

Characterising the Use of Encapsulation in Object Oriented Systems

November 3, 2009

Janina Voigt

`jvo24@student.canterbury.ac.nz`

Department of Computer Science and Software Engineering

University of Canterbury, Christchurch, New Zealand

Supervisors: Dr. Warwick Irwin and Dr. Neville Churcher

Abstract

Software is commonly very large and complex, and consequently hard to develop, understand and maintain. Encapsulation is the practice of breaking a system up into well-defined pieces and hiding internal details within each piece. It supports modularisation and information hiding, making it one of the most fundamental tools software developers have for managing complexity. Object oriented design heuristics have been proposed to help developers achieve better software designs and thus improve overall software quality; many of these design heuristics concern the use of encapsulation or are based on underlying assumptions about encapsulation. However, design advice in this area often conflicts. Little is known about how developers apply encapsulation in practice. In this work, we identify and compare two radically different schools of thought that underpin encapsulation policies and lead to the conflicting encapsulation advice. We conducted a survey to reveal which encapsulation policy is more intuitive, and found that novices' intuition about encapsulation differs from the encapsulation mechanisms supported by languages such as Java and C#. Following the survey, we empirically analysed encapsulation in real-world software to determine which encapsulation policies are followed in practice, uncovering a general culture of confusion and inconsistency around the use of encapsulation. This finding leads us to propose refactoring tools and a visualisation for helping developers improve encapsulation in software.

Acknowledgments

I would like to thank my supervisors Warwick Irwin and Neville Churcher for their constant support, encouragement and enthusiasm. Working with you on this project has been a real pleasure.

Contents

1	Introduction	4
1.1	Main research contributions	4
1.2	Report Outline	4
2	Background and Motivation	5
2.1	Object Oriented Programming	5
2.2	Encapsulation	5
2.2.1	Class versus Object Encapsulation	6
2.3	Software Quality	8
2.4	Software Metrics	8
2.5	Design Heuristics	9
2.6	Static Analysis Semantic Models	11
2.6.1	Java Symbol Table	11
2.7	Qualitas Code Corpus	12
3	A Discussion of Encapsulation Advice	12
3.1	Data Protection	12
3.2	Getters and Setters	13
4	Encapsulation Survey	14
4.1	Motivation	15
4.2	Participants	15
4.3	Task	15
4.4	Results	16
5	Analysing Encapsulation in Source Code	19
5.1	Encapsulation Analysis Tool	19
5.2	The Qualitas Code Corpus	20
5.3	The Use of JST	21
5.4	Program Design	21
5.5	Experimental Setup	22
5.6	Results	23
5.7	Discussion	25
6	Refactoring Tools	26
6.1	JST Parse Trees	26
6.2	Encapsulation Refactoring Tools	27
6.3	Limitations	27
7	A Visualisation of Encapsulation in Software	28
7.1	Design	28
7.2	Informal Evaluation	31
8	Discussion	33
9	Conclusions and Future Work	34
10	References	36
A	Large Diagrams	41
B	Detailed data from encapsulation analysis of real-world and student programs	44
C	Paper: Intuitiveness of Class and Object Encapsulation	47
D	Paper: Incest - Is it ok if you use protection?	54

E	A Guide to Using Java Symbol Table to Analyse Software	64
F	A Guide to Using JST Parse Trees for Code Generation	70
G	Human Ethics Application for the Encapsulation Surveys	78

1 Introduction

Real-world software is usually large, containing thousands or even millions of lines of code. It is also very complex and intangible making it difficult to understand and visualise. This leads to many software projects not meeting their specification, or even failing completely.

Encapsulation refers to the practice of hiding data and implementation details within a part of the program (such as a class) [27], enhancing the modularity of a system and supporting information hiding [74]. Hiding internal details within program parts decreases the amount of coupling between parts, making the system easier to understand and maintain and decreasing its overall complexity.

Over the years, many experienced software designers have attempted to formalize their knowledge about software design in an attempt to help software developers create higher quality software. Such knowledge has been collected as design heuristics, principles, methodologies, metrics and design patterns [38]. A number of design heuristics and principles advise developers how to practise encapsulation. Despite the amount of work done in this area, Garzás *et al.* note that there is still a long way to go in order to systematise this knowledge and make it accessible and easy to apply in practical cases [38].

This research was motivated by the observation that there is a lot of conflicting advice concerning the use of encapsulation in Object Oriented (OO) systems. While encapsulation is widely used in practice as a tool to counter complexity in software, it is unclear which subset of the conflicting advice developers follow in practice and why. As encapsulation is one of the most fundamental tools for managing software complexity, it is important to have a good understanding of the reality of developers' practices and the implications of those practices.

In this research we clarify the different schools of thought on encapsulation. We investigate the intuition of developers regarding encapsulation and analyse the use of encapsulation in industrial-scale Java programs. We propose refactoring tools, a visualisation and guidelines to help developers practise encapsulation correctly and consistently.

Our research uses a semantic model for Java called Java Symbol Table (JST) [45], which contains rich information about programs' semantic entities and relationships between them. We also make use of the Qualitas Code Corpus from the University of Auckland [69] as a source of programs to evaluate.

1.1 Main research contributions

The main contributions of this research include:

- The identification, comparison and clarification of different encapsulation policies and schools of thought surrounding encapsulation;
- Survey results indicating that encapsulation mechanisms in modern programming languages such as Java and C# do not match developers' intuition;
- Empirical evidence that encapsulation practices in real-world software are inconsistent and that encapsulation is generally weak;
- Tools to help developers improve encapsulation in software; and
- Guidelines and recommendations to help developers practise encapsulation more consistently.

The findings of this research about encapsulation practices have far-reaching implications. The inconsistency of encapsulation practices in real-world software casts doubt on the general quality and maintainability of software. The fact that encapsulation mechanisms in modern programming languages do not meet developers' expectations uncovers the need to rethink common design advice and the need to design better programming languages.

1.2 Report Outline

The remainder of the document outlines the background, method and results of the research. It is structured as follows:

- Section 2 presents the background, including information about encapsulation, design heuristics, software metrics, and semantic models.
- Section 3 analyses advice about encapsulation, highlighting similarities and conflicts.

- Section 4 presents the survey we conducted to reveal the encapsulation tendencies of professional developers and students and describes the results and conclusions of this survey.
- Section 5 presents the tool we developed to analyse encapsulation in real-world software and describes the results obtained by analysing a number of real-world and student programs.
- Section 6 presents a refactoring tool we have developed to automatically tighten encapsulation in a program. The section includes a description of how JST parse trees can be used for code generation.
- Section 7 presents an experimental visualisation developed to help developers more easily find encapsulation breaches in their software.
- Section 8 discusses the results obtained from the research as a whole and the implications for software development, and proposes new guidelines for using encapsulation.
- Section 9 presents the conclusions, reiterates the specific contributions of this work and describes future work to be done in this area.

2 Background and Motivation

2.1 Object Oriented Programming

OO programming is the dominant programming paradigm in modern software engineering. Many of the most widely used programming languages are object oriented, including Java, C++, C# and Python.

Objects as programming entities were introduced in the 1960s as part of Simula-67, which is generally considered to be the first OO programming language. In the 1970s, Smalltalk was developed at Xerox Parc by a team under the leadership of Alan Kay, who first coined the term ‘object oriented’ [52]. The aim of the project was to develop a high-level programming language suitable for children. The programming language they developed, Smalltalk, was exclusively based on objects and is thus recognised as the first pure OO language. OO programming’s similarities with how people understand the real-world have implications for encapsulation, as we discuss in Section 2.2.1.

Following the development of Smalltalk, it took another two decades or so for OO programming to become popular and widely used, in the mid 1990s. The C++ programming language incorporated and changed some of the concepts introduced by Smalltalk, and in this way popularised the idea of OO programming.

One of the reasons for the popularity of OO programming is that it offers advantages over traditional procedural programming [81, 91]. Riel [81] believes that the biggest advantage of OO programming is that it allows developers to more closely model the real world. He also argues that OO programming leads to a decentralised architecture and low coupling, meaning that a change in one part of the program will not affect the rest of the system.

2.2 Encapsulation

‘Programming is about managing complexity’ according to Bruce Eckel [32, page 6]. Complexity in software systems often leads to projects failing and systems not meeting their specifications. Encapsulation is arguably the most fundamental tool software developers have for managing this complexity.

Despite being such a basic and fundamental tool, even the definition of the term encapsulation is unclear. Rogers, for instance, suggests that encapsulation means only grouping of properties, and that hiding is a separate concept [82]. However, we argue that encapsulation naturally implies hiding, as suggested in a definition by Snyder [85]:

Encapsulation is a technique for minimizing interdependencies among separately-written modules by defining strict external interfaces. The external interface of a module serves as a contract between the module and its clients, and thus between the designer of the module and other designers.

Thus, encapsulation refers to the practice of breaking a program into cohesive parts with clearly defined boundaries and hiding implementation details — particularly data representations — within those parts. This means that only the part owning the data can access and modify it, leading to a modularised system. In this way, encapsulation supports the principle of information hiding [27, 74].

Although the concept of encapsulation predates OO programming, OO languages have added a number of mechanisms that further support encapsulation, including inheritance and polymorphism. These mechanisms add a level of indirection between the caller of a function and the function implementation so that the caller does not even have to know which method it is calling. However, these mechanisms also make encapsulation considerably more complex, raising a number of issues, such as whether or not subclasses should be able to directly access members in superclasses. This is made possible in many programming languages, including Java, C++ and C#, through the `protected` access modifier.

There are several benefits to be gained from the practice of encapsulation [27, 85]. Firstly, encapsulation allows a system to be divided into independent parts. Each encapsulated part has a particular responsibility and contains data related to its task. If fully encapsulated, no part can interfere with the data from another encapsulated part.

Because data is hidden within its encapsulated part, other parts of the system do not depend on it, leading to low coupling [96]. Therefore, the data representation can be easily changed or even moved without affecting a part's clients. Furthermore, a part of the system is always in control of its data, making bugs easier to find. If the data has been corrupted or changed unexpectedly, this must have happened within the part containing the data. In this way, encapsulation makes software development and maintenance less error-prone and programs easier to read [27, 85].

There is a lot of advice about how to use encapsulation, but we have observed that much of this advice is unclear and conflicting. An in-depth analysis of encapsulation advice and guidelines can be found in Section 3.

2.2.1 Class versus Object Encapsulation

We suggest that much of the conflict in encapsulation advice arises from the existence of two different schools of thought on OO encapsulation. This has led to encapsulation being supported in two incompatible ways in OO programming languages. Both approaches are described as OO, and the difference between them has received little attention in the literature, yet it has profound consequences. In this report, we will refer to the two encapsulation approaches as *object encapsulation* and *class encapsulation*.

Object encapsulation is used by languages such as Smalltalk and Ruby. In these languages, data is `private` to an object. This means that if an object contains data, only that object has the right to access and modify it. This ensures that the data cannot be inadvertently changed by other objects.

Many programming languages that are popular today, such as Java, C# and C++, use class encapsulation, meaning that data is `private` not to an object but to a class. Objects of the same class can access each others' `private` data. C++ was based on an existing procedural language that structures software using static modules, and it is therefore not surprising that it placed the encapsulation boundary around the static concept of classes. Stroustrup makes this explicit [87]: 'Note that in C++, the class — not the individual object — is the unit of encapsulation.' Other languages including C# and Java later adopted this approach.

The differences between object and class encapsulation are best explained using an example. Figure 1a shows a simple class hierarchy with four classes; Figure 1b shows some instances of these classes. Figure 1c contrasts the two encapsulation boundaries. When the encapsulation boundary is the class, data is hidden within each class. This means that two objects of the same class can access each other's private data. For example, o_2 can access the private field z in o_3 and vice versa. However, an object may not be able to access all of its own fields; private fields of the same object declared in another class are not visible. For example, o_2 does not have access to its own field x . When using object encapsulation, on the other hand, data is hidden within an object, meaning that two objects can never access each other's private data but an object always has access to all of its own data.

Class and object encapsulation represent very different underlying philosophies. Class encapsulation reflects a designer's mindset oriented around static, compile-time concepts. According to this mindset, it makes little sense to allow classes to access other classes' `private` members. However, the object encapsulation mindset is oriented around the runtime concept of objects, where each object is a single, independent entity. For this way of thinking, it does not make sense for an object to be able to access only part of itself.

The `protected` access modifier, which is included in some class encapsulation languages such as Java and C#, allows an approximation of object encapsulation. Subclasses have the ability to access the `protected` parts of their ancestors. However, even when using `protected` as an access modifier, the true encapsulation mechanism is still class encapsulation because objects can still access each other's `protected` members provided they belong to the same class. In Java, the `protected` access modifier is further removed from object encapsulation because it gives away access rights to all in the package, rather than just to subclasses.

The difference in the treatment of encapsulation between languages partly comes about because of static and dynamic typing. In statically typed languages, the type checking is done at compile-time when objects do not yet exist. This makes class encapsulation a more natural fit, allowing the compiler to enforce encapsulation at compile

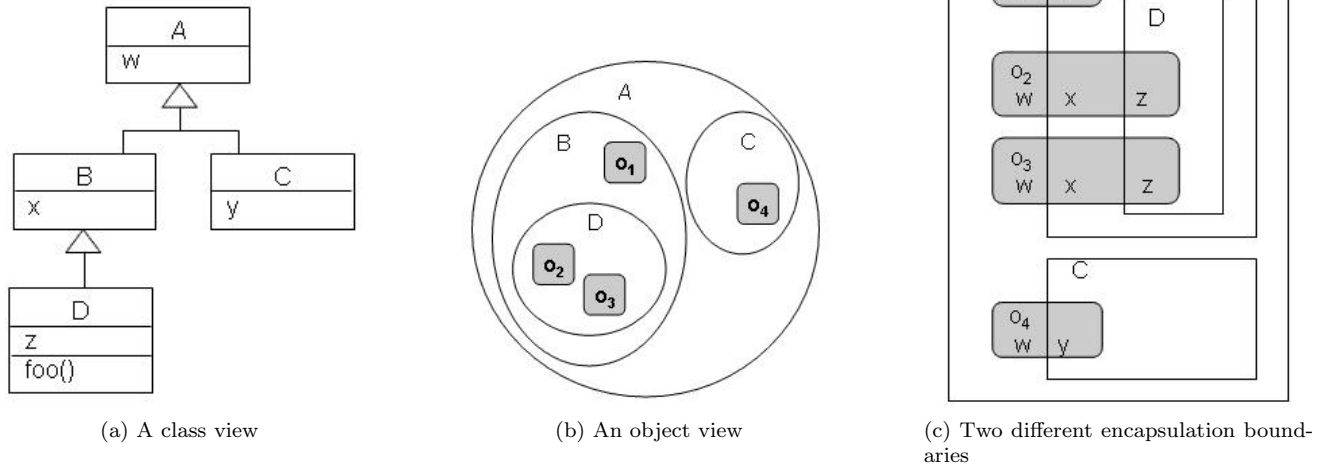


Figure 1: Object versus class encapsulation

time. In dynamically-typed languages on the other hand, type checking is done at runtime in the presence of objects, allowing object encapsulation to be enforced more easily. However, it is not the case that a statically typed language has to always use class encapsulation and a dynamically typed language has to use object encapsulation. Python, for example, is dynamically typed but uses classes as the unit of encapsulation. In Section 8, we propose simple changes to the Java grammar to allow it to enforce object encapsulation.

Each of the two types of encapsulation has its advantages and disadvantages. We argue that object encapsulation is more intuitive and we have observed that many students are surprised when they find out that objects of the same class can access each others' `private` data. This is not surprising given the parallels between OO and the real-world. Coad and Yourdon [25] quote the 1986 Encyclopaedia Britannica entry on Classification Theory [33]:

In apprehending the real world, [people] constantly employ three methods of organisation, which pervade all of their thinking:

1. the differentiation of experience into particular objects and their attributes — e.g. when they distinguish between a tree and its size or spatial relations to other objects.
2. the distinction between whole objects and their component parts — e.g. when they contrast a tree with its component branches, and
3. the formation of and the distinction between different classes of objects — e.g. when they form the class of all trees and the class of all stones and distinguish between them.

This description was probably written by a person not familiar with OO programming, but is easy for OO developers to recognise and understand because of the distinct similarities between OO and the real world. In the real world we naturally assume that each object is self-contained and independent and therefore implicitly place the encapsulation boundary around objects. This arguably means that object encapsulation is more intuitive than class encapsulation because it is a concept that we are familiar with in the real world.

However, while object encapsulation can be more intuitive, Snyder argues that it effectively breaks encapsulation in an OO system and removes all advantages to be gained from its use [85]. He believes that by allowing access to data in another class which may have been written by a different developer, the maintainability of the system is compromised. The reason he gives is that, should the other developer wish to change the internal data representation for that class, the subclass accessing the data will also be affected.

In addition to object and class encapsulation, we also recognise a third encapsulation type which we call *intersection encapsulation*. Intersection encapsulation uses the intersection of the object and class boundary as its

encapsulation boundary. This means that it disallows accesses to `private` members in other objects and other classes, crossing neither the class nor the object boundary. In Figure 1c, intersection encapsulation would only allow accesses to data within the same class and object box.

Intersection encapsulation is likely to be a strategy adopted in response to the general confusion around encapsulation boundaries. It appears to be safer than both object and class encapsulation because it crosses no boundaries. However, this is a restrictive approach because it provides minimal access to fields, which in turn may lead to heavier use of getters and setters, and hence an effective weakening of encapsulation.

We investigated which encapsulation boundary is more intuitive by conducting a survey; the results are presented in Section 4. We also looked at which encapsulation boundary is more prevalent in real source code; the results of this empirical investigation are presented in Section 5.

2.3 Software Quality

Quality is a highly important yet poorly defined property of software. It emerges from a combination of many diverse characteristics, including conformance to functional and non-functional requirements [77]. Some of these characteristics are internal to the program and cannot be perceived by the user, such as maintainability, while others are external, including usability and reliability.

A number of attempts to define software quality in order to enable measurement have been made. McCall, Richards and Walters propose a division of software quality characteristics into three categories: product revision, product operation and product transition, each containing a number of quality indicators [65]. A problem with this definition of software quality is that it is impossible to directly and objectively measure some of these quality factors [77]. A slightly simpler description of software quality was developed as part of the ISO 9126 standard [44]. It defines six key quality attributes: functionality, reliability, usability, efficiency, maintainability and portability.

Brito e Abreu and Melo studied the impact of OO design on software quality [31]. The authors collected data about the number of defects discovered and time spent on maintenance for various software systems and correlated this data with OO design characteristics such as inheritance and coupling. They found that inheritance and polymorphism, if used sparingly, decreased defect density and rework time. Coupling was positively correlated with defect density and rework time; the more coupling was present, the more defects occurred. Good encapsulation practices will lead to low coupling and should therefore reduce defect density and increase software quality.

Some of these results were confirmed by Briand *et al.* [11], who found that coupling, particularly through method invocations, and depth of a class in the inheritance tree have a significant negative influence on software quality. The results from these studies indicate that following good design principles and making use of appropriate OO programming language mechanisms such as inheritance, polymorphism and encapsulation can increase overall software quality.

2.4 Software Metrics

Software metrics [34, 40] are one solution developed to counteract the issue of complexity and intangibility in software. Metrics endeavour to measure specific characteristics of software in much the same way that other engineering disciplines measure their products. Engineers routinely use metrics to specify, develop and verify their products. However, while metrics are intrinsic to engineering disciplines and are widely used, the success of software metrics has been relatively limited.

A number of common and popular metrics predate the rise in popularity of OO programming, including Lines of Code, Halstead's *Complexity Measure* [39], McCabe's *Cyclomatic Complexity* [63, 64, 83] and *Maintainability Index* [93]. The advent of the OO programming paradigm required new metrics to measure a wide variety of new features that did not exist previously, including polymorphism and inheritance. However, the complexity of the OO model complicates the creation and calculation of metrics.

In response to the rise in popularity of OO programming, many OO metrics were proposed, among them the metrics suites developed by Chidamber and Kemerer [15, 16], Henry and Kafura [41, 42], Li and Henry [55], Lorenz and Kidd [58] and Briand *et al.* [9, 10].

Chidamber and Kemerer's suite [15, 16] of six OO metrics is the most widely known. It includes metrics to quantify inheritance, coupling and cohesion. However, this suite of metrics does not take into consideration the complexities of encapsulation in OO, among other weaknesses [62]. Many adjustments and extensions have been proposed to fill these gaps. Brito e Abreu [30, 31], for example, proposed a metrics suite called MOOD which includes metrics such as *Attribute Hiding Factor* and *Method Hiding Factor* designed to measure encapsulation. As with Chidamber and Kemerer's work, these metrics do not explicitly take into account encapsulation boundaries.

There have been various studies to validate the usefulness of software metrics [8, 11, 51, 86, 88]. Subramanyam and Krishnan [88] studied a number of large-scale industrial software systems, counted the number of defects and correlated this with various Chidamber and Kemerer metrics. They discovered that some of Chidamber and Kemerer's metrics significantly explained variances in the number of defects. Similar results were achieved by Briand *et al.* [11]. Basili *et al.* produced similar results, too finding that out of the six metrics proposed by Chidamber and Kemerer, all except *Number of Children* (NOC) appear to be useful to predict the fault-proneness of a class [8]. They conclude that 'most of Chidamber and Kemerer's OO metrics can be useful quality indicators'. However, others have criticised the metrics suite for being poorly defined and open to interpretation [23, 62].

Despite the large amount of research demonstrating the usefulness of metrics, few are used regularly by software engineers to assess their work or help make decisions. There are several reasons for this. Firstly, calculating and applying metrics can be complicated and time consuming. The results can often be hard to relate to program quality since the metrics measure very specific aspects of software. Lastly, there is usually no clear correspondence between what the metrics quantify and the decisions software engineers need to make in practice.

2.5 Design Heuristics

Good software design is essential to producing high quality software, but software design is a difficult and inexact science. There are many ways to design any one system and different people will usually come up with very different designs to solve the same problem. It can often be difficult to determine what the best solution is because each has advantages and disadvantages; there is no such thing as an optimal design. The core of software design is balancing various, sometimes conflicting, design forces [26].

Metrics can be used to point out design flaws in a system and in this way help developers improve the quality of their designs, but their success has been fairly limited. Beck and Fowler say that 'in our experience no set of metrics rivals informed human intuition.' [36]

Experienced OO designers can look at a design and identify its advantages and disadvantages. Much work has been done to try to capture their expertise. Arthur Riel, for example, published a set of 61 design heuristics which he calls 'golden rules' [81]. These heuristics are guidelines, rules of thumb, that inform developers about good and bad design practices [81]. Riel says that heuristics

are not hard and fast rules that must be followed under penalty of heresy. Instead, they should be thought of as a series of warning bells that will ring when violated. The warning should be examined, and if warranted, a change should be enacted to remove the violation of the heuristic. It is perfectly valid to state that the heuristic does not apply in a given example for one reason or another. [81, page xi]

Because of their inexact nature, design principles and heuristics can be very difficult to apply in practice [38]. Garzás *et al.* say that

a strong knowledge does not exist on items such as design principles, best practices, or heuristics. The problem confronting the designer is how to articulate all this explicit knowledge and to apply it in an orderly and efficient way in the OO design and analysis, in such a way that it is really of use to him or her. [38, page vii]

Many heuristics and principles have been proposed in addition to the 61 mentioned by Riel [81], including heuristics by John Lakos [54] and Ralph Johnson and Brian Foote [50]. Some heuristics are general, high level principles that act as guidelines for designers. *Separation of Concerns* [29] for example advocates separating unrelated parts. *Information Hiding* [74] advocates hiding data and implementation decision inside program parts and protecting it from accidental changes. *Design by Contract* [70] attempts to help developers be clear about the responsibilities of each method.

Many more specific design heuristics are based on top of these very general principles, including those proposed by Riel [81]. The *Liskov Substitution Principle* [57, 61], for example, gives developers guidance about how to use inheritance well. The *Law of Demeter* [56] discourages the use of getters and setters and aims to reduce coupling and enforce localisation. The *Acyclic Dependencies Principle* [60, 71] discourages the existence of cycles between packages.

There are various principles and heuristics concerning the use of encapsulation which are analysed in more detail in Section 3, including the *Law of Demeter* and heuristics by Riel. However, many more heuristics depend on underlying assumptions about the type of encapsulation used. Riel, for example, clearly considers the class to be the encapsulation boundary and advises developers to 'hide data within its class' [81]. This kind of thinking

has implications for other heuristics he proposes. For example, he tells developers to limit the depth of inheritance hierarchies to ensure that the number of classes a developer has to consider at any one time stays relatively small. This heuristic clearly reflects the fact that he thinks in terms of classes, not objects; when thinking in terms of objects, the depth of an inheritance hierarchy becomes less of a concern. Because encapsulation is so fundamental, it has a wide-reaching impact on a lot of design advice that may not even be directly related to encapsulation. Therefore, being aware of different encapsulation policies is essential when using and interpreting design advice.

In addition to design principles and heuristics, Fowler and Beck propose what they call *code smells*, which are similar in nature to Riel's heuristics [36]. These smells are certain code characteristics that act as warning signs, telling developers that there may be a problem. Fowler then proposes ways in which the problem can be solved through *refactoring*, or changing the design of the system without modifying its behaviour. Examples of code smells include *Long Method*, *Duplicated Code* and *Shotgun Surgery*. The term *smell* illustrates how vague and subjective code smells really are. In the case of the *Long Method* smell for example, it is up to the developer to decide when a method is too long and different developers may apply the smell in a very different way.

Design patterns form another set of tools developers can use when designing a system. Design patterns were first proposed by Christopher Alexander [4] for the domain of architecture. Alexander says

Each pattern describes a problem which occurs over and over again in our environment, and the describes the core of the solution to that problem, in such a way that you can use this solution a million times over without ever doing it the same way twice. [4]

Design patterns were first adapted to software by Coad [24] and further developed and consolidated by the Gang of Four (Gamma, Helm, Johnson and Vlissides) who proposed a total of 23 OO design patterns [37]. There are common problems that occur relatively often in software design. The Gang of Four searched for solutions that developers had used to solve these problems and presented them as software design patterns. Design patterns are essentially blueprints for solving a particular problem.

Apart from the well-known Gang of Four design patterns, a variety of other software patterns have been proposed including architectural patterns by Buschmann *et al.* [12] and analysis patterns by Fowler [35] and Konrad *et al.* [53].

Although design patterns are widely used, they can cause problems if applied incorrectly. Garzás *et al.* state that there are a number of problems that can occur when using design patterns, including 'difficult application, difficult learning, temptation to recast everything as a pattern, pattern overload, ignorance, deficiencies in catalogs, and so forth' [38]. This is backed up by Schmidt [84] and Wendell [94], who presents the problems encountered during the development of a large commercial software project. He found that the uncontrolled use and inappropriate application of design patterns led to severe maintenance problems.

Nevertheless, design patterns are widely recognised as good solutions to particular, common problems. Despite being 'good design' in many people's eyes, they often violate accepted heuristics. The reason for this is that there is simply no optimal solution, so that the best possible compromise between design forces needs to be found. Therefore, design patterns are useful in such situations to resolve these issues.

Bär and Ciupke analysed relationships between various heuristics and found that there was a number of relationships between different heuristics [6]. Firstly, they found that there are contradicting heuristics, originating from different opinions about what good design is [6, 81]. In the catalogue of 59 heuristics they considered, they found five pairs of conflicting guidelines and seven pairs that potentially conflict. We suggest that the reason for such contradictions is that heuristics are not absolute rules and there are various, sometimes conflicting schools of thought about what constitutes good design. When designing a system, there are usually a lot of tradeoffs between different design choices and the developer has to make a call about which heuristic should be followed.

An example of a conflict occurs during the design of a system that includes a graphical user interface (GUI) and background code. The *Separation of Concerns* principle [29] would tell developers to separate interface code from model code. On the other hand, one of Riel's heuristics advises developers to 'keep related data and behaviour together' [81]. The developer thus has to decide which of the principles to follow. He or she will look at the advantages and disadvantages provided by each of the heuristics and make a call on which one is more applicable in the current situation.

Bär and Ciupke also found that some heuristics imply others, meaning that conformance with one will automatically lead to conformance with the other [6]. They found 20 such implication relationships between the 59 heuristics they considered, forming a small hierarchy of heuristics.

Despite the inexact nature of heuristics, a lot of research has been done to look at automating them, allowing the computer to make developers aware of any breaches of heuristics and any potential problems [6, 13, 14, 18, 28, 72, 76]. Chatzigeorgiou *et al.* for example look at using link analysis to find *God classes* [13, 14]. *God classes* are classes

that have too much responsibility and should be avoided according to one of Riel’s heuristics [81]. Bär and Ciupke present a tool called GOOSE which uses design heuristics to assess the quality of legacy code [6]. It extracts information about the code and searches for violations of design heuristics. Correa *et al.* describe a tool which uses a Prolog knowledge base containing design heuristics, design patterns and anti-patterns to evaluate and suggest improvements to the design of a system [28]. Churcher *et al.* take a more high-level approach, using a rigorous semantic model as the basis for heuristics evaluation and automation, and propose ‘a framework in which heuristics can be proposed, expressed and evaluated.’ [18] They also suggest various visualisations of heuristics to present data to developers in an easy to understand manner. Other visualisations of heuristics have also been proposed, for example by Parnin *et al.* who present a number of lightweight visualisations to help developers find code smells [75, 76].

2.6 Static Analysis Semantic Models

Design heuristics are usually defined in terms of semantic concepts encountered in programming languages, such as classes, inheritance and method invocation. Source code syntax does not directly express these concepts and is therefore not ideally suited to extracting heuristics information. Semantic models solve this problem by providing an accurate representation of the semantics of a program, including program entities and relationships between them.

Although many design heuristics have been proposed, the number of tools measuring them and the success of such tools has remained relatively limited. One reason for this is that accurate tools such as semantic models are required for the measurement of the complex semantic features that are the subject of heuristics and principles. Irwin developed rich modelling approaches to allow such measurements, including a semantic model for Java called JST (described below) [45]. However, this is the first work which extensively uses these tools.

For procedural languages, semantic models are relatively simple, consisting mostly of functions as entities with invocation being the main relationship between two entities. However, in the OO paradigm, there is a much larger set of entities and relationships. OO entities include packages, classes, methods, variables, fields and parameters. There are a number of relationships possible between these entities such as inheritance, implementation, containment and method invocation.

Semantic models of programs can be built by analysing parse trees for a given program and extracting information about entities such as classes and methods, and the relationships between them. In this way, it produces a much richer representation of the program than the syntax alone provides, and this can be used to extract heuristics information and metrics.

2.6.1 Java Symbol Table

Java Symbol Table (JST) is a semantic model for Java, developed by Irwin and Churcher [45, 47, 48]. It constructs a comprehensive model of the semantic features of a Java program. This includes concepts such as packages, classes, methods, constructors, parameters, fields and local variables. The relationships between these entities are also represented by the model. JST currently accepts source code which is correct under Java version 1.6, although some of the latest Java syntax, including generic methods, is not yet completely handled.

Information can be extracted from JST by ‘walking’ the semantic model. This is easily done using a Visitor design pattern [37] as demonstrated in previous work [73, 92] and described in Section 5.3. More technical detail about JST visitors can also be found in a guide written by the author about using JST in Appendix E.

JST is part of the XML visualisation pipeline [19, 46], which allows an easy transition from source code to metrics and visualisations. The general structure of the visualisation pipeline can be seen in Figure 2. Source code is first parsed using a Java parser to produce parse trees. These parse trees can be read in and analysed by JST to produce a semantic model of the program. The model can be queried to calculate metrics, for example, or extract specific information about the program which can then be visualised in various ways including class cluster visualisations [20, 47] or virtual world visualisations [17, 21, 22]. The input and output for each consecutive stage of this process is in XML.

This research fits into the metric calculation and visualisation part of the visualisation pipeline, allowing it to be integrated with existing infrastructure.

Other semantic models for Java already exist, for example [49], but none of them are as rich and accurate as JST [45]. They commonly struggle to resolve polymorphic methods, leading to a model that only approximates the actual program structure. This research will make use of JST because of its greater accuracy compared to other semantic models.

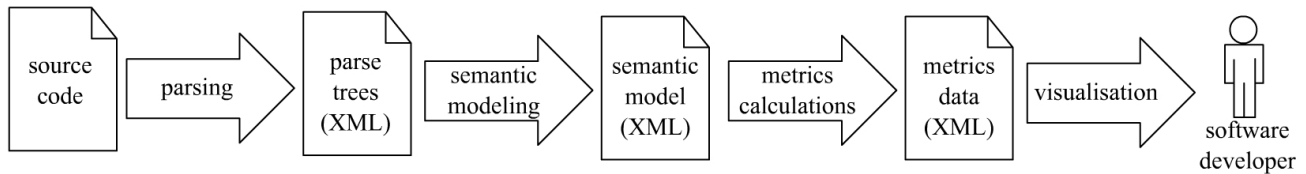


Figure 2: An overview of the structure of the XML visualisation pipeline

2.7 Qualitas Code Corpus

The Qualitas Code Corpus is a corpus of Java programs collected by Tempero *et al.* from the University of Auckland [69]. The corpus has been extensively used by Tempero and others for a variety of purposes. They have done a number of empirical studies into the structure of Java programs, including the use of inheritance [90], the use of dependency injection [95] and cycles in Java programs [67, 68]. Some of their work has also focused on the evaluation of metrics such as cohesion metrics [7] and on identifying opportunities for refactoring [66]. A recent study conducted by Tempero looked at the use of fields and access modifiers in Java [89]. Tempero measured the relative frequency of use of access modifiers and the degree to which fields were exposed. This work is in some ways similar to ours, but does not explicitly address the encapsulation boundary issue. We revisit this work when we present the results of our empirical study in Section 5.

For this research project, we used the latest version of the Qualitas Code Corpus, version 20090202 [80]. This version contains 100 different Java programs from a wide variety of backgrounds, including some very well-known programs such as Eclipse and ANTLR. When installed, the corpus takes up 18.75 gigabytes.

In addition to the programs themselves, the corpus contains metadata about the programs, including the name of the program, release data, version of Java used, and the URL used to acquire the program [79].

There are several criteria for programs to be included in the corpus [78]. They have to be written in Java and be distributed in both source and binary form. The compiled versions have to be distributed as .jar files and the programs need to be available freely to anyone.

3 A Discussion of Encapsulation Advice

Many design principles and heuristics have been proposed about the use of encapsulation. Riel [81] and Johnson and Foote [50] as well as Fowler [36] all put forward a number of heuristics about encapsulation. While there is much literature presenting such heuristics, we have found no work which compares and contrasts them to identify similarities and differences between different pieces of advice. In this research, we have collected heuristics relevant to encapsulation and analysed them to identify the main schools of thoughts and major conflicts. In the following section, we look at two contentious issues in the area of encapsulation and explain and compare the different advice and points of views.

3.1 Data Protection

Keeping data `private` to hide it from other parts of the system is a common piece of advice. This principle is part of information hiding [74] and is credited with increasing a system’s maintainability and flexibility.

Imagine that we have a `Vehicle` class with a field describing the vehicle’s `weight`. If, for example, we originally used a `double` to represent the `weight` of a `Vehicle` and then changed the `weight` field to an `int`, code that accesses the `weight` field directly may not compile anymore. In this way, other classes are affected by a trivial change. On the other hand, if we had made the `weight` field `private` and accessed it from other classes through methods such as getters and setters, we could simply modify the methods accessing the `weight` field, which would have little or no effect on the rest of the system. In this way, we have encapsulated the `weight` field and made the system more maintainable and easier to change.

While the issue of hiding data appears relatively clear-cut, there is still some disagreement about the degree to which data should be hidden. Riel [81] says that ‘All data should be hidden within its class’, meaning that data should always be made `private`. In addition, Riel advises ‘Do not change the state of an object without going

through its public interface' [81] meaning that he believes one class' data should never be changed by another class directly but that getters, setters or other methods should always be used.

Fowler and Beck [36] also agree that classes should not access each other's data but state the principle in a slightly softer way. The *Inappropriate Intimacy* code smell occurs when a class spends too much time delving into another class' `private` parts [36]. The phrasing of the code smell suggests that the authors consider that it may be justifiable for a class to access some of the data of another class as long as this is not done too frequently. This is in contrast with Riel's more absolute view.

Robert Cecil Martin softens the rule further and argues that while most data should be kept `private`, making data `public` can be good in some instances [59]. In his argument, Martin distinguishes between normal objects containing data and behaviour and data transfer objects that are designed to hold data and do not have any specific behaviour.

There are very good reasons for keeping the variables in an object `private`. We want to know which functions can manipulate them. We want to protect the invariants of the object. We don't want others to depend on our details. On the other hand there is no good reason to use getters and setters in a data structure. A data structure is simply a packet of data, nothing more.[59]

While most encapsulation advice says that data should be made `private`, it is often unclear what the data should be made `private` to: an object or a class. Riel is a definite advocate of class encapsulation, stating as we have already noted that data should be `private` to a class [81]. In another heuristic, he also advises against the use of `protected` data. Holub agrees and states that 'protected data is an abomination' [43].

This view is contrary to other OO cultures, where the use of `protected` data is actually encouraged. For example, in Objective C `protected` is the default access level for fields. In Smalltalk, there is no choice at all: all data is automatically made `protected`; that is inherited properties are not hidden. This is Smalltalk's only access level and is actually called `private`.

While Riel makes it clear that data should be made `private` to a class, most other advice only says that data should be made `private` but not what it should be `private` to.

3.2 Getters and Setters

While there is a general consensus about the benefits of `private` data, there is a lot of diverse advice about how to program without accessing other classes' data directly. The obvious and very commonly used and taught approach is to create `public` getters (accessors) and setters (mutators) to allow indirect access to `private` data. Rather than exposing the data directly, we create a getter method that will retrieve the data and return it to the caller. We can also create setter methods that allow clients to ask for data to be changed. In this way, the class containing the data has more control over the data. For example, a field could be completely hidden from other classes by not creating a getter or setter for it. A field could also be made read-only if a getter but not a setter was provided.

There are many advocates of getters and setters who generally argue that they increase the maintainability of software by creating a layer of abstraction. Johnson and Foote, for example, advise developers to 'minimize access to variables'; that is, to go through getters and setters rather than accessing variables directly [50]. Using getters and setters is also in agreement with Riel's heuristic telling developers to 'go through the public interface of a class to change its state'. Getters and setters allow the underlying data representation to be changed without affecting the client code. In addition, setters allow a class to add validity checks to ensure that the data cannot get into an inconsistent state. Because of these benefits, advocates of getters and setters argue that if data needs to be accessed, this should be done using getters and setters rather than accessing data directly.

Some go even further in their advice and say that data should always be accessed through getters and setters, even from within the class that contains it. Ken Auer, for example, advocates this approach, arguing that it increases the system's maintainability [5]:

When adding state variables, only refer to them directly in 'getter' or 'setter' methods. ... As additional behavior is added, continue to access state variables only through these getter and setter methods, to allow for simple modifications in the future. Modify any other methods which refer to state variables directly ... so that they instead refer to them indirectly, via these getter and setter methods.

Despite the arguments in favour of getters and setters and their widespread use, many people argue that getters and setters break encapsulation and are bad OO design. Johannes Brodwall for example states that 'I have yet to see an example of use of accessors that did not smell to some extent.' [1]. The problems with getters and setters are summed up by Holub:

Though getter/setter methods are commonplace in Java, they are not particularly object oriented (OO). In fact, they can damage your code's maintainability. Moreover, the presence of numerous getter and setter methods is a red flag that the program isn't necessarily well designed from an OO perspective. ... Since accessors violate the encapsulation principle, you can reasonably argue that a system that heavily or inappropriately uses accessors simply isn't object oriented. ... My experience is that maintainability is inversely proportionate to the amount of data that moves between objects. [43]

One of the main arguments is that getters and setters break encapsulation. They can allow any other part of the system to access the variables and change them. In addition, if a getter method returns a mutable object reference to data, the caller can then change the data directly without going through a setter method. This means that the class containing the data no longer has control over how it is changed.

In addition to this problem, many argue that a need for getters and setters indicates that the data is not placed with related behaviour. Data should always be placed together with the methods that need to access it. Therefore, if a different class continually asks for the `private` data of another class, the data should have been placed with the methods that need it. This is summed up by Riel's heuristic which states that 'If there are too many accessors in the public interface of a class that may be a sign that related data and behaviour is not kept together.' [81]

Those who argue that getters and setters are evil, generally propose simply not to use getters and setters by keeping related data and behaviour together. The *Law of Demeter* [56] completely disallows the use of getters and setters to ensure that related data and behaviour are kept together. It does this by restricting the set of objects that may be used to only objects that are already immediately accessible. Much criticism has been put forward about the *Law of Demeter*. Many argue that it is contradictory because there exist cases where a method can be rewritten to comply with the *Law of Demeter* while still doing the exact same thing. [2]

Tell don't ask was originally proposed as a softer version and a clarification of the *Law of Demeter* [3]. This principle essentially says that clients should tell a class what to do, rather than asking for some internal data so that they can calculate or do something with that data. This is similar to the *Law of Demeter* and again discourages the use of getters and setters by ensuring that related behaviour and data is kept in one place.

While the *Law of Demeter* and *Tell don't ask* may enforce encapsulation if used properly, they are also at odds with some very basic design principles. They tend to create large classes and interfaces, therefore violating the *Interface Segregation Principle*, the *Large Class* code smell [36] and Johnson and Foote's heuristic telling developers to 'split large classes' [50]. In addition, the *Law of Demeter* and *Tell don't ask* potentially violate *Separation of Concerns*: when creating a GUI, they would lead developers to combine the GUI code with the model code to keep related behaviour and data in the same place. However, combining GUI and model code is generally considered bad practice and is a definite violation of *Separation of Concerns*.

Overall, when thinking about using getters and setters, developers face a choice between different design forces. Alistair Cockburn sums up the arguments for and against getters and setters very concisely [26].

Accessor methods:

- Provide a constant interface against persistent interface changes (use them).
- Slow the software, adding complexity to the object's interface (don't use them).
- Provide consistency (use them always).
- Violate abstraction by exposing the object's insides (use them sparingly).
- Are controversial: some people swear by them, some at them.

4 Encapsulation Survey

We designed and conducted a survey in order to find out how developers think about encapsulation. We gave the survey to both computer science students and professional software engineers. In particular, the survey was aimed at revealing whether people were more comfortable with class or object encapsulation. This survey was intended as a pilot study before conducting an empirical investigation into encapsulation practices in actual source code (see Section 5).

The following sections describe the purpose, setup and results of the survey. The detailed information forms, questionnaires and human ethics approval for the survey can be found in Appendix G. The results of this survey

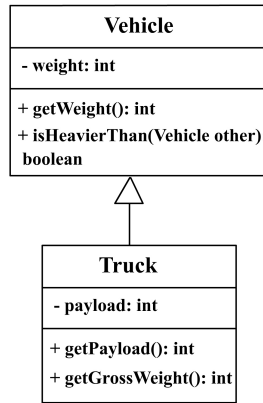


Figure 3: UML diagram of the classes used in the survey

have been summarised in a conference paper, which has been accepted for publication at ICITA'09 (International Conference on Information Technology and Applications). The following section is roughly based on this paper and the paper itself can be found in Appendix C.

4.1 Motivation

From personal experience and from working closely with computer science students, we believe that students learning OO programming using a language like Java or C++ automatically assume that data is **private** to an object and are surprised, and in some cases shocked, when they learn that it is instead **private** to a class. Many of them feel uncomfortable when accessing the **private** data in another object of the same class. It seems that this conflicts with their world view of classes and objects. Over time, however, they appear to adapt more and more to the tools a programming language provides them. They start to access **private** data from another object of the same class in places such as the equals() method in Java. Over time, they weaken their original intuition about what is right and wrong.

Prior to this research, we had only anecdotal evidence to support this theory. Therefore, we decided to conduct a formal survey involving a number of novice programmers and experienced programmers to confirm our hypothesis.

4.2 Participants

We surveyed 34 undergraduate students, 9 postgraduate students and 12 professional developers about their encapsulation practices. We chose to include students in the survey because we expected their intuitions to have had less time to adapt to encapsulation mechanisms provided by programming languages.

We surveyed two undergraduate computer science courses at the University of Canterbury. The first course was a second-year course about computational theory. The students in the course had just completed their first year of computer science, including an introduction to Java and an introduction to data structures, algorithms and software engineering. These students had relatively little programming experience in Java, having not yet completed a substantial programming project.

The second class we surveyed was the third-year software engineering course. These students had all completed the second-year software engineering course which included a substantial group project in Java. They had also learned about various object oriented design principles and were therefore likely to be more aware of design issues than the second-year students.

In addition to surveying undergraduate students, we also surveyed 9 postgraduate students, all of whom were very proficient in Java, and 12 professional software developers who were members of an industry programming users group and routinely used C# or VB.NET as part of their work. These participants were likely to be far more proficient programmers than undergraduate students and also were likely to be more aware of OO design principles.

4.3 Task

We carefully designed the survey to allow us to infer the encapsulation practices and principles of participants rather than asking them directly. We did not want participants to overthink their replies but rather to act as they

would when programming. Two parts of the survey can be seen in Figure 4.

The survey consisted of two main questions that were designed to capture the important differences between object and class encapsulation. For each question, we presented a small Java class containing a few fields and methods. A UML class diagram showing these two classes can be seen in Figure 3. Each of the questions asked developers to complete a new method by choosing between three alternatives. Each alternative completed the method in a way that achieved similar functionality. However, the difference between the options was that some used getters to access fields while others accessed data directly, crossing either the object or class encapsulation boundary.

For each of the two questions, we asked developers to rank the three options they were provided with from *best* to *worst* and to explain the ranking they decided on. The first question focused on the issue of whether or not an object should be able to access the `private` data of another object of the same class. This is allowed in class encapsulation but not in object encapsulation. For the purposes of this question, we introduced a Java class `Vehicle` which had a `private` field `weight` and a getter method for the `weight` field. Participants were then given three options for the completion of the `isHeavierThan()` method which compared the `weight` of one `Vehicle` object to that of another `Vehicle`.

The second question focused on the issue of whether or not an object should be able to access `private` data declared in superclasses. This is allowed in object encapsulation but not in class encapsulation. For this question, we introduced a class `Truck` which was a subclass of `Vehicle` and had an additional field called `payload` which stores the maximum load a truck is allowed to carry. The `Truck` class also contained a getter method for the `payload` field. Participants were then given three options to complete a method called `getGrossWeight()` which returned the sum of the truck's `weight` and `payload`. Again, the only difference between the three options they were given was that some used getters while others accessed data directly.

In addition to the two main questions, to test the competence of the participants, we included two very simple coding exercises asking participants to write a `toString()` method for the `Vehicle` and `Truck` class. These questions enabled us to eliminate one participant who clearly did not have sufficient knowledge to make informed judgements about encapsulation.

For professional software developers, we also included a question about their previous programming experience, including their first programming language and the amount of time they had used C# or VB.Net. We translated the survey to C#, making sure that the semantics of the code in the questions were not affected.

4.4 Results

From the survey results, we deduced whether respondents were using class encapsulation or object encapsulation. These results can be seen in Figure 5.

Our results from the student survey clearly confirmed our hypothesis that for novice programmers object encapsulation is much more intuitive than class encapsulation. The students we surveyed could be divided into four major groups given their responses to the questionnaire:

- Students who used getters rather than accessing data directly;
- Students who practised object encapsulation;
- Students who accessed data directly rather than using getters; and
- Students who did not mind whether getters were used or data was accessed directly as long as the approach used was consistent.

More than half the students (59 percent) preferred using getters to accessing data directly. This is not surprising since they have been taught in a number of courses that getters make a system more maintainable. They commented that using getters was better style, safer and made the system more maintainable and also said that getter methods encapsulate `private` data.

The second largest group, at about 24 percent, was the group who practised object encapsulation. They were all happy to directly access `private` data in a superclass but did not want to access `private` data from another object of the same class. From their comments, it became clear that members of this group incorrectly believed that this was what Java allowed. Some participants commented that accessing `private` data in another object of the same class was not possible because that data was `private`.

The remaining groups were both small, with about 12 percent of students preferring to always access data directly rather than using getters. They usually commented that this was more efficient. The last group of students, at about 6 percent, was simply concerned with keeping the coding approach as consistent as possible.

Now we want to complete the code in the `getGrossWeight()` method of the `Truck` class. This method is supposed to calculate the gross weight of the truck, i.e. the weight of the truck plus the maximum load it can carry. Here are several different ways in which this method could be written:

```

Option 1
public int getGrossWeight() {
    return weight + payload;
}

Option 2
public int getGrossWeight() {
    return getWeight() + payload;
}

Option 3
public int getGrossWeight() {
    return getWeight() + getPayload();
}

```

8) Which of the above options do you consider the best? Rank the three options, from best to worst.

1 _____

2 _____

3 _____

Give reasons for the way you ranked the options:

Now we want to complete the code for the `isHeavierThan(Vehicle other)` method of the `Vehicle` class.

Here are several ways in which we could complete this method:

```

Option 1
public boolean isHeavierThan(Vehicle other) {
    return this.weight > other.weight;
}

Option 2
public boolean isHeavierThan(Vehicle other) {
    return this.weight > other.getWeight();
}

Option 3
public boolean isHeavierThan(Vehicle other) {
    return this.getWeight() > other.getWeight();
}

```

6) Which of the above options do you consider the best? Rank the three options, from best to worst.

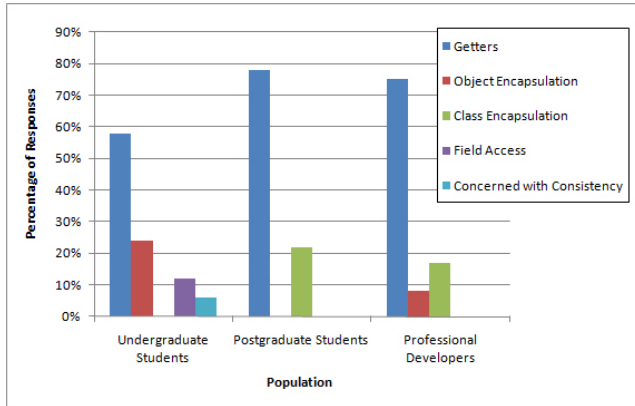
1 _____

2 _____

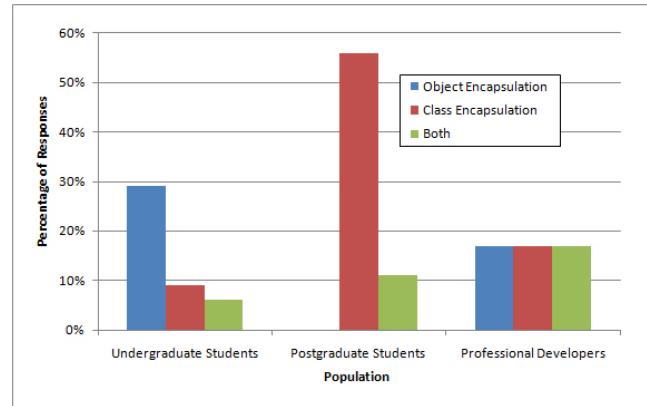
3 _____

Give reasons for the way you ranked the options:

Figure 4: An extract from the Java version of the encapsulation survey



(a) Initial classification of survey responses



(b) Encapsulation tendencies for the three populations

Figure 5: Encapsulation Survey Results

Notably, there were no students who practised explicit pure class encapsulation; that is no one thought that it was good practice to access the `private` data in another object of the same class but not the `private` data in a superclass.

The largest group of students consistently used getters to access data, giving no direct indication of their encapsulation preference. However, the study was designed to provide an indication of their preferences in this situation by making them rank their second and third preferences. We also looked more closely at the comments from the students who used getters to determine if they were aware of the issue of object and class encapsulation. Most responses showed no tendency towards either type of encapsulation. Two responses clearly showed object encapsulation thinking, with students commenting (incorrectly) that it was not possible to access the `private` field of another object of the same class. Another three responses showed traces of class encapsulation, suggesting that these students were aware of Java’s approach to encapsulation. They usually commented (correctly) that a `private` field in a superclass could not be accessed directly. This is not a surprising response since they have been taught this in class. The remaining two students occupied an uneasy middle ground, showing tendencies towards both class and object encapsulation. Clearly, these two students were confused about how to practise encapsulation in Java.

We wanted to compare the way novice programmers think to the way more experienced and professional software engineers think about encapsulation. Therefore, we surveyed nine postgraduate students, all of whom were very proficient in Java, and twelve professional software developers who were experienced .NET developers.

Interestingly, we found that none of the postgraduate students used object encapsulation, but two used class encapsulation (22 percent). They commented (correctly) that accessing the fields of another object of the same class directly was simple and valid while accessing the `private` fields in a superclass was not allowed. This clearly shows that they think differently from novice programmers; their thinking is aligned with Java’s encapsulation mechanisms. The remaining 88 percent preferred always using getters to support encapsulation.

We again had a closer look at the surveys of the postgraduate students who used getters to see if we could infer more about their way of thinking. Three of the seven respondents who used getters showed definite class encapsulation tendencies, while another one showed tendencies both ways, and appeared to be confused about encapsulation in Java.

We saw a similar effect when we surveyed twelve professional .NET developers. The largest group, at 75 percent, again always used getters. Two respondents (17 percent) used pure class encapsulation, clearly demonstrating that they were aware of what was valid in C#. Both commented that accessing `private` fields in a superclass was not valid and would not compile. One developer (8 percent) with 5 years experience using C# still believed that object encapsulation was correct.

A closer look at the surveys of developers who used getters showed that even some of the professional developers were not completely comfortable with encapsulation in C#. One developer showed object encapsulation tendencies stating (incorrectly) that accessing `private` fields of another object of the same class would cause a compile-time error. Two more developers showed both object and class encapsulation tendencies in their survey and appeared generally unsure about what was allowed and what was not.

The results from the student survey clearly indicate that novice programmers find object encapsulation more intuitive than class encapsulation. More than a quarter of the students we surveyed, with as much as two years of programming experience, still believed that Java essentially supports object encapsulation. In addition, no students were comfortable using what Java provides: class encapsulation. Some students showed signs of being aware of encapsulation mechanisms in Java but no one wanted to use them. This is a significant result because it shows that novice programmers are uncomfortable with the encapsulation mechanisms provided by many modern programming languages including Java and C#. Object encapsulation, not class encapsulation, appears to make sense to them.

Even some postgraduate students and professional software engineers, all of whom were proficient in either Java or C#, showed signs of unease and confusion about the encapsulation mechanisms provided. Some of them did not appear to be entirely sure about what was allowed despite years of programming experience. However, there was a clear sign that a number of them had adapted to what the programming language they were using provided them with, because around 20 percent used class encapsulation.

Overall, we believe that the results from our survey support our suspicion that class encapsulation as provided by many modern programming languages is not what novice programmers expect and can confuse even experienced developers.

5 Analysing Encapsulation in Source Code

The survey provided us with evidence that a deeper investigation into encapsulation was warranted. To this end, we decided to conduct an empirical investigation into encapsulation practices in a substantial corpus of Java programs. One aim of the empirical investigation was to determine whether the encapsulation practices uncovered by the survey were translated into practice when actually coding; it is possible that some developers will think about encapsulation in a particular way but act differently when they are coding because it may be faster and easier to do.

We created a program which analyses Java software and extracts relevant information from it. This program uses JST to build semantic models of Java programs. We used the program to collect encapsulation information from 33 different Java programs from the Qualitas Code Corpus as well as 11 student programs. This is an unusually large dataset for metrics research of this nature.

We carefully designed our analysis program so that it would answer the same questions as the encapsulation survey, allowing us to enrich our understanding of encapsulation practices. Our program therefore attempts to measure the relative frequency of use of class and object encapsulation, and more generally characterises encapsulation practices.

This study builds on results uncovered by Tempero in a recent empirical study (mentioned above) about the use of fields in Java [89]. Tempero measured how often different access modifiers are used and how frequently fields are accessed. He too studied programs from the Qualitas Code Corpus, analysing all 100 programs in the corpus and thus using a larger dataset than this study. However, Tempero's study is much narrower in scope, looking only at the level of exposure of fields, and does not address the different encapsulation boundaries we have identified. In fact, Tempero implicitly assumes class encapsulation in his study. Another limitation of Tempero's study is the use of bytecode analysis to extract information about programs; in this study we use a semantic model which is a more accurate representation of the program.

Section 5.1 explains in detail the specific data extracted by our analysis program. We also give more detail about the Qualitas Code Corpus which was used as a source of real-world software in Section 5.2 and we explain how JST was used as a semantic model in this research in Section 5.3. Finally, before presenting the results of our investigation in Section 5.6, we describe the design of our program and the experimental setup in sections 5.4 and 5.5.

The results from this part of our work were summarised in a paper submitted to the Australian Software Engineering Conference 2010. A copy of the paper can be found in Appendix D. Some parts of the following section have been taken from the paper.

5.1 Encapsulation Analysis Tool

The program we built to analyse encapsulation collects comprehensive data. We chose to focus our attention on encapsulation of data (as opposed to methods), which is more emphatically stressed by OO design guidelines and more readily grasped by programmers, so it is likely to support more definitive conclusions.

Our tool measures two aspects of a program: the levels of protection accorded to fields, and the ways in which fields are actually accessed. This allows us to tell, for example, if a field has been given wider scope than is used in practice, such as when a package-accessible field is only ever used locally in its class.

To characterise protection levels, we count the number of fields in a program with `public`, `package`, `protected` and `private` access. These numbers give a good overview of how rigorously data is hidden from the outside world. We distinguish between `final` fields (or constants) and non-final fields because exposing constants is usually considered less serious than exposing fields whose values can be changed.

Characterising actual accesses to fields is a little more complex. However, this information is vital when determining the strength of encapsulation, since one could argue that a `public` field which is accessed rarely may break encapsulation to a lesser degree than a `public` field which is accessed frequently.

Our encapsulation analysis tool accumulates the number of accesses to `public`, `package`, `protected` and `private` fields. In addition, it finds the least and most frequently accessed fields in the whole program. This information could easily show up a coding style like that advocated by Ken Auer, who suggests always using getters and setters to access fields, even from within the same class. This would lead to a very small number of accesses for each field.

The analysis tool also counts the number of accesses that originate inside and outside the object that contains the field, and the number of accesses that originate inside or outside the class that defines the field. This allows us to count the number of accesses that cross both types of encapsulation boundary.

Accesses from outside a class that defines a field are easy to find. However, a more sophisticated approach is required to determine if an access comes from outside an object. Because we perform static analysis of source code, objects, which are a runtime concept, do not yet exist and it is virtually impossible to determine precisely whether a reference refers to the same object as the one doing the accessing. We instead employ a simple heuristic which will work correctly in the vast majority of cases:

- If the access is of the form *fieldName* (without a qualifier), the access comes from within the same object.
- If the access is of the form *this.fieldName* or *super.fieldName*, the access comes from within the same object.
- In all other cases, the access comes from a different object.

While this strategy works well for the vast majority of cases, there are exceptions. The most common occurs with inner classes, which can access fields in the outer class. This syntax appears to be an access from within an object when it is really an access from a different (inner) object. However, we decided that this situation was sufficiently rare (and also outside our simple formulation of the concept of an object encapsulation boundary) that our straightforward heuristic would provide an acceptable approximation for a first study. We plan to refine this aspect of our instrument in later versions.

Our analysis tool has the ability to analyse the data it has collected and report on the extent to which a program uses object or class encapsulation. It does this by looking at the number of accesses that cross the object and class boundaries. In many cases, we expect that a mixture of the two will be employed. This information was deliberately gathered in order to allow us to directly compare the results of the encapsulation survey with the real-world data.

5.2 The Qualitas Code Corpus

We used the latest version of the Qualitas Code Corpus as a source of real-world programs to analyse. The Java version in which these programs were written varies from Java 1.1 to Java 1.6. Our Java parser is generated from the grammar for version 1.6. In many cases, Java is sufficiently backwards-compatible that our parser and JST can handle older source code, but some programs contain syntax that has been made illegal by changes in the Java language. The most common syntax error, which prevents about 20 programs from compiling, is caused by the introduction of `enum` as a keyword. Early versions of Java provided the unfortunately named `Enumerator` class which was later deprecated in favour of `Iterator`, and local variables of the `Enumerator` type were commonly named `enum`, leading to the name clash. The late introduction of the `assert` keyword produced a similarly widespread problem. We chose to exclude these programs from our experiments because they are no longer correct Java.

In addition to these issues, a couple of programs, including Java parsers and parser generators, would not compile since they contained deliberately wrong Java syntax in test files. We decided to exclude these programs from our experiment. They were never intended to represent well-structured software and will therefore not yield any useful information about encapsulation practices.

In the end, we excluded the 21 programs with parse errors, leaving 79 programs which contained only correct syntax. However, only 33 of these programs could be correctly processed by the available version of JST.

5.3 The Use of JST

JST forms an important part of our encapsulation analysis program. JST is very complex, with a large number of classes each representing a specific semantic concept in Java, such as packages, methods and classes. A UML class diagram of (an earlier version of) JST can be seen in Figure 15 in Appendix A. The latest version of JST adds modelling capabilities for the concept of Java generics, making it significantly more complex than the older version, with more than 100 classes.

To understand what concept each class in JST represents and what relationships exist between concepts, an in-depth knowledge of Java semantics is essential. JST was under active development in order to support Java generics at the time of this work; this greatly increased the level of challenge in using its API and understanding its structure. This project is to date the only client of the latest version of JST and so has formed an important part of the debugging and quality assurance process of JST. Many bugs were discovered as a direct result of this work and the quality and completeness of JST was significantly improved.

While working with JST, we wrote a detailed guide for future researchers explaining how to use JST. This guide explains the important classes of the semantic model and shows how to write a *Visitor* to walk through and extract information from the model. A copy of the guide can be found in Appendix E.

While JST looks very large and complex at first sight, writing a program to extract information from JST is relatively simple. Nevertheless, it does require knowledge about the internal structure of JST. Fortunately, most clients are only interested in a particular part of the model, reducing the portion of the API that must be comprehended.

The *Visitor* design pattern [37] is incorporated into JST as the mechanism for conveniently extracting information. According to the Gang of Four, the Visitor design pattern is used to

represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates. [37]

This means that Visitor cleanly separates an object structure (the JST model of a Java program) from the operations that need to be performed on it (the metrics and analysis applications). This separation is very convenient, as it groups related client operations into cohesive classes. This means that JST is not altered by the client code and vice versa.

The Visitor design pattern specifies that a visitor should contain a method to visit each part of the object structure. In terms of JST, this means that visitors should contain a method to visit each particular semantic concept, including classes, interfaces, and variable declarations. Of course, it is impractical to write a method for each semantic concept if the client is only actually interested in inspecting one or two of them. For this reason, JST contains a visitor called `CompositionVisitor` which contains methods to visit all parts of the model. These methods include no specific logic other than the code required to visit all elements in the JST model in a logical order. Clients can simply subclass `CompositionVisitor` and override the methods in which they are interested, adding the operations they want to perform. The other methods are simply inherited from the superclass. This greatly reduces the amount of code that needs to be written in the client visitor.

For more technical detail about writing visitors for JST, see the guide in Appendix E.

5.4 Program Design

Our encapsulation analysis tool is made up of four main parts: a simplified program model, a visitor and builder that construct the model, metrics calculators that extract information from the model, and writers that write out the calculation results. A UML class diagram of the program can be seen in Figure 16 in Appendix A.

Rather than extracting information straight from JST, we have chosen to build a simpler model that filters out information that is not relevant to encapsulation. This model contains classes such as `Program`, `Member` and `Field` which replicate information from the JST model of the program but provide a simpler and more convenient interface. Thus, this part of the program effectively acts as a Facade [37], hiding the more complex JST interface. This facade is constructed by a `JSTModelVisitor` which walks the model of JST and uses the `Builder` class to assemble the model.

`MetricCalculator` and its subclasses implement specific metrics that can be calculated by extracting information from the model. They each have a specific strategy for extracting this information from the model, meaning that the metrics can be calculated at different levels of aggregation, for example at the program-level or class-level.

The calculator classes pass the results they have calculated on to `ResultsWriter` and its subclasses which write the results out to file.

Program	Program Version	Description	Release Date
Cobertura	1.9	Code coverage tool	05/06/2007
Display Tag Library	1.1	JSP tag library	02/12/2006
EMMA: a free Java code coverage tool	2.0.5312	Code coverage tool	12/06/2005
Fit Java	1.1	Automated testing tool	07/04/2004
Galleon	1.8.0	TiVo media server	30/10/2005
Gantt Project	1.11.1	Gantt chart drawing	14/05/2005
HTMLUnit	1.8	Web testing tools	17/02/2006
Informa	0.6.5	RSS library and RDF tools	30/09/2005
Ivata Groupware	0.11.3	Groupware/ exchange/ intranet system	10/10/2005
Jag	5.0.1	J2EE application generator	13/10/2005
Java Assembling Language	0.10	Class file editing tool	23/05/2006
Jasper Reports	1.1.0	Reporting tool	01/11/2005
JChemPaint	2.0.12	Editor for 2D molecular structures	21/12/2005
JFreeChart	1.0.1	Charting and reporting tool	27/01/2006
JGraphPad	5.10.0.2	Graph drawing application	09/11/2006
JMoney	0.4.4	Personal finance manager	29/09/2003
Java Plug-in Framework	1.0.2	Plug-in infrastructure library	19/05/2007
JSP Wiki	2.2.33	Wiki engine	07/09/2005
Jung	1.7.6	Graph drawing	29/01/2007
2006-03-01 JUnit	4.5	Unit testing framework	08/08/2008
Log4J	1.2.13	Logging tool	21/09/2006
OSCache	2.3	Cache solution	06/03/2006
PicoContainer	1.3	Inversion of control container	18/03/2007
Pooka	1.1-060227	Email client	27/02/2007
Proguard	3.6	Obfuscator	14/05/2006
Quartz	1.5.2	Job scheduler	03/03/2006
Quickserver	1.4.7	Network server	01/03/2006
Quilt	0.6-a-5	Code coverage tool	20/10/2003
Roller	2.1.1	Server based weblogging software and web application	15/03/2006
Sablecc	3.1	Compiler/interpreter generating framework	29/09/2005
Sunflow	0.07.2	Render engine	08/02/2007
Trove	1.1b5	Collection library	27/12/2005
Velocity	1.5	Java template engine	20/02/2007

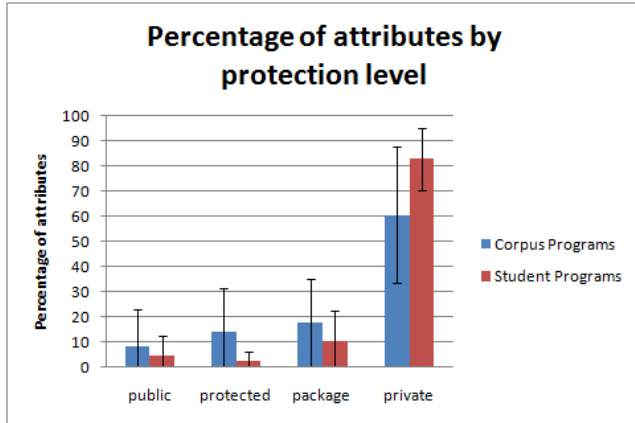
Table 1: Corpus Programs used for this experiment

Our encapsulation analysis tool has been designed to be an extensible framework to which other metrics can be added in the future. Adding a new metric to be calculated is easy: a new `MetricCalculator` subclass simply has to be added to the design. Because of its extensibility, we expect that our analysis program will form the starting point for the creation of future analysis tools.

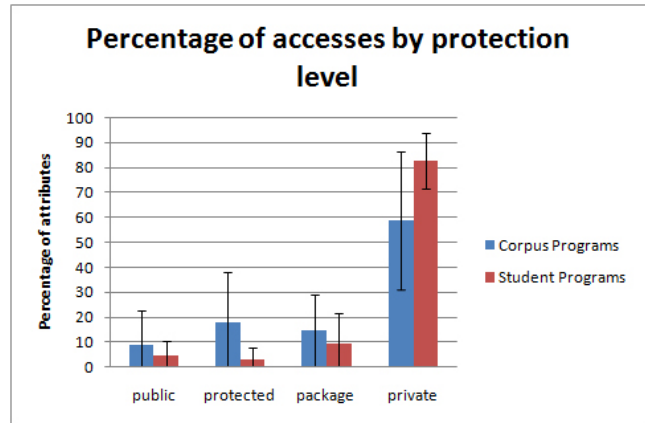
5.5 Experimental Setup

For this experiment, we analysed 33 programs from the corpus; these were the ones for which the complete source code could be processed by the current version of our tools without difficulty. More detail about these 33 programs can be seen in Table 1. We also analysed an additional 11 student programs to see if there was a similar discrepancy between students' and professionals' encapsulation practices to that found in the survey. The student programs were each produced by a group of six to seven second-year software engineering students as part of a semester-long project for real clients and were relatively similar to each other in scope and functionality.

The actual conducting of the experiment was a matter of simply running our tool over each of the programs and collating the data for later analysis.



(a) Use of protection levels in real and student programs



(b) Accesses in real and student programs by protection level

Figure 6: Protection levels and accesses in real and student programs

	Average (%)	Minimum (%)	Maximum (%)	Standard Deviation
Public	8.8	0	63.6	13.9
Protected	18.0	0	61.9	20.0
Package	14.6	0	69.7	14.6
Private	58.6	1.9	99.6	27.6

Table 2: Protection level statistics for declarations in corpus programs

5.6 Results

In the 33 corpus programs, the number of fields ranged from 69 to 2159, while the number of accesses to fields ranged from 355 to 10818. In the 11 student programs, the number of fields ranged from 55 to 469, while the number of accesses to fields ranged from 208 to 1973.

Figure 6a shows the relative numbers of different protection levels used in corpus and student programs; Figure 6b shows the relative numbers of accesses to fields with those different protection levels. Unsurprisingly, the two graphs have very similar shapes.

Clearly, **private** is the most frequently declared and the most heavily accessed protection level. This tendency is more pronounced in student programs than in corpus programs, where 40 percent of fields are not **private**. This suggests that student programs tend to be more tightly encapsulated.

It is also interesting to note that students rarely declared **protected** fields compared to corpus programs and that corpus programs tended to access **protected** fields somewhat more frequently than other types.

The above figures also show the variability of the data using error bars that span one standard deviation. We found a great diversity of encapsulation practices in the corpus programs but relatively consistent practices in the student programs, as can be seen in Table 2 and Table 3. This no doubt reflects the greater variety of domains, purposes and scales of the corpus programs as well as the diversity of the developers.

A notable characteristic of this data is the very high standard deviations of the corpus programs' use of protection

	Average (%)	Minimum (%)	Maximum (%)	Standard Deviation
Public	4.6	0	20.5	5.8
Protected	3.0	0	13.4	4.9
Package	9.6	0.2	42	11.8
Private	82.8	60.1	94.5	11.0

Table 3: Protection level statistics for declarations in student programs

	Category of Access	Percentage
1	Same object, same class	82.6
2	Same object, superclass	6.6
3	Different object, same class	2.7
4	Different object, superclass	0.2
5	Different object, different class	7.8

Table 4: Percentage of accesses by category in corpus programs

levels. **Private** data in particular spans a range from virtually no use in *Fit Java* to almost exclusive use in *HTML Unit*. This is evidence that encapsulation practices in industry are extremely inconsistent.

We found similar levels of variation in the number of accesses to fields. Of particular note was:

- **Public** fields, unsurprisingly, are used quite heavily from outside the class that declared them (57.5 percent for corpus programs and 40.8 percent for student programs), and are also used heavily internally.
- In Java, the **protected** access modifier gives access rights to subclasses, as well as to other classes in the same package. We found that in corpus programs subclass access is used much more commonly (27.9 percent) than same-package access (5.6 percent); the remaining accesses are from within the declaring class. Student programs, however, revealed a different picture. Out of the eight programs that used the **protected** protection level, two used it as **package** access, four used it as **private** access, and two used it as subclass access. This suggests a considerable degree of confusion among students regarding Java’s **protected** access mechanism.
- **Package** fields are much less commonly accessed from outside the class in which they were declared (averaging 20.1 percent). In Java, **package access** is the default protection level, and it seems likely that this level of access has been granted in many cases by developers forgetting to specify tighter access. This is the case in student programs in particular, where 55 percent of systems never accessed **package** fields from outside the declaring class.
- Because Java uses class encapsulation, **private** fields can be accessed from other objects of the same class. We found that almost all corpus and student programs made some use of this; however, the average percentage of accesses to **private** fields from other objects was very low (3.0 percent and 1.3 percent respectively).

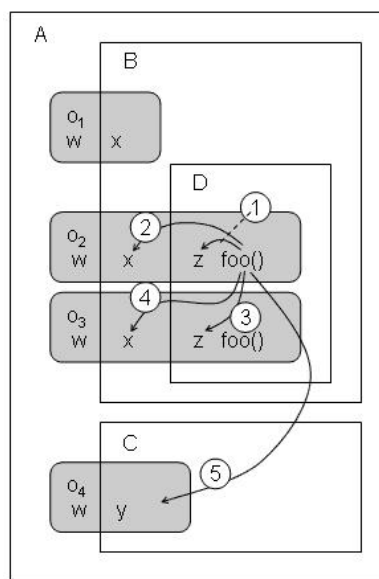


Figure 7: An overview of access categories

	Category of Access	Percentage
1	Same object, same class	93.7
2	Same object, superclass	0.3
3	Different object, same class	1.5
4	Different object, superclass	0.0
5	Different object, different class	4.5

Table 5: Percentage of accesses by category in student programs

Figure 7 shows the main categories of access we measured, numbered 1 through 5. Table 4 and Table 5 name the access types and show what percentage of all accesses belonged to each category in the corpus and student programs.

Unsurprisingly, in both populations the dominant category of access is within the same object and class (category 1). This category does not reveal anything about encapsulation boundary preferences as it crosses no encapsulation boundaries. Categories 4 and 5 similarly do not yield information about encapsulation boundary preferences as the accesses in these categories cross both kinds of boundary. Interestingly, category 4 accesses are much less frequent than category 5, suggesting that developers are more averse to accessing superclass data than data in a completely unrelated class.

Category 2 accesses cross the class boundary but not the object boundary, indicating the use of object encapsulation. Category 3 is the inverse, indicating the use of class encapsulation.

In corpus programs, when an encapsulation boundary preference is evident, object encapsulation (6.6 percent) is used more than twice as much as class encapsulation (2.7 percent). This is consistent with our expectations and earlier survey results showing that object encapsulation is more intuitive.

In student programs, the number of accesses in Category 2 and Category 3 suggest the opposite result: class encapsulation appears to be preferred. This conflicts with our findings from the survey where object encapsulation was overwhelmingly preferred by students. This difference could be explained by the fact that the scenario in the survey was a lot simpler, the students’ programs show evidence of general confusion about encapsulation mechanisms in Java, and the total number of accesses in Category 2 and Category 3 was very low.

The great majority of accesses in both populations either cross no encapsulation boundary or both kinds. No systems measured used either type of encapsulation exclusively, although some showed a strong preference for one or the other. The percentage of accesses crossing a class encapsulation boundary ranged from 0.3 percent to 39.9 percent for corpus programs and 1.3 percent to 6.1 percent for student programs. The percentage of accesses crossing an object encapsulation boundary ranged from 2.0 percent to 40.2 percent for corpus programs and 3.8 percent to 15.4 percent for student programs.

Some corpus programs were notable for having large numbers of accesses from outside both the class and the object. For example, the highest number of class and object boundary crossings (39.9 percent and 40.2 percent) occurred in one program: *Fit Java*. This indicates not only that the data is barely protected and encapsulation is very loose, but also that the data is poorly distributed amongst the classes because the program’s behaviour is not located with the data on which it acts.

Our data also suggests that when `protected` access is used, it tends to be used for object encapsulation. For both populations, the access to `protected` fields from outside the object was less common (8.3 percent for corpus programs and 4.5 percent for student programs) than for fields with other protection levels.

In this section, we presented only the most interesting and relevant findings. More detailed data extracted from the programs can be found in Appendix B.

5.7 Discussion

We measured 33 corpus programs and 11 student programs and found incoherent encapsulation practices, not only between programs but within programs. This is consistent with the findings from an earlier study conducted by Tempero [89]. Neither class nor object encapsulation was practised consistently in any of the programs. It also appears that common advice to make fields `private` was not consistently followed; while `private` was the most popular protection level in most systems, on average only 58 percent, and in the worst case 1.9 percent, of fields in corpus programs were `private`. Some corpus programs used mainly `public` data, with their encapsulation consequently being extremely weak.

Student programs were more tightly and more consistently encapsulated, no doubt because students had recently been instructed to program this way. Even so, students broke the rule to make data `private` in 17 percent of

declarations. This could be due to laziness (not wanting to write getters and setters to access data) or simply a lack of understanding of encapsulation.

Protection levels of fields can be used to enforce particular encapsulation practices but even in the absence of these protections the same encapsulation boundaries can be respected by simply choosing not to access the fields. We found, however, that accessible fields did tend to be accessed. For example, approximately half the accesses to `public` fields came from outside the declaring class. Similarly, accesses to `protected` fields came from subclasses approximately 30 percent of the time.

The dominant encapsulation practice is to access fields from within the object and class that declares it. This is the intersection encapsulation approach that we described earlier; it is likely to be an approach used in response to uncertainty over the appropriate use of encapsulation boundaries.

In those corpus programs which did exhibit a preference for object or class encapsulation, more than twice as many accesses indicated object encapsulation as opposed to class encapsulation. This adds weight to the findings of our previous study, which suggested that object encapsulation is more intuitive.

Class encapsulation allows objects of the same class to access each other's `private` fields, but this ability is rarely used in practice. In corpus programs only 3 percent of accesses to `private` fields come from outside the object. This suggests that there is a certain level of uneasiness among developers regarding this access mechanism.

Java defaults to `package` access. This default supports neither object nor class encapsulation. It appears that in many cases omission of the access modifier is in fact unintended, particularly in student programs, where `package` fields are mainly used as if they were `private`.

In C++ and C# it is possible to grant access to subclasses exclusively but in Java `protected` access also grants `package` access. In practice, however, `protected` fields tend not to be accessed outside the class hierarchy and are often used to support object encapsulation in corpus programs. Java's `protected` mechanism appears to cause confusion among students, leading to inconsistent use of this protection level in student programs.

6 Refactoring Tools

After discovering that the encapsulation practices employed by professional software engineers are far from ideal, we decided to create software to automatically refactor code to improve its consistency and ensure that it conforms to a single encapsulation policy such as class or object encapsulation. In this section, we describe how JST parse trees can be used to create refactoring tools. We also describe one specific refactoring tool we created as a proof of concept and suggests other refactoring tools that could be produced in the future.

6.1 JST Parse Trees

JST reads in parse trees created by a Java parser. A Java parser, as generated using the parser generator Yakyacc [45], outputs XML files that can be read in by JST to create a semantic model of the program.

Most of the time when working with JST, little or no knowledge of the original parse tree structure is required and instead information can be extracted directly from the JST model. However, JST keeps a copy of the parse trees used to generate the model. In this way, the JST model of the program essentially forms the semantic layer of the analysis tool from which semantic information can be easily extracted, while the parse trees form the syntactic layer of the tool.

We used JST parse trees to create refactoring tools. Once the model of a program has been read in by JST and a semantic model has been created, the encapsulation in the program can be easily analysed and evaluated. If encapsulation problems are found, these can be solved by simply modifying the parse trees that correspond to the program. When all necessary changes have been made, the parse trees can be used to re-generate the code.

Figure 8 shows part of a JST parse tree. This part of the parse tree declares a field; specifically it corresponds to the code `private int i;`. The original code is preserved in the tokens or leaf nodes of the parse tree. The tokens are chained together in their original order, each token pointing to the next token. The token chain also contains additional whitespace tokens which are not connected to any other parts of the parse tree. It is easy to re-generate code from the parse trees by simply starting at the first token and walking along the token chain.

In order to document the structure of JST parse trees for future researchers and describe how they can be used for code re-generation, we wrote a guide about JST parse trees which can be found in Appendix F. This guide contains a more detailed description of the JST parse tree structure and more technical detail concerning the implementation of code re-generation tools.

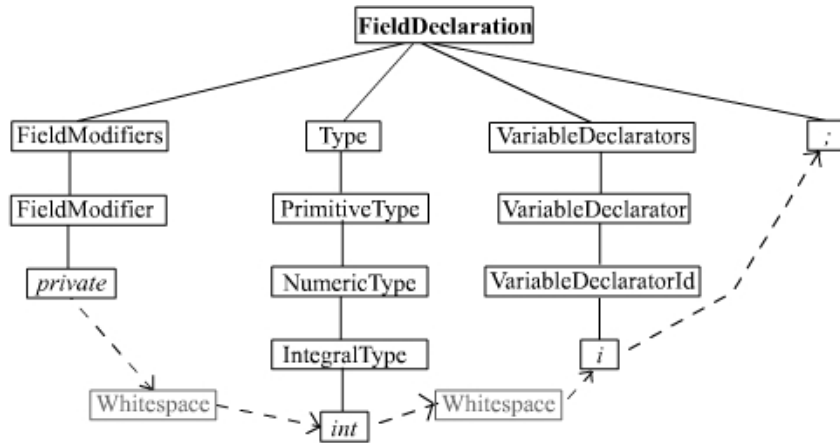


Figure 8: Part of a JST parse tree

6.2 Encapsulation Refactoring Tools

To show that JST can be used for code re-generation and refactoring tools, we created a simple refactoring tool that tightens access modifiers as much as possible given existing accesses to improve the encapsulation of a program. For each field, the tool checks if its protection level can be tightened, for example from `protected` to `package` access. It does this by checking where the accesses to the field come from and whether these accesses would all still be legal at a tighter protection level. If this is the case, it updates the protection level of the field; otherwise it does nothing.

Modifying the parse trees of a program to change the protection level of a field is relatively simple because only a small part of the parse tree is involved (specifically the part that can be seen in Figure 8). However, adding or removing `package` access is a special case because it is specified by omission of the access modifier and therefore requires adding and removing parts of the parse tree. In addition to this, modifying the parse tree is complicated by the fact that the chain of tokens need to be kept up-to-date.

Another complicating factor in the implementation of our refactoring tool is the fact that Java allows several fields to be declared in a single statement, for example `protected int i, j, k;`. However, when looking at tightening the encapsulation of fields, we need to consider each field in isolation. If several fields are declared in one statement but the protection level can be tightened for only one of those fields, the field declarations need to be divided into separate statements, for example `private int i; protected int j; protected int k;`. This involves the creation of parse tree chunks similar to that shown in Figure 8. To simplify the tightening of access levels, our refactoring tool first separates out all field declarations, so that each field is declared in a separate statement, before tightening protection levels.

The initial refactoring tool we created is relatively simple but can nevertheless have a significant impact on the encapsulation in software. If a program already correctly uses all protection levels, this tool will have no influence on the encapsulation of the program because it will not be able to tighten the protection levels of any fields. On the other hand, it is capable of tightening access where a field is less carefully protected than it could be. Tightening access in such cases will remove the temptation of accessing the field from another protection level in the future. Our empirical results show that there are a number of fields which could be protected more carefully. For example, a number of `package` fields in software are used as if they were `private`, and access could be tightened in such cases.

We plan to develop more sophisticated refactoring tools in the future. For example, we want to develop a tool that allows developers to choose particular encapsulation policies such as object encapsulation, class encapsulation and intersection encapsulation and refactors the program to be consistent with this approach. This will involve adding getters and setters to classes and changing direct accesses to a class' fields to use getters and setters instead.

6.3 Limitations

The refactoring tool has a problem caused by a bug in JST, which is beyond the scope of this project to fix. The effect of the bug is that comments and whitespace tokens are not properly added to the parse trees, although these tokens were originally intended to be included in the token chain. Comment tokens are completely omitted from

the parse trees, and all whitespace tokens (including tabs and newlines) are replaced by single spaces. This means that re-generating code strips out comments and whitespace.

We expect these problems to be solved soon so that the next JST parse tree version will contain the correct spacing and comments. The way in which our current refactoring tool is written means that it should continue to work correctly with the new version of the parse trees.

7 A Visualisation of Encapsulation in Software

As our results from real world software show, encapsulation is commonly weak in industry software. This is likely caused by the conflicting advice about encapsulation given to developers and the general confusion among developers about encapsulation mechanisms provided by programming languages. The situation is compounded by the fact that encapsulation breaches can be difficult to spot when only looking at the source code.

To find an encapsulation breach, not only is it necessary to look at the access modifiers given to fields, but also at where these fields are accessed. Furthermore, it is very difficult to get a quick overview of the strength of encapsulation in a system by looking at the code; a number of different source files would have to be analysed for this purpose.

Therefore, we have designed and implemented a VRML visualisation of software which not only gives a quick overview of the encapsulation strength in software, but can also be used to find specific encapsulation problems. In addition, it can show whether object or class encapsulation is more prevalent.

7.1 Design

We wanted our visualisation to be easy and quick to use to find encapsulation problems so that software developers could use it as part of their day-to-day routine. We envisage that developers could look at our visualisation after finishing a particular part of the software to see if there are any encapsulation problems that should be looked at more closely. We also imagine that our visualisation could be used as part of code reviews to find code that needs to be refactored.

We wanted our visualisation to be able to work with as many different pieces of software as possible. Therefore, we decided to use an XML file as input to the visualisation. This allows our visualisation to fit in with the existing XML pipeline that has been used in previous work [19, 46]. We also wrote a program which uses JST to analyse and extract relevant information from Java programs and to write the results out as an XML file. In this way, we can analyse any Java program to produce an XML file which can then be read in by our visualisation. By decoupling the visualisation from the program and the programming language itself, we have gained the flexibility to create visualisations of programs written in any object oriented programming language, provided that a suitable XML file is supplied.

Our visualisation shows a number of different encapsulation aspects. For each field in the system, users can see not only how often it is accessed, but also where in the system the accesses come from. Classes are represented as gray boxes and fields as cylinders that grow out of the bottom and the top of the class they are declared in. To make it easy to see which box represents which class, all class boxes are labeled with the name of the class they represent. Cylinders are also labeled with the name of the field they represent, although these labels are hidden by default so that they do not clutter the visualisation. They can be made visible by hovering over the field cylinder with the mouse.

While the metaphor we chose for our visualisation is very simple, we believe that it works well for the particular domain of our visualisation since it makes it very clear which fields belong to which class. In designing the metaphor for our visualisation, we wanted to make it obvious which fields are contained in which classes and we felt that cylinders resting on top of a class box were an intuitive way to visualise this.

A simple example visualisation can be seen in Figure 9. In this example, there are three different classes called Test1, Test2 and Test3 and several fields.

Looking more closely at the representation of fields, we can see that one field is represented by two cylinders: one growing out of the top of the class box and the other growing out of the bottom of the class box. The total height of the cylinders represents the number of accesses to the field it represents, making it easy to spot fields that are accessed frequently or not at all. The colour of the cylinders represents the access modifier of the field. **Private** fields are green, **package** fields are yellow, **protected** fields are orange and **public** fields are red. The colours were chosen to represent the level of exposure of a field, with red signifying high exposure and green signifying low

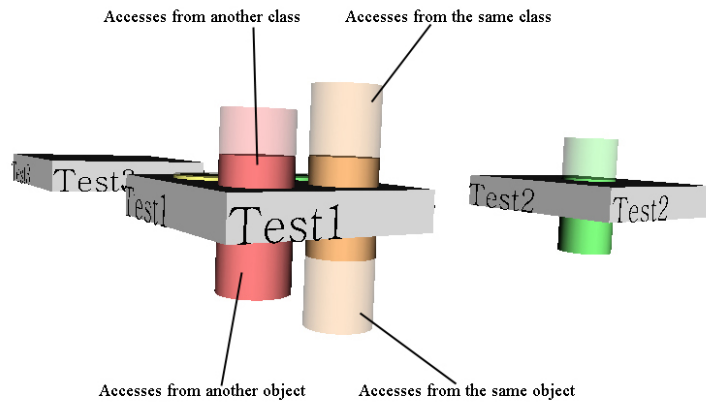


Figure 9: A simple example visualisation

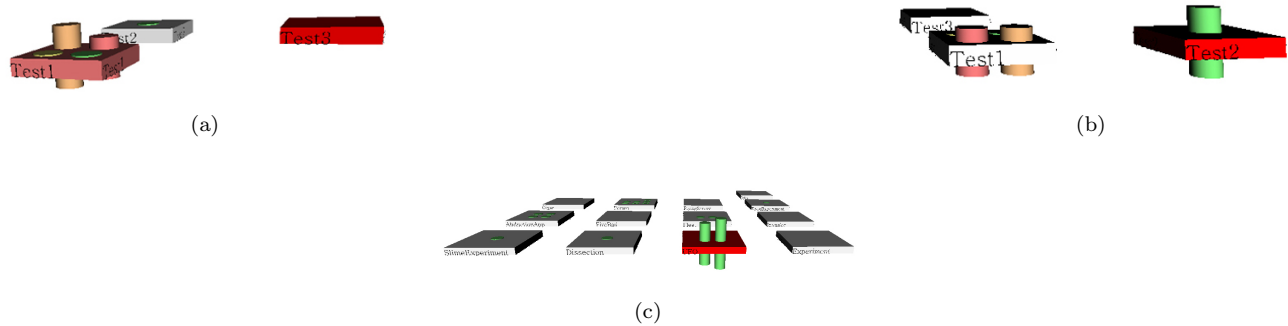


Figure 10: The visualisation after a class is selected

exposure. This part of the visualisation assumes that the software is written in Java or a similar language with four distinct access modifiers. However, this does not mean that the visualisation could not be used for other languages as long as an appropriate XML file is supplied.

As can be seen in Figure 9, the top and bottom cylinder which represent a single field both have two separate parts. The part closest to the class box is solid while the remaining part of the cylinder is slightly transparent. The solid part of the top cylinder represents the number of accesses to the field that come from outside the class the field is declared in, while the transparent part represents the number of accesses to the field from within the class. For the bottom cylinder on the other hand, the solid part represents the number of accesses from outside the object that contains the field while the transparent part represents the number of accesses from within the same object. In Section 7.2, we explain how this can be used to distinguish between object and class encapsulation.

In addition to the basic display of information about fields and field accesses, our visualisation includes some interactivity which can be used to find more detailed information about a particular part of the system. To get more information about the fields accessed by a particular class, a user can click on one of the class boxes. The effects of this interaction can be seen in Figure 10.

The class which was clicked is highlighted in red to make it easy to see which class the current visualisation applies to. For each of the fields in the entire system, the height of the cylinders representing the field is adjusted according to the number of accesses to that field from the selected class. This means that a lot of field cylinders will disappear if the field they represent is never accessed from the class that is selected. By adjusting the height of the field cylinders, we can easily see which fields are accessed from a class and where in the system those fields are. For example, in Figure 10c, it is clear that the selected class accesses only its own fields and does not access fields in other classes. This indicates good encapsulation since the class does not depend on the internal details of other classes. In this way, our visualisation can be used to see good encapsulation features in a system as well as problems.

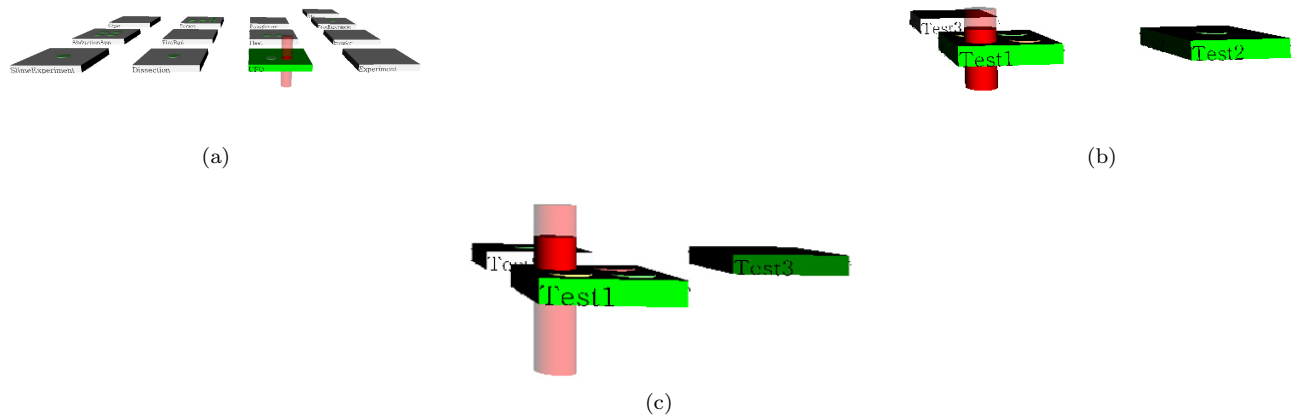


Figure 11: The visualisation after a field is selected

In addition to adjusting the height of field cylinders, the subclasses and superclasses of the selected class will also be highlighted if they contain fields that are accessed by the selected class. The superclasses will be highlighted in a lighter red while the subclasses will be dark red. This makes it easy to spot a class which accesses fields in its superclasses or subclasses. The shade of the colour will become progressively lighter and darker the further away in the class hierarchy a class is from the selected class.

An access to a field in a superclass may not constitute a severe breach in encapsulation. In fact, supporters of object encapsulation would argue that accessing fields in a superclass is totally natural. A class accessing a field in its subclasses on the other hand may constitute a problem since it is widely accepted that classes should have little or no knowledge of their subclasses. A class accessing fields that are completely outside its class hierarchy is also likely to be a problem and shows a breach of encapsulation.

In Figure 10a, we can clearly see that the selected class, Test3 only accesses fields in its superclass Test1. In Figure 10b on the other hand, Test2 accesses fields in Test1 which is not its superclass since it is not highlighted. This constitutes a potential breach of encapsulation and should be investigated.

In addition to getting more detailed information about the fields accessed by a particular class, users can also find out from which classes a particular field is accessed by clicking on a field cylinder. The effects of this interaction can be seen in Figure 11.

When a field is selected, it is highlighted in red to make it easy for the user to see which field was selected. The height of the field bar is restored to the default height, showing the total number of accesses to the field. The solid and transparent parts of the cylinder again show the number of accesses from within and outside the object or class.

Any classes that access the selected field are highlighted in green to make them easy to find. For example, in Figure 11a, we can clearly see that the selected field is only accessed by the class it is declared in. This means that there is no breach of encapsulation in this case, since no other classes depend on the selected field. In the same way, it would be easy to see any fields that are never accessed by the class they are declared in but are accessed by other classes. Such fields may be candidates for moving to a different class.

As with the class selection visualisation above, the superclasses and subclasses of the class a field is declared in are highlighted in light and dark green respectively. The shading for the classes is used to show the position of a class in the class hierarchy relative to the class containing the selected field. This makes it easy to see if a field is accessed from a superclass or a subclass. Again, an access coming from a subclass may not be considered a problem, particularly by proponents of object encapsulation. However, accesses coming from a superclass or from a class outside the class hierarchy in may be considered breaches of encapsulation.

For example, in Figure 11b, the selected field in class Test1 is accessed both by the class it is declared in and by another class, Test2. Because Test2 is not part of class hierarchy that the field belongs to, it is also highlighted in bright green. In Figure 11c on the other hand, the selected field in class Test1 is accessed from within the class it is declared in and also from the subclass Test3. It is easy to see that Test3 is a subclass of Test1 because it is highlighted in dark green.

It is easy to go back to the global visualisation of the system by clicking the *Global View* button on the bottom



Figure 12: A class with a lot of accesses to fields

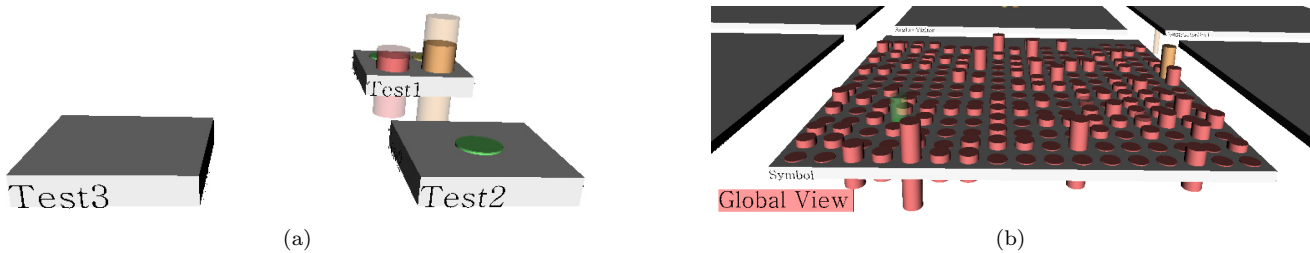


Figure 13: Potential encapsulation problems seen in the visualisation

left of the screen. This restores all the field cylinders and class boxes to their original size and colour.

In a very large system, it can be difficult to compare two classes that are far away from each other. Therefore, classes in the visualisation can be moved around on the XZ-plane by clicking on the label of the class and dragging the class box to the desired location. In this way, classes can be placed side by side and compared easily.

