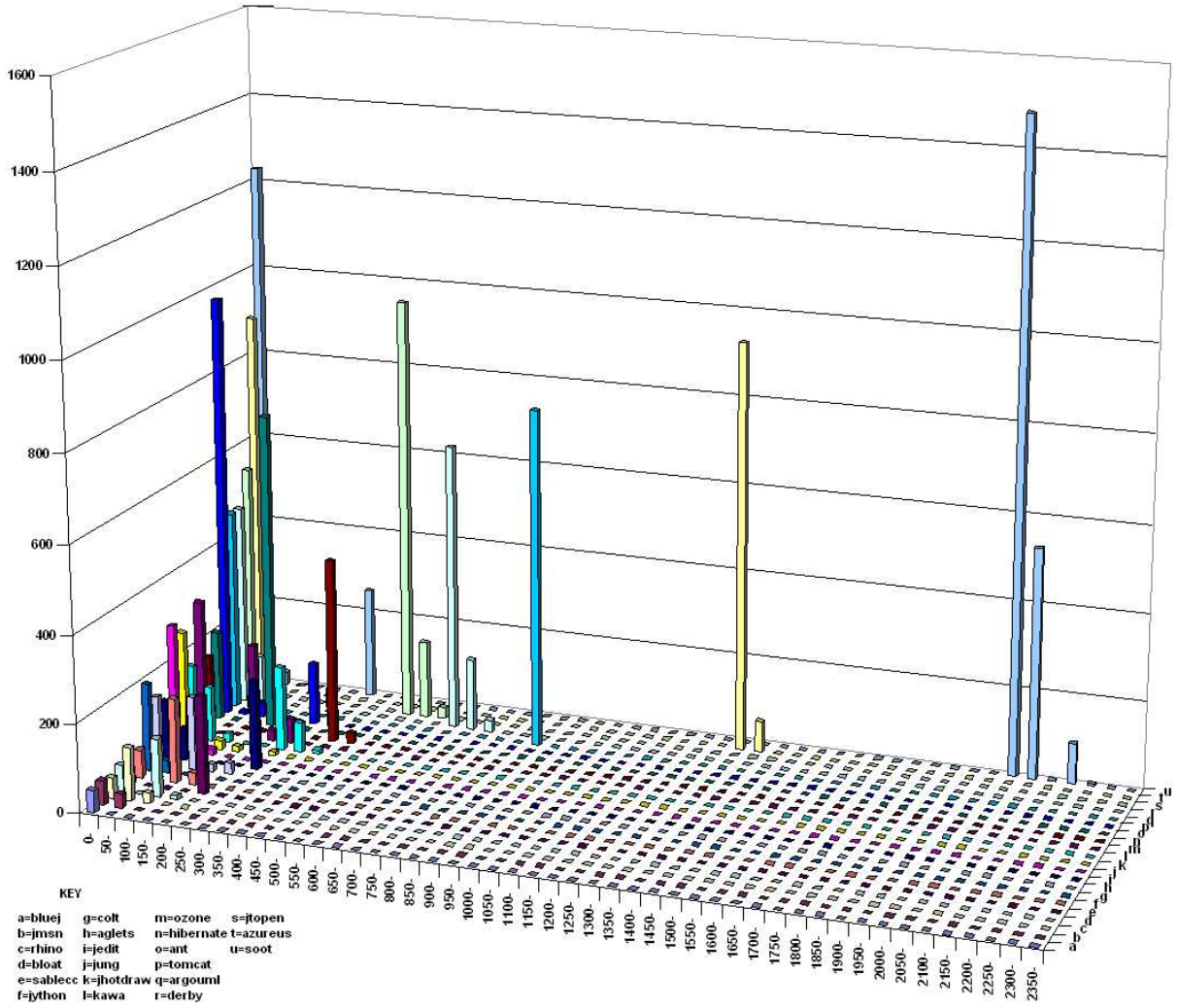

Figure 5: Software Corpus CRSS Distributions

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

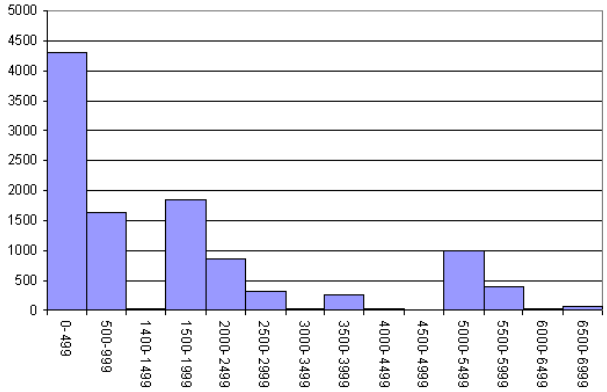

Figure 7: Eclipse CRSS Distribution

uing 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.

5.3 Eclipse

We also collected CRSS values for classes in the open-source IDE Eclipse, version 3.0.2 for Windows[1]. 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 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.

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 `OpenRes- ourceDialog` 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).

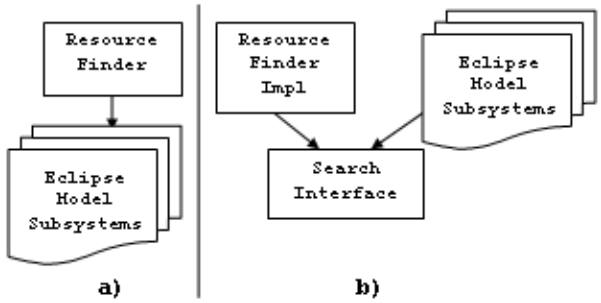

Figure 8: Resource Finder Subsystem

6 Refactoring

6.1 Strategy

We have developed a refactoring strategy based on the *Dependency Inversion Principle (DIP)* (Martin 1996*a*) 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

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

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* (Gamma, Helm, Johnson & Vlissides 1995, p.130).

