Quantitatively Measuring Object-Oriented Couplings

JEFF OFFUTT offutt@gmu.edu
AYNUR ABDURAZIK aynur.abdurazik@gmail.com
Information and Software Engineering Department, George Mason University, Fairfax, VA 22030, USA

STEPHEN R. SCHACH srs@vuse.vanderbilt.edu
Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, TN 37235, USA

Abstract. One key to several quality factors of software is the way components are connected. Software coupling can be used to estimate a number of quality factors, including maintainability, complexity, and reliability. Object-oriented languages are designed to reduce the number of dependencies among classes, which encourages separation of concerns and should reduce the amount of coupling. At the same time, the object-oriented language features change the way the connections are made, how they must be analyzed, and how they are measured. This paper discusses software couplings based on object-oriented relationships between classes, specifically focusing on types of couplings that are not available until after the implementation is completed, and presents a static analysis tool that measures couplings among classes in Java packages. Data from evaluating the tool on several open-source projects are provided. The coupling measurement is based on source code, which has the advantage of being quantitative and more precise than previous measures, but the disadvantage of not being available before implementation, and thus not useful for some predictive efforts.

Keywords: object-oriented programming, quality analysis and evaluation, analysis, complexity measures, maintainability, test execution

1 Introduction

Software coupling has long been used to evaluate software. Specific ways to measure coupling have long been known for procedural software, but most measures for object-oriented software have been based on design artifacts such as class diagrams. This paper presents research that statically analyzes source code to measure coupling in object-oriented software. Coupling is a metric for software, thus the following paragraphs set this work in context and describe metrics terms as used in this paper.

Coupling information can be obtained from design documents before implementation, or from the source after implementation. Both methods have advantages and disadvantages. Obtaining coupling information before implementation allows the information and measurements to be used in project planning, implementation, and test preparation. Coupling information obtained from the implementation can incorporate decisions and reflect changes made during the implementation, giving more information and allowing measurements to be more precise. This research project addresses post-implementation coupling measurement.

This paper uses metrics and complexity terms as defined by IEEE standards. The IEEE defines a *quality factor* to be an attribute of a program that affects its quality [37, 38]. These were called *external attributes* by Briand et al. [15]. Examples include maintainability, reliability, and complexity. Factors are often qualities on which we wish to base a decision. Some can be measured directly and some cannot. For example, Schneidewind defines a factor as "a type of metric that provides a direct measure of a software quality" [60], implying that he assumes factors can be measured. A *quality metric* is an attribute that can be measured by one or more functions whose inputs are data obtained from software artifacts (such as design documents, program source, or requirements documents). These were called *internal attributes* by Briand et al. [15]. A *measurement* is a specific function for computing values for a metric. Metrics often represent information that is not directly relevant to the developers (such as how many executable lines are in a program), but are used to *estimate* factors that are directly relevant to developers (such as how hard it is to change the program). A metric is said to be *validated* for a factor if it has been statistically shown to accurately estimate the factor [60]. Only measurable factors can be validated–we cannot show statistically what we cannot measure.

For example, the metric *cyclomatic complexity* uses the measurement $M = e - n + 2p$ (where e is the number of edges in the control flow graph, n is the number of nodes, and p is the number of connected components) to estimate the factor *complexity* of software [52] and as a test heuristic. Validating a metric for a factor is impossible for factors that cannot be measured and extremely difficult for those that can [60]. Thus factors have rarely been validated. One of the more commonly used metrics is software coupling, which has been validated for development time and error rate [28], number of likely faults [7, 61], and aspects of maintenance such as impact analysis [14], the cost of maintenance [39], and the cost of making changes [25].

Coupling is a metric, or internal attribute, that was first introduced by Stevens, Myers and Constantine [53, 65]. Coupling measures information flow among program components [21]. More coupling means that modules are more related and harder to change, understand and repair. Empirical evidence has been published that supports this theory for structured development [61, 67].

Coupling has often been estimated in qualitative terms, or if it is measured at all, the measurements are based on information flow metrics [44, 34]. Despite being widely used as a metric, it is difficult to measure coupling in ways that are quantitative and precise. Most measurements for coupling are qualitative in nature [25, 26, 31, 39, 61]. Some quantitative measurements are based on procedural (non-object-oriented) software [25, 28, 32, 33, 54, 61], and do not apply to object-oriented software.

The concept of coupling has been adapted to object-oriented software by Coad and Yourdon [18, 19] and numerous metrics for object-oriented software have been defined. Object-oriented languages introduce several new mechanisms for coupling classes and methods, thus defining coupling is more complicated. As pointed out by Briand et al. [15], the field does not have standard terminology and formalisms for expressing metrics, so metrics are not fully operationally defined and are thus ambiguous. This has made it difficult to relate different coupling metrics and hard to understand how to use different coupling metrics together. The unified framework defined in the paper by Briand et al. gives fully operational metrics and provides a context in which new metrics and ways to measure them can be defined. This paper uses their framework, focusing on statically measuring coupling in the implementation source.

Most object-oriented coupling research has focused on usage dependencies among classes. This information can be derived from static analysis of design documents, static analysis of implementation source, or dynamic (run-time) analysis of implementation source. Dynamic issues have been addressed in the context of testing [3, 4, 56], but work on measurement has started only recently [5]. This paper focuses on static analysis of implementation source with some discussions of extensions to dynamic analysis.

Most previous object-oriented coupling measures are based on high level design models such as statecharts and collaboration diagrams (now called "communication diagrams"[1]).

[12, 15, 23, 69]. Deriving coupling metrics from design information allows the metrics to be available before the program is written, thereby allowing the information to be used for predictive purposes. On the other hand, deriving coupling metrics from static analysis of the source code allows the measurement to be more precise and to reflect accurately the actual implementation. Although code-based information cannot be determined as early as a post-design measurement, coupling can be useful during maintenance, during test generation, and to evaluate how much the software deviates from the design.

The tool presented in this paper is completely static; that is, it analyzes but does not run the software. Arisholm [5] proposed the use of dynamic coupling measures to quantify the flow of messages between objects at runtime. Although dynamic coupling measures can capture some dependencies that static coupling measures overlook, the dependencies they capture are not complete because they depend on the specific executions used. Dynamic measurement could be integrated with the measurement in this paper.

This paper looks specifically at object-oriented software, and uses the framework by Briand et al. [15] to identify language characteristics that allow software components to be connected in ways that non-object-oriented languages could not. An original goal of object-oriented design was to improve maintainability. Conversely, some aspects of object-oriented language features have a negative

[1] Online at http://www.uml.org/

impact on maintenance. One difficulty has been called the "complex interdependency problem" [46]. Properly designed object-oriented software should have simple classes and methods, but often exhibits complex relationships among the classes. These relationships include inheritance, aggregation, association, class nesting, dynamic object creation, member function invocation, polymorphism, and dynamic binding relationships. These relationships introduce dependencies on other classes that have not yet been fully analyzed and understood. For example, the inheritance relationship implies that a derived class reuses data and function members of a base class and hence depends on the base class.

This research uses the complex interdependencies in object-oriented languages to define coupling types. The focus of this research is on inter-class couplings rather than intra-class couplings. Intra-class couplings are the same for both object-oriented and procedural software, have been well studied, and are out of scope of this paper. The inter-class couplings are then used to define a quantitative measurement for coupling in object-oriented software that is precise and algorithmically computable. This measurement is based on static analysis of the source implementation, which is more expensive than analyzing design artifacts and occurs later, but yields more precise results. We have fully implemented this measurement in an analysis tool for Java programs, Java Coupling Analysis Tool (JCAT), and data from applying it to several large open-source projects are included.

This paper proceeds by providing background on software coupling in Section 2, then presents our particular model of object-oriented coupling in Section 3. Applying this theory to specific programming languages presents many practical difficulties. Section 4 presents innovative solutions to these problems by defining how the measure is computed in Java. Section 5 describes an automated tool for measuring coupling. Data from applying this tool to open-source software are given in Section 6, Section 7 discusses those data, and Section 8 provides conclusions and discusses future work.

2 Software Coupling Background and Related Work

Research into how software components are coupled dates back to the late 1960s. In 1974, a paper by Stevens, Myers and Constantine [65] and a book by Myers [53] refined the concept of coupling by presenting well-defined, though informal and qualitative, levels of coupling. Because their levels were neither precise nor prescriptive definitions, coupling could be determined only by hand, leaving room for subjective interpretations of the levels. Other researchers [36, 44, 61, 67] have used coupling types or similar measures to evaluate the complexity of software design and relate this complexity to the number of software faults. El Emam et al. [24] established a correlation for predicting faulty classes in object-oriented software that was based on a non-quantitative evaluation of coupling. Fenton and Melton [26] developed a measurement theory that provides a basis for defining software complexity and used hand-derived coupling measures to demonstrate their theory. They enhanced previous work in coupling by incorporating the number of interconnections between modules into the measure and by considering the effects on coupling of return values and reference parameters as well as input parameters.

Historically, module coupling was used as an imprecise measure of software complexity in procedural software. In 1991, Jalote said that coupling is "an abstract concept and is as yet not quantifiable" [40]. Offutt and Harrold [54] extended previous work to reflect type abstraction, and quantified coupling of procedural software by developing a general software metric system that allows coupling to be automatically measured. They offered precise definitions of the coupling levels so that they can be determined algorithmically, incorporated the notion of direction into the coupling levels, and accounted for different types of non-local variables as found in modular, but non-object-oriented, programming languages.

Jin and Offutt later used couplings as a basis for integration testing of procedural software [42, 43]. Previous researchers in applying coupling to design quality had a very fine grained notion of coupling, with up to twelve types, but it was found that many were equivalent for testing purposes, so Jin and Offutt included only four types. These were used to define formal integration testing criteria that required testing to proceed through couplings from data definitions to data uses.

Several researchers have defined object-oriented coupling types. In 1992, Chidamber and Kemerer [17] defined numerous metrics for object-oriented software design, including some related to coupling. In 1994, Eder et al. [22] identified three different types of relationships: interaction relationships between methods, component relationships between classes, and inheritance between classes. These three relationships were used to classify different dimensions of couplings according to strength of coupling. In 1995, Hitz and Montazeri [35] presented two types of coupling: object level coupling, determined by the state of an object; and class level coupling, determined by the state of an objects implementation. They also proposed different coupling strengths. In 1997, Briand et al. [11] defined coupling as interactions between classes. The strength of the coupling is determined by the type of the interaction, the relationship between the classes, and the direction of the interaction (its "locus of impact").

This paper defines a coupling model, partially based on previous models as summarized above, of different levels of coupling. Measure are defined on the implementation source. Part of the challenge is in making the definitions precise enough to be applied to static analysis of programs. The conceptual definitions of the coupling types have to be defined for specific object-oriented language features. The next section describes our object-oriented coupling model, fitting it into the context of the framework by Briand et al. [15], and defining the connections in ways that can be used for a quantitative measure.

3 A Model of Object-Oriented Coupling

Most of the material in this section applies generally to any object-oriented language, but the tool developed for this research is specific to Java, and a few concepts may need to be adapted to other object-oriented languages. Most of the couplings defined here are the same or similar to those in other papers; the major contributions are in Section 4, which presents techniques for applying the theory and concepts of coupling to Java source, and Section 5, which describes the static analysis tool that supports this research. Some of the previous types of couplings have very little effect on coupling at the source level, so are not used separately. For example, how a parameter is referenced inside a method is not important, so the old types of "control parameter coupling" and "data parameter coupling" are merged in one type, "parameter coupling." The types of coupling used in this research are:

1. *Parameter coupling* is any method call, possibly including parameters.
2. *External/file coupling* refers to classes that access the same external medium, including external files.
3. *Inheritance coupling* occurs when one class is a subclass or descendant of another. The coupling is made through inherited but not re-defined data members of a superclass by its subclass. Figure 1 shows a class **B** and its subclass **A**. **B** defines a data member **b**, and **A** inherits **b** from **B**. Accordingly, classes **B** and **A** are inheritance coupled through **b**. (A number of details for how to discern inheritance coupling in special case situations are discussed in Section 4.)
4. *Global coupling* (which also has been called *common*, *shared*, and *non-local*) refers to variables that are defined in one class and used in others. Modern object-oriented languages such as C++ and Java have several access specifiers as well as storage class `static`, all of which affect global coupling. Accordingly, this coupling type is expanded below.

Table 1 compares the coupling types that are used in this paper and in the papers by Eder [22], Hitz and Montazeri [35], and Briand et al. [11]. The terminology used in this paper is shown on the left, and the terms used in the other papers are in the appropriate cells. Hitz and Montazeri did not explicitly label the coupling types, so Table 1 simply shows an 'X' in the appropriate column. The *class-attribute* and *class-method* types from Briand et al.'s paper are not considered in this paper, so they are shown in last row in the table.

Some of the previous literature on coupling has discussed which types of couplings are "better" or "worse," based on qualitative analysis of the coupling types. For example, it is widely believed that global coupling introduces more possibilities for faults than other types of coupling, and students are usually taught to prefer parameter coupling over global coupling. However, it is hard to quantify these differences, and how much better one type of coupling is than another may depend on the purpose of the coupling. The analysis in this paper does not treat any one type of coupling as being better than another. Each type is measured and presented independently.

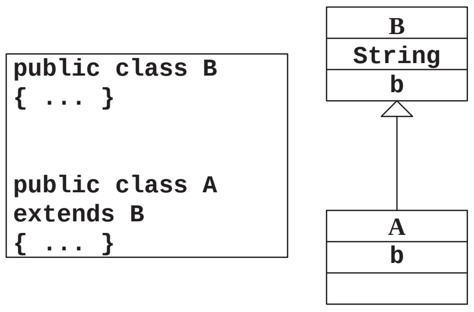
Table 1. Coupling Types Compared

This paper	**Eder [22]**	**Hitz & Montazeri [35]**	**Briand et al. [11]**
Parameter coupling	interaction control, stamp, data	X	method-method
External / File coupling	interaction external		
Inheritance coupling	inheritance	X	inheritance
Global coupling	interaction context, common		friendship (C++) (package in Java)

Not used		X	class-attribute
			class-method

Three out of four of the coupling types have *direction* (called *locus of impact* by Briand et al. [11]), which is used in the measurement and analysis. The coupling direction is considered to be in the direction of the *use*, that is, if class **P** uses **Q**, then **P** is *coupled to* **Q**. The rationale for this is that if a programmer changes **P**, there can be no effect on **Q**. However, changes to **Q** can effect **P**, possibly significantly.

For parameter coupling, the direction is assumed to be the direction of the call. That is, if **R** calls **S** with a parameter, then **R** uses **S** and the coupling is from **R** to **S**. If **S** returns a value, there is also coupling back to **R** (bi-directional). External coupling occurs when two classes, **A** and **B**, both access the same external device, say a printer. The coupling is considered to be between **A** and **B**, so the coupling is bi-directional. If **A** inherits from **B** as in Figure 1, then **A** is thought of as "using" elements of **B**, so the coupling direction is from **A** to **B**. Finally, global coupling follows the definition and use pattern; if **T** uses a variable **x** that is defined (given a value) in **U**, then the coupling is from **T** to **U**.



```
public class B
{ ... }


public class A
extends B
{ ... }
```

Fig. 1. Inheritance Coupling Example.

Java supports four distinct access levels for member variables and methods: `private`, `default` (also called `package`), `protected`, and `public`. Table 2 summarizes these access levels. A `private` member is available to only the class in which it is defined. If access is not specified, the access level defaults to `package`, which allows access to classes in the same package. A `protected` member is available to the class itself, classes in the same package, and all descendants including those outside of the package. A `public` member is available to any class in any inheritance hierarchy or package (the world).

Table 2. Java's Access Levels

Specifier	Same Class	Same package subclass or non-subclass	Different package subclass	Different package non-subclass
`private`	Y	N	N	N
`package`	Y	Y	N	N
`protected`	Y	Y	Y	N
`public`	Y	Y	Y	Y

Additionally, variables can be `static` (`class`) or `instance`. Static variables are shared among all objects of the class and instance variables have a unique copy for each object. The type of coupling varies by both access and variable definition. Specifically, static variables can be accessed whenever the class name is available, whereas instance variables can be accessed only through appropriate object references. Therefore, the coupling that arises from instance variables is less broad in scope than coupling from static variables.

The following list describes the resulting three pairs of combinations that make global coupling possible. Variables with private access specifiers cannot be accessed by any class other than the owner, so cannot appear in coupling.

1. `public static global coupling` is "true global" in the traditional sense, that is, the variables are shared by all objects in the software system. `public instance global coupling`, on the other hand, is more limited. When two or more methods have access to the same *instance* of an object reference, then they share access to the object. Accordingly, the object is shared by some, but not all methods in the system.
2. `package static global coupling` refers to classes in the same package that can share variables defined in any class in the package. `package instance global coupling` is more limited. When two or more methods in classes in the package have access to the same *instance* of an object reference, then they share access to the object. The methods could be in the same class or different classes. Accordingly, the object is shared by some, but not all methods in the package.
3. `protected global coupling` is the same as package coupling except access is extended to subclasses that are not in the same package. `protected static global coupling` refers to classes in the same package or subclasses that can share variables defined in any class in the package. `protected instance global coupling` describes methods inside the package or sub-

class that have access to the same object reference and that refer to the same variable.

Different strategies are required to identify each coupling type and subtype. However, once identified, all the global coupling types can be considered identically. That is, the distinctions are at the definition and implementation level, not at the conceptual level. It is likely, however, that the different global coupling types will impact maintenance and reliability differently.

4 Analyzing Source Code to Find Couplings

Extracting coupling information from the implementation is very different from extracting coupling information from design artifacts. Obtaining coupling information before implementation allows the information and measurements to be used in project planning, implementation, and test preparation. Coupling information obtained from the implementation can incorporate decisions and reflect changes made during the implementation, giving more information and allowing measurements to be more precise. Most researchers have thought about coupling conceptually and at design and modeling levels. The definitions that exist for coupling, therefore, tend to be fairly generic in terms of implementation. Much as design decisions have to be refined when programmers implement the design in specific language features, the coupling definitions need to be refined to accommodate the exact behavior of the specific language features. If we tried to define coupling to accommodate all these behaviors, we would probably start to lose sight of the forest for the trees, and also need to define coupling separately for each language. Thus, this section *refines* the generic coupling definitions to apply to specific language features.

Part of the challenge is in making the definitions precise enough to be applied to static analysis of programs. As an example, whether a programmer uses another class's identifier as a return value is entirely an implementation decision, and whether that constitutes coupling (and of what type) is something that would not be articulated in coupling definitions or the designs. The individual discussions in this section represent in-depth analysis of coupling types and language features. Numerous decisions had to be made, and these decisions are documented in detail by explaining how to extract couplings from program source code.

By analyzing the source code, we are able to use more information and compute a more precise measurement of coupling. In purely numeric measures, more precision usually equates to more digits of accuracy. With the coupling measure, the precision available from looking at the source code allows for a more detailed analysis, allowing us to find differences in coupling that are subtle or impossible to measure from design documents. For readers who are already intimately familiar with object-oriented programming, much of this analysis will be familiar. We include sufficient detail for our analysis to be repeatable and fully computable.

The four coupling types from Section 3 are determined by information that is usually not available at the design level, so must be computed from the program source code. Doing so, however, is difficult because some of the relationships appear only implicitly. The information must be extracted from the software by an analysis tool. Programming languages have subtle complexities that make finding coupling information more difficult than might be expected. Accordingly, the theoretical ideas in Section 3 must be *refined* to a more concrete level. Some calls are implicit instead of explicit, there are several types of global variables and uses, and the effect of inheritance with regard to coupling is not obvious. This subsection discusses the subtle decisions that were made to determine coupling, some that are generic, and others that are influenced by or unique to the language (in this case, Java). Examples in this section are taken from actual open-source software, mostly the parser generator tool ANTLR (ANother Tool for Language Recognition) [58].

4.1 Occurrences of parameter coupling

In Java, parameter coupling occurs through only method and constructor calls. This research refines the generic definition of parameter coupling to be *the occurrence of an invocation of a call to a method or constructor through an object or class.*

Java allows two explicit types of method calls, instance and static, and one implicit type, through a constructor. If a method in class **A** explicitly calls method **m()** in class **B** through an object instance (**b.m()**), this represents parameter coupling between **A** and **B**. An explicit static call occurs when class **A** calls a public static method **m()** in class **B** (that is, **B.m()**). An implicit constructor call is made when a variable of type **B** is defined and instantiated in class **A**, for example, **B b = new B()**. All three of these types are considered to be parameter coupling in this research, even if no actual parameter value is passed in and no value is returned.

The three types of parameter coupling are summarized as:

1. **B b = new B();** // implicit, constructor
2. **b.m();** // explicit, through an object reference
3. **B.m();** // explicit, static

As said in Section 1, the primary focus of this research is on parameter couplings between different classes (inter-class) as opposed to couplings between methods in the same class (intra-class). The effects of intra-class couplings are very different from the effects of inter-class couplings. Intra-class coupling has no direct impact on the external system, although it can have an indirect impact. If the class is viewed as a black box, then intra-class coupling is invisible. Therefore, intra-class coupling is used for entirely different problems and out of scope of this project. Inter-class parameter coupling has the potential to propagate problems from within one class to other classes, especially during maintenance and reuse.

4.2 Variations of global, inheritance, and external coupling

Global coupling is a kind of inter-class coupling that refers to the coupling that takes place through variables that are defined in one class and used in others. These variables will typically have `public` or `protected/package` access specifiers. `public` variables represent a traditional, or *true global coupling* as described in Section 3, if the variable is static, otherwise it is a *global coupling* with an object reference. All of these variations must be detected.

Inheritance coupling refers to the coupling that is related to the inheritance between pairs of classes. The coupling takes place through attributes and methods that are inherited and used by a subclass but that are not re-defined. If a subclass does not actually use anything from its superclass, or if it re-defines everything it uses, this is not considered to be inheritance coupling.

In Java, the inheritance relation is established through the keywords `extends` and `implements`. Therefore, an implementation can detect an inheritance coupling between two classes or interfaces if one class extends from another class or implements one or more interfaces, or if an interface extends from another interface.

Section 3 defined *external coupling* as *access to an external device by two or more classes.* In other words, external coupling happens when two classes share something that is outside the application that owns the classes. External resources can include files on a hard disk, printers, or other shared devices. The challenge in designing an algorithm to analyze coupling is to find out the unique interfaces between these resources and the application. Specifically, different classes or applications may use the same resource, but refer to them with different names. Binding to a physical device may be done at the OS level, not the program. This is necessary for symbolically linked files and devices with multiple names. If there is no unique interface, then all interfaces must be enumerated.

4.3 Discovering actual types and dynamic binding

When analyzing the software for coupling, the analyzer must first discover the types of each reference. This is simple for direct references to names. However, when a reference is made through an object reference (**o.b** or **o.m()**), we must first find the type of **o**. Inheritance and dynamic binding means that the type of **o** cannot be determined statically, because it can change during execution. Consider the following example:

```
throw new TokenStreamIOException (((CharStreamIOException)cse).io);
```

Class **CharStreamIOException** inherits from class **CharStreamException**, which in turn inherits from **Exception**. So the variable **cse** is of type **CharStreamException**, and the variable **io** is defined in the class **Exception**, which is part of the Java class library. This confusing case brings up the question: does the following example constitute global coupling?

```
catch (CharStreamException cse)
{
   if (cse instanceof CharStreamIOException)
   {
      throw new TokenStreamIOException
               (((CharStreamIOException)cse).io);
   }
   else
   {
      throw new TokenStreamException(cse.getMessage());
   }
}
```

To accurately decide if this example constitutes global coupling, the actual type of **cse** must be found. The actual type at the two `throw()` statements can be inferred because of the `instanceof` test. If **cse** is of actual type **CharStreamIOException**, then the first `throw()` is reached. Otherwise the second `throw()` is reached. That is, if **cse** is of actual type **CharStreamException** (the second `throw()`), this *is* a global coupling, but if **cse** is of actual type **CharStreamIOException** (the first `throw()`), the coupling is *not* global.

The actual type must first be determined to decide if a `throw()` statement represents global coupling. In general, the actual type cannot be determined statically, as shown by Alexander and Offutt [3, 4]. They defined the *polymorphic call set* to be the set of methods to which a specific method call could refer.

The analysis in this research uses a simpler analysis technique than computing the polymorphic call set. The actual type of a variable depends on the Java language statements `new` and `instanceof`, and any type casting . When a variable reference **a.b** is found, the type of **a** is found through one of several sources. It may be a variable that is defined in the current class, a parameter that is passed to the current method, a variable defined in an ancestor class, or a variable that is defined in the method. Type casting confuses the issue because the type of the object reference changes during execution of the statement. Consider the following example:

((A)((C)d.v()).g()).m

Here **A** and **C** are class types, **d** and **m** are variables that are instances of classes, and **v()** and **g()** are method calls. A call to method **v()** through the variable **d** ("**d.v()**") returns an instance of class **C** or one of **C**'s descendants. The return value is cast to be of type **C** ("**((C)d.v())**"). A call to method **g()** through this returned instance ("**((C)d.v()).g()**") returns an instance of class **A** or one of **A**'s descendants, which is cast to be of type **A** ("**(A)((C)d.v()).g()**"). Finally, **m** is a variable that is defined in class **A**. (Yes, this is terrible programming style, but unfortunately some programmers use it. We did *not* make this example up!)

With this kind of code, we assume that the final type, after all casting, is the actual type of the reference. Accordingly, any coupling is through the class that represents the actual type. This is a simple assumption, though the parsing is quite complicated.

4.4 Issues with nested coupling

Nested coupling occurs when a class references an attribute from its parent class, and a definition for the attribute appears in the parent's parent or another ancestor. The question is whether the class is coupled with its parent, or with the class where the attribute is defined. Figure 2 shows an example of nested coupling.

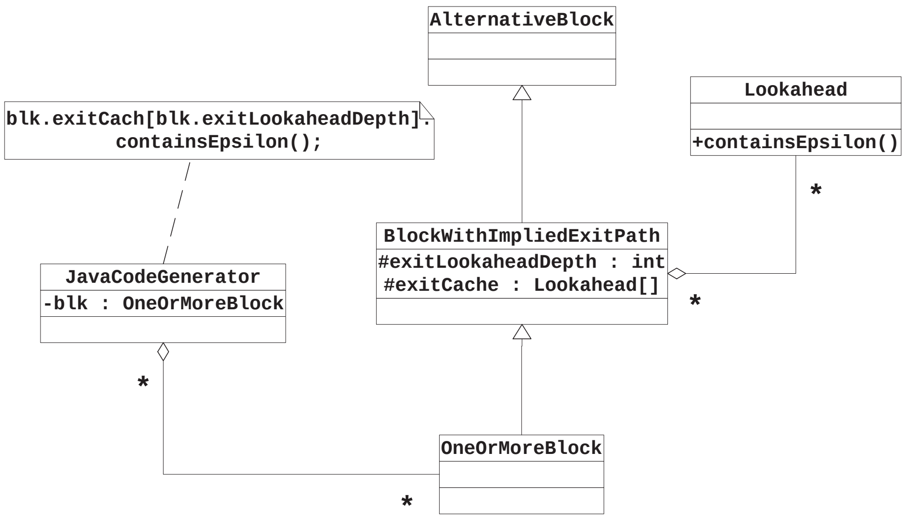


Fig. 2. Example of Nested Coupling in ANTLR

Class **JavaCodeGenerator** is coupled with class **OneOrMoreBlock**, because it is aggregated with **OneOrMoreBlock** and uses the **exitCache** and **exitLookaheadDepth** attributes. However, **exitCache** and **exitLookaheadDepth** are physically defined in the parent class of **OneOrMoreBlock**, **BlockWithImpliedExitPath**.

This leaves the question: is **JavaCodeGenerator** coupled with **OneOrMoreBlock**, or with **BlockWithImpliedExitPath**? Our analysis is based on the possible effects of changes to the program. If **BlockWithImpliedExitPath** is changed, that might affect **JavaCodeGenerator**, so the coupling should be with the class that actually defines the variables. Therefore, **JavaCodeGenerator**'s use of **exitCache** and **exitLookaheadDepth** creates a coupling with both **BlockWithImpliedExitPath** and **OneOrMoreBlock**.

Note that this analysis cannot be done without the source; the design models do not contain enough information. This is an example of why the coupling measurement described in this paper is considered to be more precise than design-based measurements.

5 A Tool for Measuring Coupling

We have built the Java Code Analysis Tool (JCAT) to analyze the structure and components of Java source-code packages and identify couplings among classes. JCAT identifies the coupling types defined in Section 3 using the techniques described in Section 4.

JCAT is a source code static analysis tool that was developed in Java and collaborates with several software applications, including a Java Parser that is generated by ANTLR [58], Excel, and Access. Figure 3 shows the JCAT *system context diagram*, which shows data flows between the main application and the other entities and abstractions with which it communicates.

JCAT accepts the absolute pathname of a Java source-code package (**Java Source** in Figure 3) as input, then uses **JavaParser** to generate abstract syntax trees (**AST Files**) for each source code file in the package. **JavaParser** is generated by the **ANTLR** system [58] from the Java grammar. ANTLR helps to build abstract syntax trees (ASTs) by providing grammar annotations that indicate what tokens are to be treated as subtree roots, which are leaves, and which should be ignored with respect to tree construction.

Next, JCAT extracts information from the **AST Files** about class definitions, variable definitions, variable uses, method definitions, parameters and parameter uses, and method calls for each class. The intermediate results are saved in a database (**Access Database**). After all ASTs are processed, JCAT finds couplings via SQL queries. The computed coupling metrics are formatted into tabular forms and saved into **ASCII Text File** and **Excel Spreadsheet**.

5.1 JCAT user interface

Figure 4 shows JCAT's main screen, which has seven tabs. The **Main** tab (shown) lets the user enter a package to be analyzed, select the desired coupling types, choose the presentation format of the computed couplings, and determine where to save the computation results. The user can either enter the pathname of the target Java source-code package or browse to the target.

The user can select which couplings to compute by choosing from the check-boxes; any subset or all four can be chosen. The **Record** radio box gives the user the option of choosing between finding the existence of or computing the number of occurrences of couplings between two classes. The **Names** option tells JCAT to show the existence of a coupling between two classes by giving the coupling name, and the **Binary** option just shows whether coupling exists with the number "1." The intent is that the **Binary** option will simplify integration

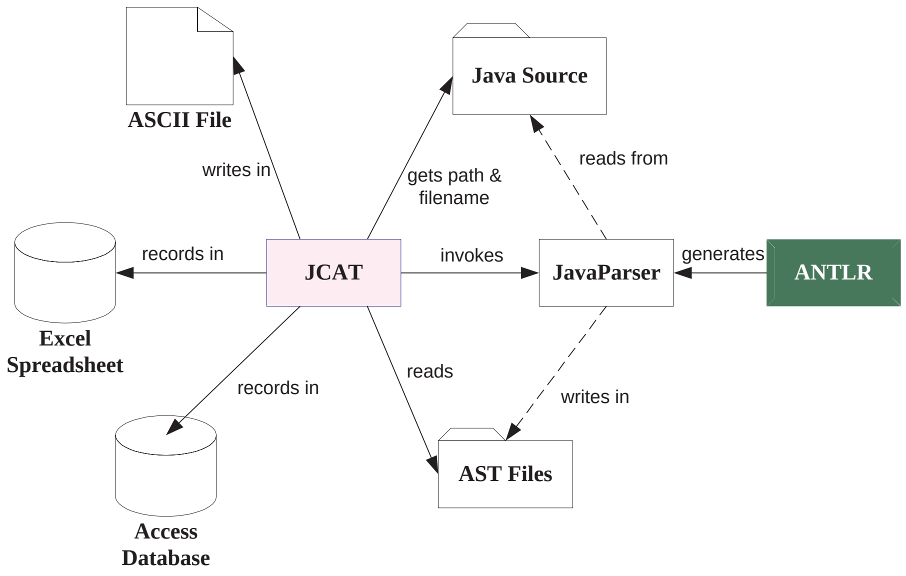


Fig. 3. JCAT Context Diagram

with other tools and the **Names** option will be clearer to human users. If **Count** is chosen, JCAT computes all the instances of couplings for each coupling type.

The **Save to** radio box allows the user to save the coupling results to either an ASCII text file or a spreadsheet. The table rows and columns both represent Java class file names and each table entry has the coupling information between the two files. Finally, the **Run** button starts the computation, and the **Exit** button terminates JCAT.

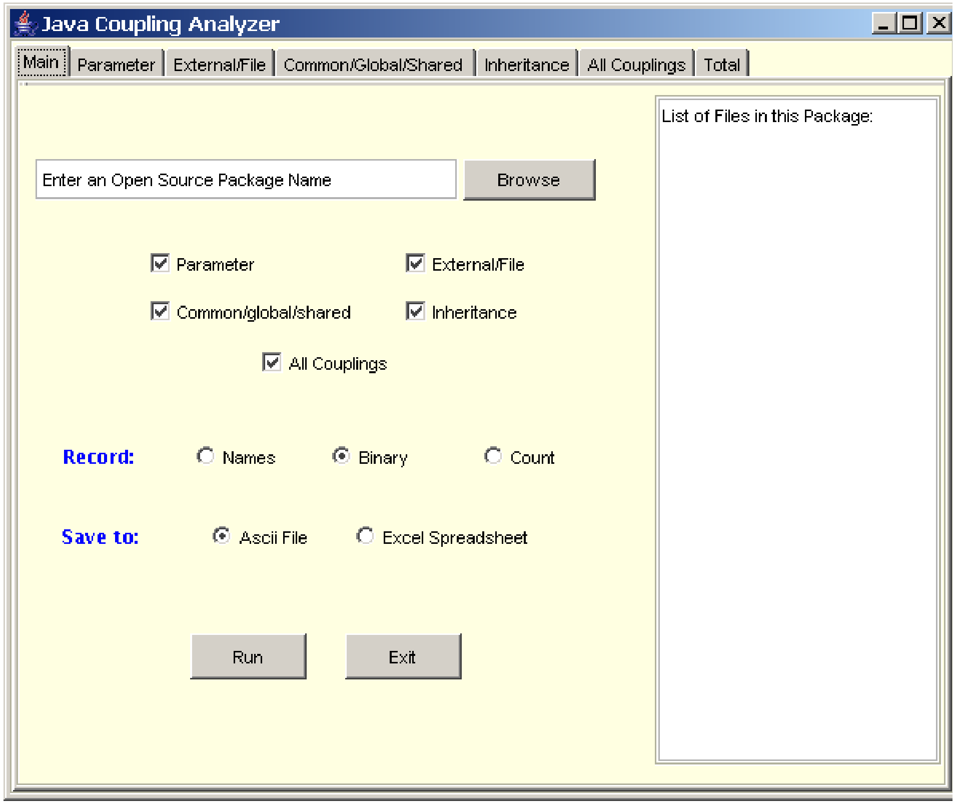
After the target Java source-code package is analyzed, the **Parameter**, **External/file**, **Common/global/shared**, and **Inheritance** tabs show the parameter, external, global, and inheritance couplings. The **All Couplings** tab shows all couplings together in one table. The **Total** tab shows the total number of each coupling for each class and for the whole package.

5.2 JCAT software design and implementation

JCAT has two packages, *coupling* and *query*. The *coupling* package is responsible for accepting the input, parsing the Java source code, formatting the coupling result for presentation, and exporting the results to an output file and user interface. It uses the *query* package by invoking methods from its classes to extract coupling information.

JCAT considers five independent categories of information to detect coupling, all statically available in the source code:

1. How are classes related to each other structurally? (e.g., association, aggregation, or inheritance)



[✓] Parameter [✓] External/File
[✓] Common/global/shared [✓] Inheritance
[✓] All Couplings

Record: Names [✓ Binary] Count

Save to: [✓ Ascii File] Excel Spreadsheet

Fig. 4. The Main User Interface of JCAT.

2. How are the public and protected variables used? (e.g., locally or in other classes)
3. How are the public and protected method/constructors used?
4. What is the scope of a variable or a method?
5. If there is an external device used by the system, is the external device shared among classes in the package?

JCAT has several algorithms for identifying the above information based on the generation and use of static structure information, as discussed in Section 3. JCAT has been tested with packages of various sizes, and the coupling measures have been verified by hand. The largest package that we have used with JCAT has 421 classes, 2863 methods, 12680 lines of executable code, and JCAT needed about 25 minutes on a PC running Windows XP. There is no theoretical upper limit on the number of classes that JCAT itself can handle.

6 Coupling Measures from Open-Source Software

We used JCAT to measure the couplings in 11 open-source projects of various sizes. These projects were developed in Java and the source files were downloaded from their project web sites.

Of these 11 projects, nine were downloaded from the Apache software foundation website. *Jakarta Servlet*[2] contains the code for the implementation classes of the Java Servlet and JavaServer Pages (JSP) APIs (packages `javax.servlet`, `javax.servlet.http`, `javax.servlet.jsp`, and `javax.servlet.jsp.tagext`). *Tomcat*[3] is a servlet container for Java Servlets and JSPs. It provides a Java virtual machine and associated elements to give a complete Java runtime environment, and also provides web server software to make the environment accessible on the Web. *Tomcat Catalina* is the Servlet container portion of Tomcat and *Tomcat Jasper* is the JSP engine used by Tomcat.

ANTLR[4] is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing Java, C#, or C++ actions. We used ANTLR to generate JCAT's Java parser.

TulipChain[5] is a specialized web browser, link checker, and editor that presents the contents in an explorer view.

Velocity[6] is a Java-based template engine. It permits web page designers to reference methods defined in Java code. Web designers can work in parallel with Java programmers to develop web sites according to the Model-View-Controller (MVC) model, allowing web page designers to focus on creating a well-designed site, and programmers to focus on writing well-designed software. Velocity separates Java from the HTML, making the web site more maintainable and providing an alternative to JSPs and PHPs.

Tapestry[7] is an open-source and all-Java framework for creating web applications in Java.

Table 3 shows the projects and the descriptive summary of the results. For each project, JCAT calculated the occurrences of each of the four types of coupling. The second column gives the number of classes (NOC) in each project, the third column gives the number of lines of code (LOC), and the third column gives the number of public variables (pvar). The next three pairs of columns provide data for parameter, global, and inheritance coupling. In each, the first sub-column gives the total number of couplings of that type in the package and the second sub-column gives the ratio of total coupling counts to the number of classes. The acronyms PCC, ECC, GCC, and ICC represent parameter, external, global, and inheritance coupling counts, respectively. Because global coupling has traditionally been of special concern on the part of developers, it

[2] Online at: `http://cvs.apache.org/viewcvs.cgi/jakarta-servletapi-5/`
[3] Online at: `http://jakarta.apache.org/tomcat/`
[4] Online at: `http://www.antlr.org/download.html`
[5] Online at: `http://ostermiller.org/tulipchain/`
[6] Online at: `http://jakarta.apache.org/velocity/`
[7] Online at: `http://linux.cs.lewisu.edu/apache/jakarta/tapestry/source/3.0-beta-1/`

is broken out in the column labeled GCC/TCC, which gives the percentage of global coupling to the total coupling counts. None of these packages exhibited any external couplings, so it is omitted from Table 3.

Table 3. Coupling Measures of Eleven Open-Source Software Packages, Including 3 Versions of Jakarta Servlet

Project	Size			Param		Global		Inheritance		GCC/
Name	NOC	LOC	pvar	PCC	PCC/NOC	GCC	GCC/NOC	ICC	ICC/NOC	TCC
Jakarta Servlet (v152(5.0.9))	42	576	32	17	0.40	2	0.05	15	0.36	6%
Jakarta Servlet (v152(5.0.12))	42	558	32	14	0.33	1	0.02	15	0.36	3%
Jakarta Servlet (v154)	40	819	45	24	0.60	10	0.25	17	0.43	20%
Tomcat Catalina (v5.0.12)	348	28,887	320	1953	5.61	321	0.92	158	0.45	13%
Tomcat Jasper2 (v5.0.12)	79	10,451	106	224	2.84	8	0.10	18	0.23	3%
ANTLR	197	16,673	218	3881	19.70	1879	9.54	135	0.69	32%
TulipChain	27	3636	66	142	5.26	8	0.30	11	0.41	5%
Velocity	30	1216	38	46	1.53	5	0.17	17	0.57	7%
Tapestry (Framework)	421	12,680	70	1605	3.81	59	0.14	337	0.80	3%
Tapestry (Contrib)	100	2403	17	52	0.52	12	0.12	21	0.21	14%
Tapestry (Example-vlib)	99	3200	51	251	2.54	29	0.29	32	0.32	9%
Min Value	27	558	17	14	0.33	1	0.02	11	0.21	3%
Max Value	421	28,887	320	3881	19.70	1879	9.54	337	0.80	32%

The last two rows in Table 3 show the minimum and the maximum values for each column. Figure 5 shows how the values are distributed for each of the measurements in a "normalized box-plot" graph. Because the ranges of the values varied so much, the numbers from each column in Table 3 are normalized to the range 0 to 100. Each box represents the 50% of the values in the middle, and the lines above and below the boxes extend to the highest and lowest value.