7.2 Informal Evaluation

We have found that our visualisation can be very useful for finding certain types of encapsulation problems. For example, it is very easy to spot any variables that are accessed frequently. This can be seen in Figure 12. The variables of this class are clearly accessed heavily, particularly from outside of the class. On the other hand, they are rarely accessed from outside the object. From this we can infer that they are commonly accessed from subclasses. This may be a concern for anyone practising class encapsulation but is acceptable if object encapsulation is used.

It is also easy to find any fields that are not accessed at all. This becomes evident when looking at Figure 13a. It is easy to see that the field in Test2 is never accessed. While this is not really an encapsulation problem as such, fields that are never accessed can be difficult to spot and should be removed to simplify the code.

Ken Auer advises software developers to create accessor and mutator methods for each field and always access fields through these methods, even within the class the field is declared in [5]. He argues that this will increase the flexibility and maintainability of software, since subclasses are free to override the accessors and mutators. In such a system, the number of accesses to each field would be very low. Therefore, such practices would be easy to spot using our visualisation.

In addition to helping find fields that are accessed frequently or not at all, our visualisation is also very useful for finding non-private fields. Such fields are an encapsulation hazard since they can be accessed from outside the class or object in which they are declared. In Figure 13b we can easily see that one class has a large number of fields, many of which are `public`.

As can be seen from the same example, it is also easy to find classes with a large number of fields. Too many fields can make a class difficult to manage and therefore there is a lot of advice telling developers to split such classes, including the *Large class* code smell [36].

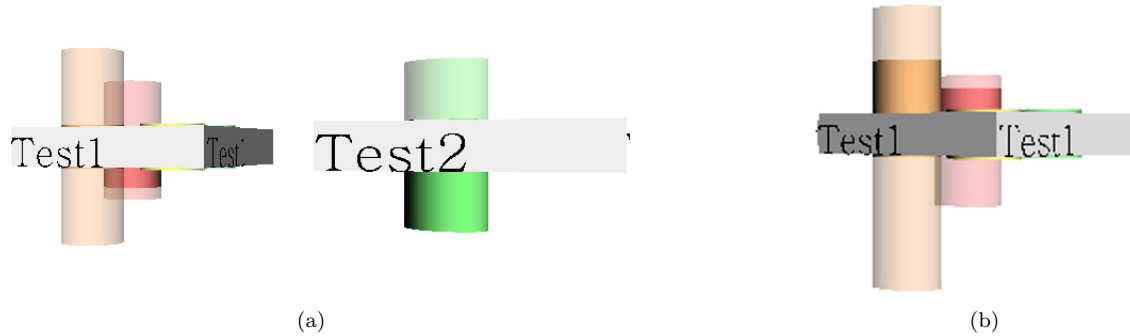


Figure 14: The difference between object and class encapsulation in the visualisation

Overall, our visualisation makes it easy to look at the protection level of fields and to determine where a field is accessed from, particularly when using the interactions described in Section 7.1. This can help us to find fields that may have been placed in the wrong class. If we find a field that is never or very rarely accessed by the class it is declared in, it may be worth considering moving it into another class which accesses it more frequently.

We are primarily interested in the differences between object and class encapsulation. Using our visualisation, it is easy to see the difference between object and class encapsulation and to determine which is used more in a particular program. Figure 14a shows the visualisation of a system that uses pure class encapsulation and Figure 14b shows a system that uses pure object encapsulation. For class encapsulation, we can clearly see that there are field accesses from outside the object but no accesses coming from outside the class. For object encapsulation, we can see that there are no accesses from outside the object but some accesses from outside the class.

While our visualisation can be used to find encapsulation breaches in software and to determine what kind of encapsulation is used in a system, it does have some problems in its current form. Because of the interactions it supports, the size of the VRML worlds containing the visualisations is very large, even for medium-sized systems. Even an older version of JST which contains only 45 classes produces a 143MB VRML world for our visualisation. JST is relatively small compared to industrial-scale software so visualisations of such systems would likely be infeasible because of the size of the VRML world. Even the visualisation of JST is quite slow to load in the browser and to navigate around. This indicates that we either need to reduce the size of our VRML visualisations or choose a different technology to implement our visualisation. The interactions that we have implemented lead to the increase in size of our VRML worlds because VRML is not very well suited for a large amount of event handling and dynamic content. We expect that implementing our visualisation using a different technology such as Java3D or OpenGL would likely solve this problem.

Currently, the classes in the system are randomly placed in the visualisation in no particular order, which can make it hard to find particular classes. However, in future visualisations, the spatial position of a class could be used to convey further information to visualisation users. For example, classes that are in the same package could be placed close to each other so that it would be easy to find a particular class in the visualisation. This would also make it easy to spot classes in different packages accessing each other's data.

Classes that communicate a lot and are tightly coupled could also be placed close to each other in the visualisation. Such classes are more likely to access each other's data. This would help highlight unusual accesses that occur between classes which are not tightly coupled and are thus further apart in the visualisation.

Rather than positioning classes on a simple 2D plane, we could consider extending our visualisation to position classes in a 3D space. This could for example be used to reinforce inheritance relationships in the system, where classes from different levels of the class hierarchy could be placed on different planes in the visualisation. In this way, a lot of additional information that may be relevant to encapsulation could be added to the visualisation.

Finally, given the size of visualisations of industrial-scale software it can be hard to compare visualisations of subsequent versions of a system to see what has changed. There is no support for such a comparison built into the current visualisation. However, in the future we could work on creating a visualisation that, rather than showing the features of a particular version of the system, compares two versions of the same system, highlighting the changes between them. This could be very useful to gauge the effectiveness of refactorings or redesigns and their effect on the encapsulation in a system.

8 Discussion

As part of this research, we surveyed developers and students about their encapsulation preferences, and we conducted an empirical investigation of encapsulation practices in student programs and real-world programs. Our results clearly show that encapsulation practices, particularly in industry, are inconsistent, and many programs are weakly encapsulated. We have also uncovered confusion among both students and professional software developers about encapsulation and the encapsulation mechanisms provided by programming languages.

These results have far-reaching implications because encapsulation is such a fundamental tool for managing complexity. Not using it correctly can lead to tight coupling and significantly decrease the maintainability and understandability of software.

The large variation of encapsulation practices we have seen employed in real software is, we suggest, due to a lack of awareness of the existence of different encapsulation boundaries. This problem is compounded by conflicting advice on how to use encapsulation, which has arisen due to the different schools of thought. However, what is surprising is the great number of variations within single programs, indicating the confusion and inconsistency within single software development teams.

The evidence we have gathered indicates that software developers need to be more aware of exactly how they practise encapsulation, so that they understand the consequences of the specific encapsulation policy they have chosen. This includes being aware of the different types of encapsulation, including class encapsulation, object encapsulation and intersection encapsulation. In addition, development teams should know about the different encapsulation advice and make a conscious choice about which advice they agree with and choose to follow.

Advice about how to practice encapsulation is influenced by the way the author thinks about encapsulation and which encapsulation boundary they assume. For example, Riel's heuristics assume that class encapsulation is used. This means that in some cases these heuristics may not apply when object encapsulation is used and they need to be re-interpreted to work with a different encapsulation boundary. Therefore, it is essential for developers to be aware not only of the encapsulation boundary they use in their program, but also which encapsulation boundary is assumed by design heuristics and principles.

We have proposed the development of encapsulation tools allowing developers to choose a particular encapsulation policy which will then be consistently implemented in the system. This allows programs to be encapsulated at specific points during the development process. However, it is still a good idea for developers to practise encapsulation at all stages of the software development process, particularly in the absence of such tools. Only by practising encapsulation consistently and constantly can developers reap all the benefits it provides.

In addition to refactoring tools we have also designed and implemented a visualisation to help developers spot encapsulation breaches during the development process. We hope that such a visualisation will make it easier for developers to understand how encapsulation is being used in a system, apply encapsulation consistently and find and address any encapsulation problems.

We hope that in the future, better languages will be designed to support different encapsulation policies and to be more consistent with developers' expectations. In particular, we would like to see statically typed languages supporting object encapsulation, which we have shown to be more intuitive than class encapsulation. We believe that lining up a programming language's encapsulation mechanisms with developers' expectations will greatly improve encapsulation practices and lessen the confusion currently surrounding encapsulation. In addition, we believe that this would limit the use of intersection encapsulation, the most restrictive kind of encapsulation, which is commonly used in current software. Furthermore, we would hope that future languages would enforce encapsulation more strictly, for example by providing only a single protection level or at least by making `private` or `protected` the default protection level.

In the meantime, however, developers can still apply different encapsulation policies using mainstream languages such as Java by following some simple guidelines.

To practise *class encapsulation* in Java, developers should:

- Make all fields `private`; and
- Access fields directly within the class they are declared in, even if those accesses come from other objects.

To practise *object encapsulation* in Java, developers should:

- Make all fields `protected`;
- Never access fields from outside the class hierarchy in which they are declared; and

- Avoid accesses of the form *qualifier.member* where *qualifier* is anything other than *this* or *super* because these accesses may come from other objects.

To practise *intersection encapsulation* in Java, developers should:

- Make all fields `private`; and
- Avoid accesses of the form *qualifier.member* where *qualifier* is anything other than *this* or *super* because these accesses may come from other objects.

In addition to following guidelines like these, it would also be possible to create different versions of Java which support different encapsulation policies simply by modifying the Java grammar. This would be a simple change that would be much faster than designing and implementing completely new languages. A change to the Java grammar could enforce the different encapsulation policies, forcing developers to be aware of different encapsulation boundaries, choose one for their program and adhere to it.

For the *class encapsulation* version of Java we would suggest the following changes to the grammar:

- Remove access modifiers for fields; and
- Make `private` the default protection level.

For the *object encapsulation* version of Java we would suggest the following changes to the grammar:

- Remove access modifiers for fields;
- Make `protected` the default protection level; and
- Disallow accesses to fields of the form *qualifier.member* where *qualifier* is not *this* or *super*.

For the *intersection encapsulation* version of Java we would suggest the following changes to the grammar:

- Remove access modifiers for fields;
- Make `private` the default protection level.
- Disallow accesses to fields of the form *qualifier.member* where *qualifier* is not *this* or *super*.

The changes to the grammar described above would enforce the different kinds of encapsulation in Java. This shows that only simple changes to languages are required to support different encapsulation policies, indicating that it is potentially feasible to have different versions of the language for different encapsulation policies. We are particularly interested in creating an object encapsulation version of Java because we believe that object encapsulation is the most intuitive encapsulation type.

9 Conclusions and Future Work

Encapsulation is one of the most fundamental tools developers have for managing complexity in software, making it central to developing maintainable and understandable code. Despite being so basic, there is a lot of confusion among developers about how to practise encapsulation. This is in part caused by the wide range of sometimes conflicting advice about encapsulation and is compounded by the fact that encapsulation mechanisms in most mainstream programming languages do not meet programmers' needs and expectations.

We have collected, compared and contrasted a wide range of advice about practising encapsulation, showing the main conflicts and identifying three different encapsulation policies: *object encapsulation*, *class encapsulation* and *intersection encapsulation*. The existence of these different policies appears to have mostly gone unnoticed so far, despite the fact that it has important implications for software development. We believe it is vital that developers are aware of these differences, not only so that they can choose the policy that will provide them with the most advantages, but also because a lot of design advice relies on assumptions about encapsulation policies, meaning that one set of design advice may not apply to all encapsulation policies.

Through a survey of students and developers we have shown that object encapsulation is more intuitive for novice programmers and that even experienced programmers appear confused about encapsulation. Following the survey we have analysed the encapsulation practices in 33 real-world programs and 11 student programs and found that encapsulation practices are very varied, confused and inconsistent, especially in industry programs, leading to generally weak encapsulation. This is a significant finding because the chaos in current encapsulation practices is likely to have far-reaching implications for software quality.

We have proposed refactoring tools and a visualisation to help developers make the encapsulation in a program stronger and more consistent. We hope that programming languages with encapsulation mechanisms that better meet developers' needs and enforce encapsulation policies will be developed in the future; in the meantime we have proposed some simple guidelines and changes to the Java grammar which can enforce specific encapsulation policies and practices in the short run.

The specific contributions of this work include:

- The identification of three different encapsulation policies: object encapsulation, class encapsulation and intersection encapsulation;
- The identification of conflicts in encapsulation advice;
- A survey showing that object encapsulation is more intuitive for novice programmers, and that more experienced programmers have adapted to some degree to the encapsulation mechanisms provided by programming languages, although they may still be confused about encapsulation in general;
- Empirical evidence that encapsulation in real-world software is inconsistent and weak;
- Empirical evidence that, when an encapsulation boundary preference is shown in real world software, there is a tendency towards object encapsulation, rather than class encapsulation;
- A refactoring tool to improve encapsulation in software;
- A visualisation to help developers find encapsulation problems in software;
- Guidelines for developers using Java for different encapsulation policies;
- Suggestions of specific changes to the Java grammar to support different encapsulation policies;
- Testing and improvements to the Java semantic model JST; and
- The submission of two conference papers about this work, one of which has been accepted, with the other pending.

In this work, we have focused on data encapsulation; that is the hiding of data. However, there are other things that can be encapsulated, including methods, which are also given protection levels. In the near future, we hope to extend our work to include method protection levels to further deepen our understanding of encapsulation practices.

We have already implemented a simple refactoring tool using JST parse trees. In future work we hope to create a more sophisticated tool which can change a program to be consistent with a particular encapsulation policy such as object or class encapsulation.

We have also proposed changes to the Java grammar to allow the language to enforce certain encapsulation policies. We are particularly interested in implementing an object encapsulation version of Java by modifying the grammar and providing a new parser. This new version of Java could then be given to students and professional developers to see how it changes their encapsulation practices and whether or not they prefer it to the usual Java version. This could give us further insight into how much the programming language and the encapsulation mechanisms it provides affect the encapsulation practices employed by developers.

Our current investigation into how developers think about encapsulation and how developers use encapsulation in practice has focused on developers using Java or C#, which both use class encapsulation. The encapsulation mechanisms of the language used by a developer almost certainly impact a developer's encapsulation habits and practices. Therefore, it would be interesting to redo these experiments with developers using an object encapsulation language such as Ruby or Smalltalk. However, object encapsulation languages are less widely used and so finding enough developers to take part in such an experiment would be problematic. They are also usually dynamically typed, meaning there is much less information available to our static analysis tools. We could potentially use the object encapsulation version of Java we plan to develop for this purpose.

10 References

- [1] Accessors are evil. <http://c2.com/cgi/wiki?AccessorsAreEvil>, 2007.
- [2] Law of Demeter. <http://c2.com/cgi/wiki?LawOfDemeterAndCoupling>, 2008.
- [3] Tell don't ask. <http://c2.com/cgi/wiki?TellDontAsk>, 2008.
- [4] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobsen, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language*. Oxford University Press, New York, 1977.
- [5] Ken Auer. Reusability through self-encapsulation. pages 505–516, 1995.
- [6] Holger Bär and Oliver Ciupke. Exploiting design heuristics for automatic problem detection. In *ECOOP '98: Workshop on Object-Oriented Technology*, pages 73–74, London, UK, 1998. Springer-Verlag.
- [7] Richard Barker and Ewan Tempero. A large-scale empirical comparison of object-oriented cohesion metrics. In *APSEC '07: Proceedings of the 14th Asia-Pacific Software Engineering Conference*, pages 414–421, Washington, DC, USA, 2007. IEEE Computer Society.
- [8] Victor Basili, Lionel Briand, and Walcélio Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.
- [9] Lionel Briand, Prem Devanbu, and Walcelio Melo. An investigation into coupling measures for C++. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 412–421, New York, NY, USA, 1997. ACM.
- [10] Lionel Briand, Sandro Morasca, and Victor Basili. Defining and validating measures for object-based high-level design. *IEEE Transactions on Software Engineering*, 25(5):722–743, 1999.
- [11] Lionel Briand, Jürgen Wüst, John Daly, and Victor Porter. Exploring the relationship between design measures and software quality in object-oriented systems. *Journal of Systems and Software*, 51(3):245–273, 2000.
- [12] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc., New York, NY, USA, 1996.
- [13] Alexander Chatzigeorgiou, Nikolaos Tsantalis, and George Stephanides. Application of graph theory to OO software engineering. In *WISER '06: Proceedings of the 2006 international workshop on Workshop on interdisciplinary software engineering research*, pages 29–36, New York, NY, USA, 2006. ACM.
- [14] Alexander Chatzigeorgiou, Spiros Xanthos, and George Stephanides. Evaluating object-oriented designs with link analysis. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 656–665, Washington, DC, USA, 2004. IEEE Computer Society.
- [15] Shyam Chidamber and Chris Kemerer. Towards a metrics suite for object oriented design. In *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 197–211, New York, NY, USA, 1991. ACM.
- [16] Shyam Chidamber and Chris Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.
- [17] Neville Churcher and Alan Creek. Building virtual worlds with the big-bang model. In *APVis '01: Proceedings of the 2001 Asia-Pacific symposium on Information visualisation*, pages 87–94, Darlinghurst, Australia, Australia, 2001. Australian Computer Society, Inc.
- [18] Neville Churcher, Sarah Frater, Cong Phuoc Huynh, and Warwick Irwin. Supporting OO design heuristics. In *ASWEC '07: Proceedings of the 2007 Australian Software Engineering Conference*, pages 101–110, Washington, DC, USA, 2007. IEEE Computer Society.
- [19] Neville Churcher and Warwick Irwin. Informing the design of pipeline-based software visualisations. In *APVis '05: proceedings of the 2005 Asia-Pacific symposium on Information visualisation*, pages 59–68, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.