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

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 (Gamma et al. 1995). 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.

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.

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

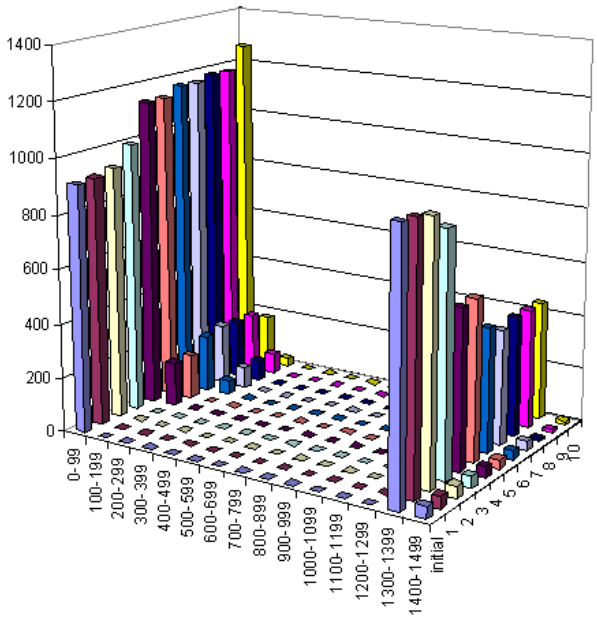

Figure 9: Refactoring Azureus

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`, `COConfig -urationManager`, `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.

7 Related Work

The work in this paper extends a prior work (Melton & Tempero 2006*b*) and is also related to work we have done looking at dependency cycles among classes in Java software (Melton & Tempero 2006*a*). Here we review other work that has been done in metrics for package design. Hautus (Hautus 2002), Lakos (Lakos 1996) and Ducasse et al. (Ducasse, Lanza & Ponisio 2005) have each produced literature on this topic.

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" (Hautus 2002). 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 packages 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 partioned 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 (Martin 1996*b*). 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.

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 (Lakos 1996, 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 (Lakos 1996, 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'.

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 (Ducasse et al. 2005). 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.

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.

References

Bass, L., Clements, P. & Kazman, R. (1998), *Software Architecture in Practice*, Addison-Wesley Longman, Reading, MA.

Binder, R. V. (1999), *Testing object-oriented systems: models, patterns, and tools*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Booch, G. (1987), *Software components with Ada: Structures, tools, and subsystems*, Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA.

Booch, G. (1991), *Object oriented design with applications*, Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA.

Coad, P. & Yourdon, E. (1991), *Object-oriented analysis (2nd ed.)*, Yourdon Press, Upper Saddle River, NJ, USA.

Ducasse, S., Lanza, M. & Ponisio, L. (2005), Butterflies: A visual approach to characterize packages, *in* 'Proceedings of 11th IEEE International Software Metrics Symposium (METRICS 2005)'.

Gaffney Jr, J. E. & Cruickshank, R. D. (1992), A general economics model of software reuse, *in* 'ICSE '92: Proceedings of the 14th international conference on Software engineering', ACM Press, New York, NY, USA, pp. 327–337.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995), *Design patterns: elements of reusable object-oriented software*, Addison-Wesley Longman Publishing Co. Inc., Boston, MA, USA.

Grothoff, C., Palsberg, J. & Vitek, J. (2001), Encapsulating objects with confined types, *in* 'OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications', ACM Press, New York, NY, USA, pp. 241–255.

Hautus, E. (2002), Improving Java software through package structure analysis, *in* 'The 6th IASTED International Conference Software Engineering and Applications'.

IEE (1990), 'IEEE standard glossary of software engineering terminology'. IEEE Std 610.12-1990.

Lakos, J. (1996), *Large-scale C++ software design*, Addison Wesley Longman Publishing Co. Inc., Redwood City, CA, USA.

Martin, R. C. (1996*a*), 'The Dependency Inversion Principle', *C++ Report* **8**(6), 61–66.

Martin, R. C. (1996*b*), 'Granularity', *C++ Report* **8**(10), 57–62.

Melton, H. & Tempero, E. (2006*a*), An empirical study of cycles among classes in Java, *in* 'Tech. Report UoA-SE-2006-1', Department of Computer Science, University of Auckland.

Melton, H. & Tempero, E. (2006*b*), Identifying refactoring opportunities by identifying dependency cycles, *in* 'CRPITS'48: Proceedings of the 48th conference on Computer science 2006', Australian Computer Society, Inc., pp. 35–41.

Meyer, B. (1995), *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., Upper Saddle River, NJ, USA.

Miller, G. A. (1956), 'The magical number seven, plus or minus two: Some limits on our capacity for processing information', *The Psychological Review* **63**, 81–97.
URL: `http://www.well.com/user/smalin/miller.html`

Shlaer, S. & Mellor, S. J. (1992), *Object lifecycles: modeling the world in states*, Yourdon Press, Upper Saddle River, NJ, USA.

Szyperski, C. (1998), *Component software: beyond object-oriented programming*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

Thomas, D. & Hunt, A. (2002), 'Mock objects.', *IEEE Software* **19**(3), 22–24.

Wirfs-Brock, R., Wilkerson, B. & Wiener, L. (1990), *Designing object-oriented software*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA.