

JooJ: Real-time Support for Avoiding Cyclic Dependencies

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

Abstract

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 feasibility our 'real-time checking' approach via an Eclipse plugin we have built called JooJ.

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* (Sun 1999)). Correctness guidelines aim to help programmers avoid common or subtle errors (e.g., "Class overrides equals() without overriding hashCode()" (Bloch 2001)). Design guidelines aim to help programmers make decisions about the internal structure of a system (e.g., Riel's *Object-Oriented Design Heuristics* (Riel 1996) and *Design Patterns* (Gamma, Helm, Johnson & Vlissides 1995)).

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 (Pressman 2001, 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 (Briand, Daly & Wust 1998) 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 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.

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.

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 (Parnas 1978). 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 (Lakos 1996, p.185), and that cycles among packages inhibit development, marketing, usability, production and reliability (Lakos 1996, 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” (Riel 1996, 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” (Booch 1995). 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* (Riel 1996, 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.

2.2 Definition of Cycle

We have adapted Lakos’ work with cycles in C++ to Java (Melton & Tempero 2006). 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 (Melton & Tempero 2006). 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.

2.3 Prevalence of Cycles

Our main motivation for providing tool support to avoid dependency cycles comes from an empirical study we performed (Melton & Tempero 2006). 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 (Hashim, Schmidt & Ramakrishnan 2005, Briand, Labiche & Wang 2003) (Binder 1999, 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 (Lakos 1996) and others (Binder 1999, 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 (Binder 1999, p.983-984).

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* (Lakos 1996, 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 cy-

cle growth and throwing systems away are phenomena we have reported (Melton & Tempero 2006).

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” (Foote & Yoder 2000). 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).

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 (Pressman 2001, p.214-215).

Pressman (Pressman 2001, 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 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.

2.6 Applicability

It has been claimed that avoiding dependency cycles among modules is most applicable to large-scale software systems (Martin 1996, Lakos 1996). 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” (Lakos 1996, 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 (Melton & Tempero 2006) 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.

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 shrinkwrapped 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 . . . (Fayad, Laitinen & Ward 2000).

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 (Booch 1996, 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 (Melton & Tempero 2006). 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.

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.

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.