The stars represent outlier values. For example, consider the number of classes (NOC). The absolute values ranged from 27 to 421, and are normalized to the range 6.4 to 100. Half the values are in the box, or the range 10 to 42, and the mean is 19.

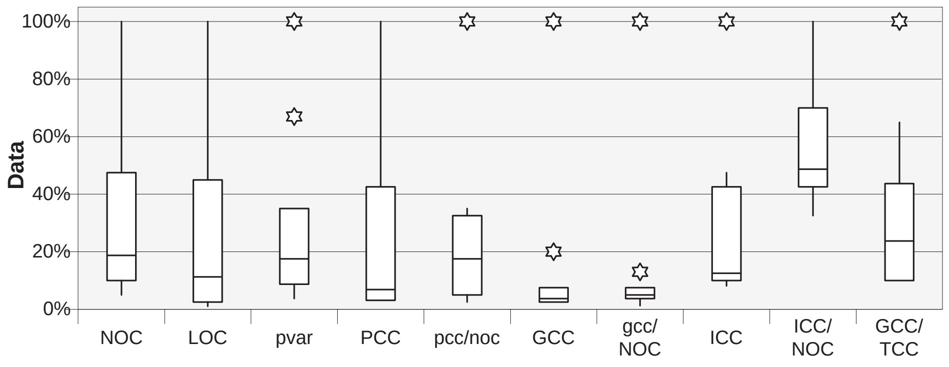


Fig. 5. Normalized Box Plot for Data in Table 2.

7 Analysis and Discussion

This section analyzes and discusses the data in Table 3. Several general observations are made. The amount of coupling across softwares system varies greatly, and coupling sometimes increases as software systems go through maintenance. Finally, there are a number of correlations between our coupling measure and size measures of the packages.

7.1 Coupling Ranges

The amount of couplings in the 11 products of Table 3 varies considerably. For example, the average parameter coupling per class (PCC/NOC) ranges from .33 for Jakarta Servlet 5.0.12 to 19.70 for ANTLR. Global coupling (GCC/NOC) ranges from a low of .02 for Jakarta Servlet 152(5.0.12) to a high of 9.54, also in ANTLR. The variation for inheritance coupling per class (ICC/NOC) is less, only from .21 in Tapestry (Contrib) to .80 in Tapestry (Framework). The ratio of global coupling to total coupling (GCC/TCC) also varies greatly, from a low of 3 percent in Tapestry (Framework) and in Tomcat (Jasper2) to 32 percent in ANTLR. Based on these numbers, we would anticipate that it would be more difficult to make changes to ANTLR than to the other projects. Indeed, the developer agreed[8] and has recently redesigned ANTLR to reduce the amount of global coupling.

[8] The developer, Terence Parr at University of San Francisco, explained via email that version 2 had significant problems, which recently motivated him to write version 3.

7.2 Coupling Increases with Version Number

Consider the data for the three versions of the same project, Jakarta Servlet. Although the number of classes shrank from 42 to 40, the amount of coupling increased. In particular, the relative percentage of global coupling increased dramatically, from 6 percent of the total coupling to 20 percent. Although the number of lines of code also increased, many design experts would view the increased use of global coupling as a negative trend.

This result agrees with a previous study [59], which observed an increase in global coupling with version number in 365 versions of Linux. However, Linux is written largely in C, not Java. The importance of implementation language is discussed in the next subsection.

7.3 Java Global Coupling Is Intentional

In the C programming language, the default access is `public`. In C++, the default access of methods and attributes is private, but the default access of `struct`s is `public`. Also, access modifiers appear before an entire group of declarations in C++, so attributes that are added carelessly to a class header file might accidentally become publicly available if there is a `public` access modifier. Accordingly, in some languages, global coupling can arise simply out of ignorance or carelessness.

In Java, however, the default access is `package`. Consequently, global coupling in a Java project can arise only if the designers or programmers explicitly insert it into the project.

This contrasts somewhat with the earlier study of Linux [59], which is implemented largely in C. It is possible that at least some global coupling might have arisen accidentally. In contrast, every instance of global coupling in the sub-column of Table 3 headed GCC must have been explicitly inserted into the code, including the 321 instances of global coupling in Tomcat Catalina and the 1,879 instances in ANTLR. This is a considerable amount of global coupling and must be considered a medium- to long-term threat for Tomcat and ANTLR.

7.4 Correlations between Coupling Measures and Size

An obvious question about the data in Table 3 is whether the differences in couplings are entirely due to size or something else. Accordingly, we examined the data to look for correlations. As a start, we applied the Anderson-Darling test of normality [63] to each column of Table 3.

Parametric statistics uses statistical significance testing based on sampling distributions. If we understand how a variable is distributed in a population, then we can predict how in repeated samples of equal size, this variable will behave in terms of how it is distributed. However, this parametric statistical significance testing does not apply if the underlying distribution is not normal [62]. Because none of the data in our study are even remotely normally distributed, non-parametric statistics such as Spearman's rank correlation [47] have to be employed.

We calculated the Spearman rank correlation between the quantities shown in Table 3. The results are shown in Table 4. Not surprisingly, there was statistically significant correlation at the 95 percent level of confidence between the number of lines of code (LOC) and the number of instances of each type of coupling (PCC, GCC, and ICC). That is, the more code, the more coupling. We observe that this does *not* mean we can use LOC to infer the amount of coupling.

Table 4. Correlations in Terms of Spearman Rank Correlation R Values

Spearman Rank Correlation	**NOC**	**LOC**	**PCC**	**GCC**	**ICC**	**TCC**	**GCC / TCC**
NOC	1.000	0.597	0.542	**0.603**	**0.792**	0.542	0.173
LOC		1.000	**0.982**	**0.875**	**0.767**	**0.982**	0.218
PCC			1.000	0.834	0.767	1.000	0.173
GCC				1.000	0.558	0.834	0.547
ICC					1.000	0.767	-0.032
TCC						1.000	0.173
GCC/TCC							1.000

We turn now to the Spearman rank correlation between the number of classes (NOC) and the other variables. The correlation between NOC and LOC and between NOC and the number of instances of parameter coupling (PCC) was *not* statistically significant at the 95 percent level of confidence. Contrariwise, the correlation between NOC and the number of instances of global (GCC) and inheritance (ICC) coupling was indeed statistically significant (as indicated in bold). The fact that the data indicate that NOC was not correlated with parameter couplings is interesting. This lack of correlation leads us to deduce that, for the sample of 11 open-source Java projects we examined, parameter coupling is primarily used within classes (where we did not count), whereas global and inheritance coupling is primarily used among classes.

8 Conclusions and Future Work

This paper presents techniques for measuring couplings in object-oriented relationships between classes, specifically focusing on types of couplings that are not available until after the implementation is completed, and presents a static analysis tool that measures couplings among classes in Java packages. The measurement incorporates object-oriented language features and is based on static analysis of the software source code. Deriving coupling information from design documents allows the information to be available earlier (pre-implementation) and is therefore more useful for predictive purposes, whereas deriving coupling information from source code allows the information to be more precise and reflects decisions made during implementation that are not specified in the design documents.

The paper divides object-oriented couplings into four types. Source code patterns for these four couplings types have been identified and analysis techniques developed. The tool gives us the ability to gather and analyze data to study the impact of these types of coupling. It is widely believed that high coupling negatively impacts maintenance, and this tool can be used to validate or refute this belief. In addition, we can decide if different types of coupling are more problematic than others.

Coupling has been validated against several quality factors for procedural programs. The tool presented in this paper will allow the same validations to be attempted with object-oriented software.

In the future, we plan to use the coupling measurement presented in this paper to support numerous specific software engineering activities. Most of these are maintenance related, thus post-implementation measurements are useful.