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

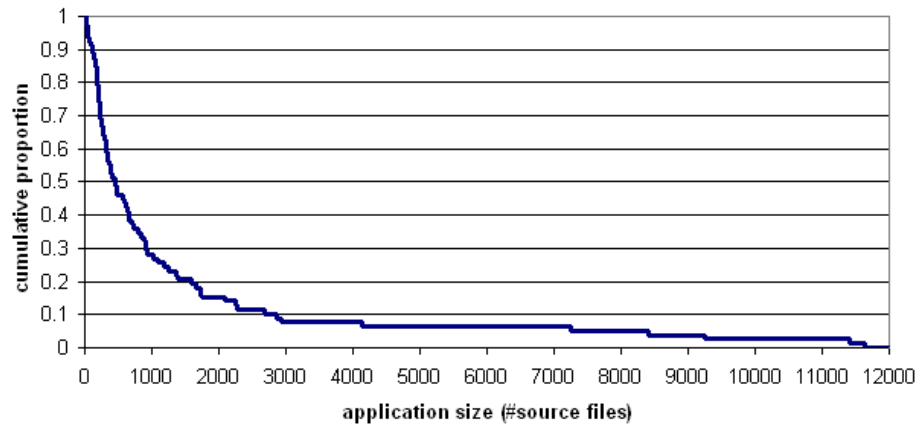


Figure 1: Distribution of application size across 78 Java applications

```

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 3: Refactoring suggestions for style and correctness violations in Eclipse

first of these annotations are the *squiggles*¹ on lines 10, 14 and 15. These squiggles 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 squiggles 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 squiggles are continuously recomputed as text is typed into the Java editor.

Another annotation evident in Figure 2 is the appearance of lightbulb icons in the left margin on the lines where squiggles 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 de-

pendency 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) (Lakos 1996, 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.

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. (Cormen, Leiserson & Rivest 1990, 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 (Eades, Lin & Smyth 1993). 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).

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 (Melton & Tempero 2006). As dis-

¹This is the term used for these wavy, coloured underlines in the Eclipse help documents

cussed 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 JUNG². 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.

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 (Melton & Tempero 2006), 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 super-types. 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.

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 squiggles 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 a 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.

5 Evaluation

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

²<http://jung.sourceforge.net/>

develop even large projects comprising thousands source files (Arthorne & Laffra 2004, 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.

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.

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 squiggles 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, jedi or jung.

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 ‘mas-

ter 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.

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 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 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 (Arthorne & Laffra 2004, 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

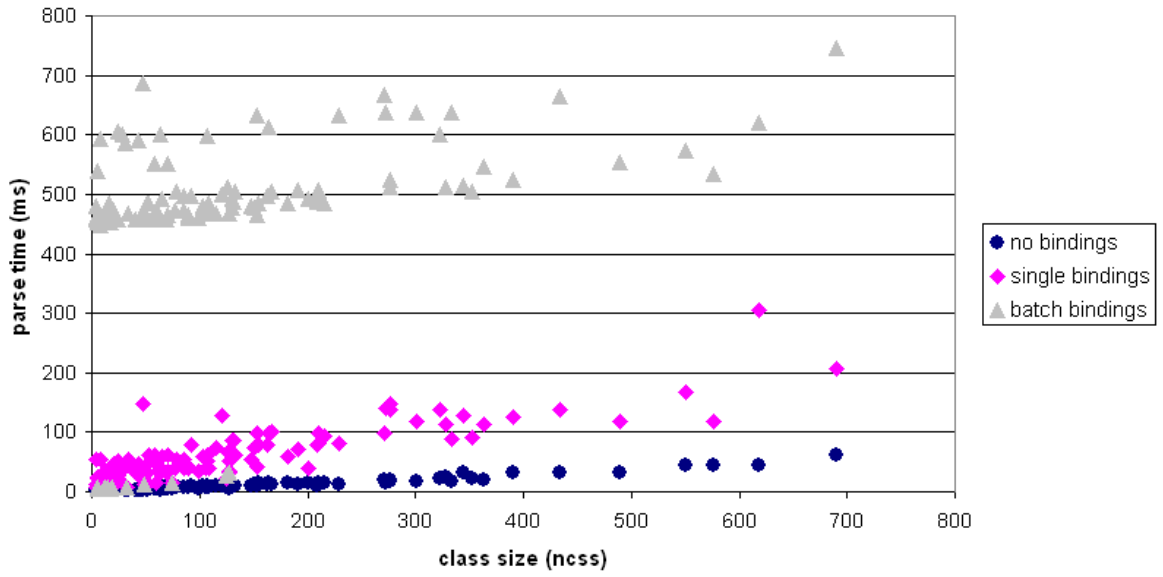


Figure 4: Time to create ASTs for a sample of . java files

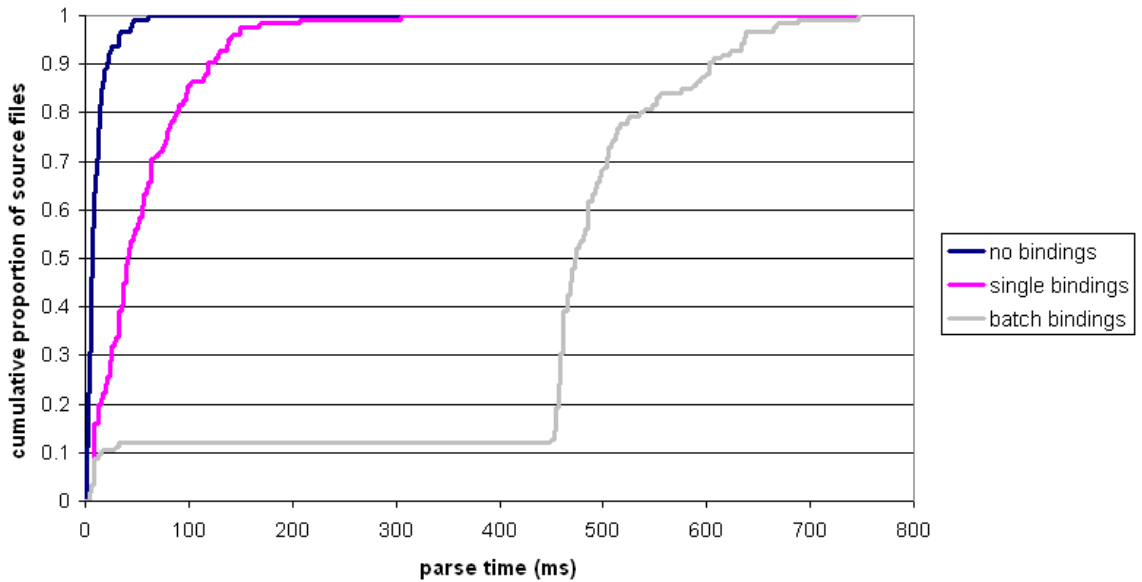


Figure 5: Distribution of AST creation times with and without bindings

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

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.

6 Related Work

6.1 ByeCycle

ByeCycle³ 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* (Melton & Tempero 2006) (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.

6.2 Design Level Tools

There are several tools available that do ‘real-time’ checking of a UML design. ArgoUML (Robbins, Hilbert & Redmiles 1998) 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.

Egyed (Egyed 2006) has also produced a tool *UMLAnalyser* 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.

³<http://byecycle.sourceforge.net/>

6.3 Batch-style Cycle Tools

There are a plethora of other batch-style cycle checking tools for Java. Classycle⁴ 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.

JDepend⁵ 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 (Melton & Tempero 2007).

Lattix LDM (Sangal, Jordan, Sinha & Jackson 2005) 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 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 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.