An ongoing study is to correlate coupling with defects in new versions of the software. That is, when software is modified, are defects more likely to appear in parts of the program that exhibit high coupling? This information will allow maintenance programmers to try to avoid changing potentially problematic portions of the program, and help regression testers focus their efforts after changes are made.

The *class integration and test order* (CITO) problem is that of deciding the order in which software classes should be integrated into the complete system [13, 45, 49, 66]. We have already used the precise information available from code-based couplings to improve existing solutions to the CITO problem [1, 2].

The *change impact analysis* problem is that of deciding how much of an impact, or ripple effect, a proposed change to a software system will have on the rest of the system [8–10, 16, 20, 55, 64, 68]. Code-based coupling metrics will allow a more precise, quantitative, way to solve the change impact analysis problem.

The *software design quality assessment* problem is that of evaluating the overall design of a software system [21, 27, 50, 53, 57]. The code-based coupling metric can be used to evaluate implementations directly, to evaluate how well an implementation conforms to a design, and to determine whether software design patterns are present in the implementation.

Stability refers to the extent to which the structure of a design is preserved as it goes through successive maintenance versions [6, 29, 30, 41]. Elish recently looked for correlations between couplings and stability by measuring changes between versions of software [30, 48, 70]. The changes examined were classes added, classes deleted, and classes changed (all changes were treated the same way). He found a correlation between the number of added and deleted modules and the number of errors reported in the subsequent version, but did *not* find a correlation between changes. It seems likely that the reason no correlation between changes was found was because all changes were measured as being the same, that is, the measurement was too imprecise to observe the difference. By looking at the code itself, our measure has more precision and has the potential to give a deeper analysis of the changes.

Current methods that are used for these activities are often qualitative in nature and rely on substantial judgment from engineers. Our coupling measurement is based on source-level analysis, and can lead to more precision and better results. Experimental evaluation using JCAT can eliminate the estimation and subjective inputs. Basing the class integration and testing on couplings may lead to a safer and more efficient ordering. These coupling measures should be good indicators of the impacts of changes. The coupling measures above may not fully capture all the code-visible dependencies that are important for impact analysis, so they may need to be extended. The level of coupling analysis in this work will yield more detailed and precise ways to assess the quality of design, hopefully leading to a stronger scientific basis for evaluating designs.

In the future, we also hope to investigate how frequently the different variations of global coupling appear . It seems likely that the different global coupling types will impact maintenance and reliability differently. This type of analysis could guide developers toward better designs by helping them consider on which variations of global coupling to concentrate. JCAT currently does not differentiate among the different variations of coupling.

The JCAT tool is only a beginning. One possibility is to use it to evaluate the importance of different types of coupling. One possible method would be to use multiple linear regression to investigate the importance of each coupling type for predicting things such as likelihood of defects and effort required for maintenance.

When we wrote this paper, we did not consider the possibility of using another class's ID as a return value (called component coupling by other researchers [22, 51]). We are currently incorporating this type of coupling into the tool.

In conclusion, we believe that the techniques for measuring coupling presented in this paper can be used to improve software quality in a number of different ways. The extensions proposed at the end of this section may be even more useful.

Acknowledgments This work was sponsored in part by the National Science Foundation under grant number CCR-0097056. Offutt and Abdurazik were sponsored in part by National Institute of Standards and Technology (NIST), Software Diagnostics and Conformance Testing Division (SDCT) in support of its standards and conformance testing program. Thanks to Terence Parr, the developer of ANTLR, for the tool and for quick answers to our pushy questions.

References

1. Aynur Abdurazik and Jeff Offutt. Coupling-based class integration and test order. In *Workshop on Automation of Software Test (AST 2006)*, pages 50–56, Shanghai, China, May 2006.

2. Aynur Abdurazik and Jeff Offutt. Using coupling-based weights for the class integration and test order problem. *The Computer Journal*, pages 1–14, August 2007. DOI: 10.1093/comjnl/bxm054.
3. Roger T. Alexander and Jeff Offutt. Analysis techniques for testing polymorphic relationships. In *Proceedings of the Thirtieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA '99)*, pages 104–114, Santa Barbara CA, August 1999. IEEE Computer Society Press.
4. Roger T. Alexander and Jeff Offutt. Criteria for testing polymorphic relationships. In *Proceedings of the 11th International Symposium on Software Reliability Engineering*, pages 15–23, San Jose CA, October 2000. IEEE Computer Society Press.
5. Erik Arisholm. Dynamic coupling measures for object-oriented software. In *Proceedings of the Eighth IEEE Symposium on Software Metrics (METRICS'02)*, pages 33–42. IEEE, June 2002.
6. J. Bansiya. Evaluating framework architecture structural stability. *ACM Computing Surveys*, 32(1), March 2000.
7. Victor Basili, Lionel Briand, and Walclio Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, October 1996.
8. Haider Zuhair Bilal and Sue Black. Computing ripple effect for object oriented software. In *Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE)*, Nantes, France, July 2006.
9. Sue Black. Computing ripple effect for software maintenance. *Journal of Software Maintenance: Research and Practice*, 13(4):263, July-August 2001.
10. Shawn A. Bohner and Robert S. Arnold. *Software Change Impact Analysis.* IEEE Computer Society Press, Los Alamitos, California, 1996.
11. Lionel Briand, Prem Devanbu, and Walcelio Melo. An investigation into coupling measures for C++. In *Proceedings of the Nineteenth International Conference on Software Engineering*, pages 412–421, Bostom, MA, May 1997. IEEE Computer Society Press.
12. Lionel Briand, J. Feng, and Yvan Labiche. Using genetic algorithms and coupling measures to devise optimal integration test orders. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, pages 43–50, Ischia, Italy, 2002. IEEE Computer Society Press.
13. Lionel Briand, Yvan Labiche, and Yihong Wang. An investigation of graph-based class integration test order strategies. *IEEE Transactions on Software Engineering*, 29(7):594–607, July 2003.
14. Lionel Briand, Jürgen Wüst, and Hakim Lounis. Using coupling measurement for impact analysis in object-oriented systems. In *Proceedings of the 1999 IEEE Conference on Software Maintenance*, pages 475–482, Oxford, UK, August 1999.
15. Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, January/February 1999.
16. Lionel C. Briand, Jürgen Wüst, and Hakim Lounis. Using coupling measurement for impact analysis in object-oriented systems. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, Oxford, UK, 1999. IEEE Computer Society Press.
17. S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1992.
18. P. Coad and E. Yourdon. *Object-Oriented Analysis.* Prentice-Hall, Englewood Cliffs NJ, 1991.

19. P. Coad and E. Yourdon. *Object-Oriented Analysis.* Prentice-Hall, Englewood Cliffs NJ, 2nd edition, 1991.
20. J. S. Collofello and D. A. Vennergrund. Ripple effect analysis based on semantic information. In *AFIPS Conference Procceddings (NCC)*, volume 56, pages 657–82. ACM SIGSOFT/SIGPLAN, 1987.
21. L. L. Constantine and E. Yourdon. *Structured Design.* Prentice-Hall, Englewood Cliffs, NJ, 1979.
22. J. Eder, G. Kappel, and M. Schrefl. Coupling and cohesion in object-oriented systems. Technical report, University of Klagenfurt, 1994.
23. Mahmoud Elish. A case study on structural characteristics of object-oriented design and its stability. In *Proceedings of the 23rd IASTED International Multi-Conference: Software Engineering*, Innsbruck, Austria, February 2005.
24. Khaled El Emam, Walcelio Melo, and Javam C. Machado. The prediction of faulty classes using object-oriented design metrics. *The Journal of Systems and Software*, 56(1):51–62, July 2001.
25. Andreas Epping and Christopher M. Lott. Does software design complexity affect maintenance effort? In *Proceedings of the 19th NASA Software Engineering Laboratory Workshop*, Goddard Space Center, December 1994.
26. N. Fenton and A. Melton. Deriving structurally based software measures. *The Journal of Systems and Software*, 12(3):177–886, July 1990.
27. N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous & Practical Approach.* PWS Publishing Company, Boston, Massachusetts, 2nd edition, 1997.
28. Elaine Ferneley. Coupling and control flow measures in practice. *Journal of Systems and Software*, 51(2):99–109, 2000.
29. D. Grosser, H. Sahraoui, and P. Valtchev. Predicting software stability using case-based reasoning. In *Proceedings 17th IEEE International Conference on Automated Software Engineering*, pages 295–298, Edinburgh, UK, September 2002.
30. D. Grosser, H. Sahraoui, and P. Valtchev. An analogy-based approach for predicting design stability of Java classes. In *Proceedings 9th International Software Metrics Symposium*, pages 252–262, Sydney, Australia, September 2003.
31. M. Jazayeri H. Gall, K. Hajek. Detection of logical coupling based on product release history. In *Proceedings of the 1998 IEEE Conference on Software Maintenance*, pages 190–198, Bethesda, MD, November 1998.
32. Gregory A. Hall, Wenyou Tao, and John C. Munson. Measurement and validation of module coupling attributes. *Software Quality Journal*, 13:281–296, 2005.
33. Mark Harman, Margaret Okunlawon, Bala Sivagurunathan, and Sebastian Danicic. Slice-based measurement of coupling. In *IEEE/ACM ICSE workshop on Process Modelling and Empirical Studies of Software Evolution (PMESSE'97)*, pages 28–32, Boston, Massachusetts, May 1997.
34. S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, 7(5):510–518, 1981.
35. M. Hitz and B. Montazeri. Measuring coupling and cohesion in object-oriented systems. In *Proceedings of the International Symposium on Applied Corporate Computing*, pages 412–421, Monterrey, Mexico, October 1995.
36. D. H. Hutchens and V. R. Basili. System structure analysis: Clustering with data bindings. *IEEE Transactions on Software Engineering*, 11(8):749–757, August 1985.
37. IEEE. *Standard Glossary of Software Engineering Terminology.* Institute of Electrical and Electronic Engineers, New York, 1990. ANSI/IEEE Std 610.12-1990.
38. IEEE. *Standard for a Software Quality Metrics Methodology.* Institute of Electrical and Electronic Engineers, New York, 1998. ANSI/IEEE Std 1061-1998.

39. Darrel Ince. *Software Development: Fashioning the Baroque.* Oxford University Press, Oxford, England, 1988.
40. P. Jalote. *An Integrated Approach to Software Engineering.* Springer-Verlag, New York NY, 1991.
41. M. Jazayeri. On architectural stability and evolution. In *Reliable Software Technlogies-Ada-Europe 2002*, pages 13–23. Springer-Verlag, Lecture Notes in Computer Science, 2002.
42. Zhenyi Jin and Jeff Offutt. Integration testing based on software couplings. In *Proceedings of the Tenth Annual Conference on Computer Assurance (COMPASS 95)*, pages 13–23, Gaithersburg MD, June 1995. IEEE Computer Society Press.
43. Zhenyi Jin and Jeff Offutt. Coupling-based criteria for integration testing. *Software Testing, Verification, and Reliability*, 8(3):133–154, September 1998.
44. D. Kafura and S. Henry. Software quality metrics based on interconnectivity. *The Journal of Systems and Software*, 2:121–131, 1981.
45. David Kung, J. Gao, P. Hsia, J. Lin, and Y. Toyoshima. Class firewall, test order, and regression testing of object-oriented programs. *Journal of Object-Oriented Programming*, 8(2):51–65, 1995.
46. David Kung, Jerry Gao, Pei Hsia, Yasufumi Toyoshima, Chris Chen, Young-Si Kim, and Young-Kee Song. Developing an object-oriented software testing and maintenance environment. *Communications of the ACM*, 38(10):75–88, October 1995.
47. Russell Langley. *Practical Statistics Simply Explained.* Dover Publications, New York, 1971.
48. W. Li, L. Etzkorn, C. Davis, and J. Talburt. An empirical study of object-oriented system evolution. *Information and Software Technology*, 42(6):373–381, 2000.
49. Brian A. Malloy, Peter J. Clarke, and Errol L. Lloyd. A parameterized cost model to order classes for class-based testing of C++ applications. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, Denver, Colorado, 2003. IEEE Computer Society Press.
50. Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices.* Prentice Hall, Upper Saddle River, NJ, 2003.
51. J. May, G. Hughes, and N. Shaban. Formal coupling of software components. In *Fifteenth Annual UK Performance Engineering Workshop*, pages 35–44. Research Press, 1999.
52. T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1967.
53. G. Myers. *Reliable Software Through Composite Design.* Mason and Lipscomb Publishers, New York NY, 1974.
54. Jeff Offutt, Mary Jean Harrold, and P. Kolte. A software metric system for module coupling. *The Journal of Systems and Software*, 20(3):295–308, March 1993.
55. Jeff Offutt and Li Li. Algorithmic analysis of the impact of changes to object-oriented software. In *Proceedings of 1996 IEEE Conference on Software Maintenance*, pages 171–184, Monterey, CA, November 1996. IEEE Computer Society Press.
56. Alex Orso and Mauro Pezze. Integration testing of procedural object oriented programs with polymorphism. In *Proceedings of the Sixteenth International Conference on Testing Computer Software*, pages 103–114, Washington DC, June 1999. ACM SIGSOFT.
57. M. Page-Jones. *The Practical Guide to Structured Systems Design.* YOURDON Press, New York, NY, 1980.

58. Terence Parr. ANother Tool for Language Recognition, 1997. http://www.antlr.org, last access March 2008.
59. Stephen R. Schach, Bo Jin, David R. Wright, Gillian Z. Heller, and A. Jefferson Offutt. Maintainability of the Linux kernel. *IEE Proceedings, Special Issue on Open Source Software Engineering*, 149(1):18–23, February 2002.
60. N. F. Schneidewind. Methodology for validating software metrics. *IEEE Transactions on Software Engineering*, 18(5):410–422, May 1992.
61. R. W. Selby and V. R. Basili. Analyzing error-prone system structure. *IEEE Transactions on Software Engineering*, 17(2):141–152, February 1991.
62. Inc StatSoft. *Electronic Statistics Textbook*. StatSoft (online), Tulsa, OK, 2006. http://www.statsoft.com/textbook/stathome.html.
63. M. A. Stephens. EDF statistics for goodness of fit and some comparisons. *Journal of the American Statistical Association*, 69(347):730–737, September 1974.
64. W. Stevens, G. Meyers, and L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.
65. W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.
66. Kuo-Chung Tai and F. J. Daniels. Test order for inter-class integration testing of object-oriented software. In *The Twenty-First Annual International Computer Software and Applications Conference (COMPSAC '97)*, pages 602–607, Santa Barbara CA, August 1997. IEEE Computer Society.
67. D. A. Troy and S. H. Zweben. Measuring the quality of structured designs. *The Journal of Systems and Software*, 2:112–120, 1981.
68. F. G. Wilkie and B. A. Kitchenham. Coupling measures and change ripples in C++ application software. *Journal of Systems and Software*, 52(2-3):157–164, 2000. Maintenance, OO, Metrics.
69. Franck Xia. On the concept of coupling, its modeling and measurement. *Journal of Systems and Software*, 50(1):75–84, January 2000.
70. S. S. Yau and J. S. Collofello. Some stability measures for software maintenance. *IEEE Transactions on Software Engineering*, 6(6), November 1980.

Jeff Offutt is Professor of Software Engineering at George Mason University. His current research interests include software testing, analysis of Web applications, object-oriented software, and software maintenance. He has published over 100 refereed research papers and the textbook *Introduction to Software Testing* (Campbridge University Press, 2008). Offutt is the editor-in-chief of Wiley's Software Testing, Verification and Reliability journal, and on editorial boards for EmSE, SoSyM, and SQJ. He received the Best Teacher Award from the School of Information Technology and Engineering in 2003. Offutt received a PhD degree from the Georgia Institute of Technology.

Aynur Abdurazik received the BEng degree in Computer Engineering from Beijing University of Posts and Telecommunications, Beijing, China, the MS degree in Software Engineering from George Mason University, and the PhD degree in Computer Science from George Mason University. Her research interests are in the area of software engineering, including object-oriented software analysis and testing.

Stephen R. Schach is an Associate Professor in the Department of Electrical Engineering and Computer Science at Vanderbilt University, Nashville, Tennessee. Steve is the author of over 130 refereed research papers. He has written twelve software engineering textbooks, including *Object-Oriented and Classical Software Engineering*, Seventh Edition (McGraw-Hill, 2007). He consults internationally on software engineering topics. Steve's research interests are in empirical software engineering and open-source software engineering. He obtained his PhD from the University of Cape Town.