References

- Arthorne, J. & Laffra, C. (2004), *Official Eclipse 3.0 Faq (Eclipse Series)*, Addison-Wesley Professional.
- Binder, R. V. (1999), *Testing object-oriented systems: models, patterns, and tools*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

⁴<http://classycle.sourceforge.net/>

⁵<http://www.clarkware.com/software/JDepend.html>

- Bloch, J. (2001), *Effective Java programming language guide*, Sun Microsystems, Inc., Mountain View, CA, USA.
- Booch, G. (1995), *Object solutions: managing the object-oriented project*, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.
- Booch, G. (1996), *Best of Booch: Designing Strategies for Object Technology*, SIGS Books.
- Briand, L. C., Daly, J. W. & Wust, J. (1998), 'A unified framework for cohesion measurement in object-oriented systems', *Empirical Softw. Engg.* **3**(1), 65–117.
- Briand, L. C., Labiche, Y. & Wang, Y. (2003), 'An investigation of graph-based class integration test order strategies', *IEEE Trans. Softw. Eng.* **29**(7), 594–607.
- Cormen, T. T., Leiserson, C. E. & Rivest, R. L. (1990), *Introduction to algorithms*, MIT Press, Cambridge, MA, USA.
- Eades, P., Lin, X. & Smyth, W. F. (1993), 'A fast and effective heuristic for the feedback arc set problem', *Inf. Process. Lett.* **47**(6), 319–323.
- Egyed, A. (2006), Instant consistency checking for the uml, in 'ICSE '06: Proceeding of the 28th International Conference on Software Engineering', ACM Press, New York, NY, USA, pp. 381–390.
- Fayad, M. E., Laitinen, M. & Ward, R. P. (2000), 'Thinking objectively: software engineering in the small', *Commun. ACM* **43**(3), 115–118.
- Foote, B. & Yoder, J. W. (2000), Big ball of mud, in N. Harrison, B. Foote & H. Rohnert, eds, 'Pattern Languages of Program Design', Vol. 4, Addison Wesley, pp. 654–692.
- 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.
- Hashim, N. L., Schmidt, H. W. & Ramakrishnan, S. (2005), Test order for class-based integration testing of Java applications, in 'QSIC '05: Proceedings of the Fifth International Conference on Quality Software', IEEE Computer Society, Washington, DC, USA, pp. 11–18.
- Hautus, E. (2002), Improving Java software through package structure analysis, in 'The 6th IASTED International Conference Software Engineering and Applications'.
- Lakos, J. (1996), *Large-scale C++ software design*, Addison Wesley Longman Publishing Co. Inc., Redwood City, CA, USA.
- Martin, R. C. (1996), 'Granularity', *C++ Report* **8**(10), 57–62.
- Melton, H. & Tempero, E. (2006), 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. (2007), The CRSS metric for package design quality, in 'Thirtieth Australasian Computer Science Conference (ACSC2007)', Australian Computer Society, Inc.
- Parnas, D. L. (1978), Designing software for ease of extension and contraction, in 'ICSE '78: Proceedings of the 3rd international conference on Software engineering', IEEE Press, Piscataway, NJ, USA, pp. 264–277.
- Pressman, R. S. (2001), *Software engineering: a practitioner's approach (5th ed.)*, McGraw-Hill, Inc., New York, NY, USA.
- Riel, A. J. (1996), *Object-Oriented Design Heuristics*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- Robbins, J. E., Hilbert, D. M. & Redmiles, D. F. (1998), Software architecture critics in argo, in 'TUI '98: Proceedings of the 3rd international conference on Intelligent user interfaces', ACM Press, New York, NY, USA, pp. 141–144.
- Sangal, N., Jordan, E., Sinha, V. & Jackson, D. (2005), Using dependency models to manage complex software architecture, in 'OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications', ACM Press, New York, NY, USA, pp. 167–176.
- Skiena, S. S. (1998), *The algorithm design manual*, Springer-Verlag New York, Inc., New York, NY, USA.
- Sun (1999), 'Code conventions for the Java programming language'.
URL: <http://java.sun.com/docs/codeconv/>