- [20] Neville Churcher, Warwick Irwin, and Carl Cook. Inhomogeneous force-directed layout algorithms in the visualisation pipeline: from layouts to visualisations. In *APVis '04: Proceedings of the 2004 Australasian symposium on Information Visualisation*, pages 43–51, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.
- [21] Neville Churcher, Warwick Irwin, and Ron Kriz. Visualising class cohesion with virtual worlds. In *APVis '03: Proceedings of the Asia-Pacific symposium on Information visualisation*, pages 89–97, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.
- [22] Neville Churcher, Lachlan Keown, and Warwick Irwin. Virtual worlds for software visualisation. In Aaron Quigley, editor, *SoftVis99 Software Visualisation Workshop*, pages 9 – 16, Sydney, Australia, December 1999.
- [23] Neville Churcher and Martin Shepperd. Comments on 'a metrics suite for object oriented design'. *IEEE Transactions on Software Engineering*, 21(3):263–265, 1995.
- [24] Peter Coad. Object-oriented patterns. *Communications of the ACM*, 35(9):152–159, 1992.
- [25] Peter Coad and Edward Yourdon. *Object-oriented design*. Yourdon Press, Upper Saddle River, NJ, USA, 1991.
- [26] Alistair Cockburn. The interaction of social issues and software architecture. *Communications of the ACM*, 39(10):40–46, 1996.
- [27] Anthony Cohen. Data abstraction, data encapsulation and object-oriented programming. *SIGPLAN Notices*, 19(1):31–35, 1984.
- [28] Alexandre Correa, Cláudia Werner, and Gerson Zaverucha. Object oriented design expertise reuse: An approach based on heuristics, design patterns and anti-patterns. In *ICSR-6: Proceedings of the 6th International Conference on Software Reuse*, pages 336–352, London, UK, 2000. Springer-Verlag.
- [29] Edsger Dijkstra. On the role of scientific thought. *Selected Writings on Computing: A Personal Perspective*, pages 60 – 66, 1982.
- [30] Fernando Brito e Abreu. The MOOD metrics set. In *Proceedings of ECOOP95 Workshop Metrics*, 1995.
- [31] Fernando Brito e Abreu and Walclio Melo. Evaluating the impact of object-oriented design on software quality. In *In Proceedings of the 3rd International Software Metrics Symposium*, pages 90–99, 1996.
- [32] Bruce Eckel and Ziegler Fix Sysop. Thinking in Java. In *PTR*, Upper Saddle River, NJ. Prentice Hall, 1998.
- [33] Encyclopaedia Britannica Inc. *Encyclopaedia Britannica*. Chicago, Illinois, 1986.
- [34] Norman Fenton and Shari Pfleger. *Software Metrics: A Rigorous and Practical Approach, 2nd edition*. International Thomson Computer Press, london, 1997.
- [35] Martin Fowler. *Analysis patterns: reusable objects models*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.
- [36] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [37] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
- [38] Javier Garzás and Mario Piattini. *Object-Oriented Design Knowledge: Principles, Heuristics, and Best Practices*. IGI Publishing, Hershey, PA, USA, 2007.
- [39] Maurice Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., New York, NY, USA, 1977.
- [40] Brian Henderson-Sellers. *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.
- [41] Sallie Henry and Dennis Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, 7(5):510–518, 1981.

- [42] Sallie Henry and Dennis Kafura. The evaluation of software systems structure using quantitative software metrics. pages 99–111, 1993.
- [43] Allen Holub. Why getter and setter methods are evil. *JavaWorld*, 2003.
- [44] Interational Organization for Standardization. ISO/IEC 9126 - information technology - software product evaluation - quality characteristics and guidelines for their use. *ISO JTCUSC7*, 1991.
- [45] Warwick Irwin. *Understanding and Improving Object-Oriented Software Through Static Software Analysis*. PhD thesis, University of Canterbury, 2007.
- [46] Warwick Irwin and Neville Churcher. XML in the visualisation pipeline. In *VIP '01: Proceedings of the Pan-Sydney area workshop on Visual information processing*, pages 59–67, Darlinghurst, Australia, Australia, 2001. Australian Computer Society, Inc.
- [47] Warwick Irwin and Neville Churcher. Object oriented metrics: Precision tools and configurable visualisations. In *in METRICS2003: 9th IEEE Symposium on Software Metrics, IEEE*, pages 112–123. Press, 2003.
- [48] Warwick Irwin, Carl Cook, and Neville Churcher. Parsing and semantic modelling for software engineering applications. In *ASWEC '05: Proceedings of the 2005 Australian conference on Software Engineering*, pages 180–189, Washington, DC, USA, 2005. IEEE Computer Society.
- [49] Keith Johnson and Bill Smith. Javascrc. <http://javascrc.sourceforge.net>, 2002.
- [50] Ralph Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22 – 35, 1988.
- [51] Stephen Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [52] Alan C. Kay. The early history of smalltalk. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pages 69–95, New York, NY, USA, 1993. ACM.
- [53] Sascha Konrad, Betty Cheng, and Laura Campbell. Object analysis patterns for embedded systems. *IEEE Transactions on Software Engineering*, 30(12):970–992, 2004.
- [54] John Lakos. *Large-scale C++ software design*. Addison-Wesley, 1996.
- [55] Wei Li and Sallie Henry. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2):111–122, 1993.
- [56] Karl Lieberherr and Ian Holland. Assuring good style for object-oriented programs. *IEEE Software*, pages 38 – 48, 1989.
- [57] Barbara Liskov. Data abstraction and hierarchy. In *ACM SIGPLAN Notices*, pages 17 – 34, May 1987.
- [58] Mark Lorenz and Jeff Kidd. *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.
- [59] Robert Martin. Oh no! DTO! *Artima Developer*, February 1994.
- [60] Robert Martin. Granularity. *C++ Report*, 8(10):57 – 62, 1996.
- [61] Robert Martin. The Liskov Substitution Principle. *C++ Report*, 8(3):16 – 17, 20 – 23, 1996.
- [62] Tobias Mayer and Tracy Hall. A critical analysis of current OO design metrics. *Software Quality Control*, 8(2):97–110, 1999.
- [63] Thomas McCabe. A complexity measure. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, page 407, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.
- [64] Thomas McCabe and Charles Butler. Design complexity measurement and testing. *Communications of the ACM*, 32(12):1415–1425, 1989.
- [65] Jim McCall, Paul Richards, and Gene Walters. *Factors in Software Quality*. NTIS, 1977.

- [66] Hayden Melton and Ewan Tempero. Identifying refactoring opportunities by identifying dependency cycles. In *ACSC '06: Proceedings of the 29th Australasian Computer Science Conference*, pages 35–41, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.
- [67] Hayden Melton and Ewan Tempero. An empirical study of cycles among classes in Java. *Empirical Software Engineering*, 12(4):389–415, 2007.
- [68] Hayden Melton and Ewan Tempero. Static members and cycles in Java software. In *ESEM '07: Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, pages 136–145, Washington, DC, USA, 2007. IEEE Computer Society.
- [69] Hayden Melton and Ewan Tempero. The CRSS metric for package design quality. In *ACSC '07: Proceedings of the thirtieth Australasian conference on Computer science*, pages 201–210, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.
- [70] Bertrand Meyer. *Object-oriented software construction*. Prentice Hall, 2nd edition edition, 1988.
- [71] Bertrand Meyer. *Object-oriented software construction*. Prentice-Hall international series in computer science. Prentice Hall, New York, 1988.
- [72] Naouel Moha. Detection and correction of design defects in object-oriented designs. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 949–950, New York, NY, USA, 2007. ACM.
- [73] Blair Neate, Warwick Irwin, and Neville Churcher. Coderank: A new family of software metrics. In *ASWEC '06: Proceedings of the Australian Software Engineering Conference*, pages 369–378, Washington, DC, USA, 2006. IEEE Computer Society.
- [74] David Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.
- [75] Chris Parnin and Carsten Görg. Lightweight visualizations for inspecting code smells. In *SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization*, pages 171–172, New York, NY, USA, 2006. ACM.
- [76] Chris Parnin, Carsten Görg, and Ogechi Nnadi. A catalogue of lightweight visualizations to support code smell inspection. In *SoftVis '08: Proceedings of the 4th ACM symposium on Software visualization*, pages 77–86, New York, NY, USA, 2008. ACM.
- [77] Roger Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 2001.
- [78] Qualitas Research Group. Criteria for inclusion in qualitas corpus. <http://www.cs.auckland.ac.nz/~ewan/corpus/criteria.html>, February 2009.
- [79] Qualitas Research Group. Qualitas corpus metadata. <http://www.cs.auckland.ac.nz/~ewan/corpus/metadata.html>, February 2009.
- [80] Qualitas Research Group. Qualitas corpus version 20090202. <http://www.cs.auckland.ac.nz/~ewan/corpus>, February 2009.
- [81] Arthur Riel. *Object-Oriented Design Heuristics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.
- [82] Paul Rogers. Encapsulation is not information hiding. *Java World*, May 2001.
- [83] Peter Sanders and David Steurer. Software complexity. *Crosstalk, Journal of Defense Software Engineering*, 7:5–9, 1994.
- [84] Douglas Schmidt. Using design patterns to develop reusable object-oriented communication software. *Communications of the ACM*, 38(10):65–74, 1995.
- [85] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *OOPSLA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 38–45, New York, NY, USA, 1986. ACM.

- [86] Ioannis Stamelos, Lefteris Angelis, Apostolos Oikonomou, and Georgios Bleris. Code quality analysis in open source software development. *Information Systems Journal*, 12(1):43 – 60, 2002.
- [87] Bjarne Stroustrup. *The C++ programming language, third edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.
- [88] Ramanath Subramanyam and Mayuram Krishnan. Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects. *IEEE Transactions on Software Engineering*, 29(4):297–310, 2003.
- [89] Ewan Tempero. How fields are used in java: An empirical study. In *ASWEC '09: Proceedings of the 2009 Australian Software Engineering Conference*, pages 91–100, Washington, DC, USA, 2009. IEEE Computer Society.
- [90] Ewan Tempero, James Noble, and Hayden Melton. How do Java programs Use inheritance? An empirical study of inheritance in Java software. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 667–691, Berlin, Heidelberg, 2008. Springer-Verlag.
- [91] John Verity and Evan Schwartz. Software made simple; will object oriented programming transform the computer industry? *Business Week*, pages 92–97, 1991.
- [92] Janina Voigt. Coderank: Extensions and experiments. Technical Report TR-COSC 01/08, Department of Computer Science and Software Engineering, University of Canterbury, 2008.
- [93] Kurt Welker and Paul Oman. Software maintainability metrics models in practice. *Crosstalk, Journal of Defense Software Engineering*, 8:19–23, 1995.
- [94] Peter Wendorff. Assessment of design patterns during software reengineering: Lessons learned from a large commercial project. In *CSMR '01: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, page 77, Washington, DC, USA, 2001. IEEE Computer Society.
- [95] Hong Yul Yang, Ewan Tempero, and Hayden Melton. An empirical study into use of dependency injection in Java. In *ASWEC '08: Proceedings of the 19th Australian Conference on Software Engineering*, pages 239–247, Washington, DC, USA, 2008. IEEE Computer Society.
- [96] Edward Yourdon and Larry Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1979.

A Large Diagrams

A larger version of Figure 16 can be found at <http://wiki3.cosc.canterbury.ac.nz/index.php/Image:JaninasNewDesign21.png>.

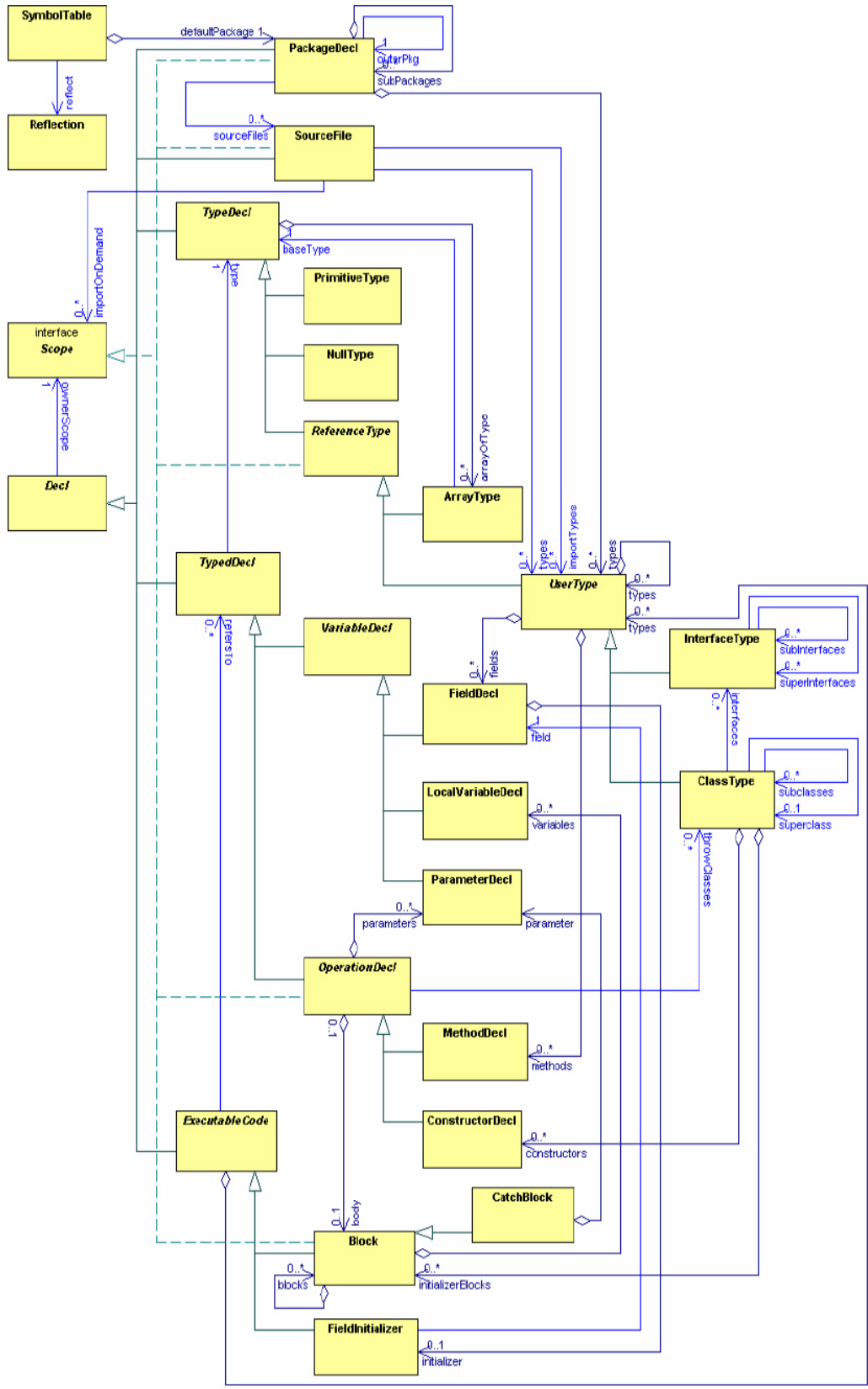


Figure 15: A UML class diagram of JST

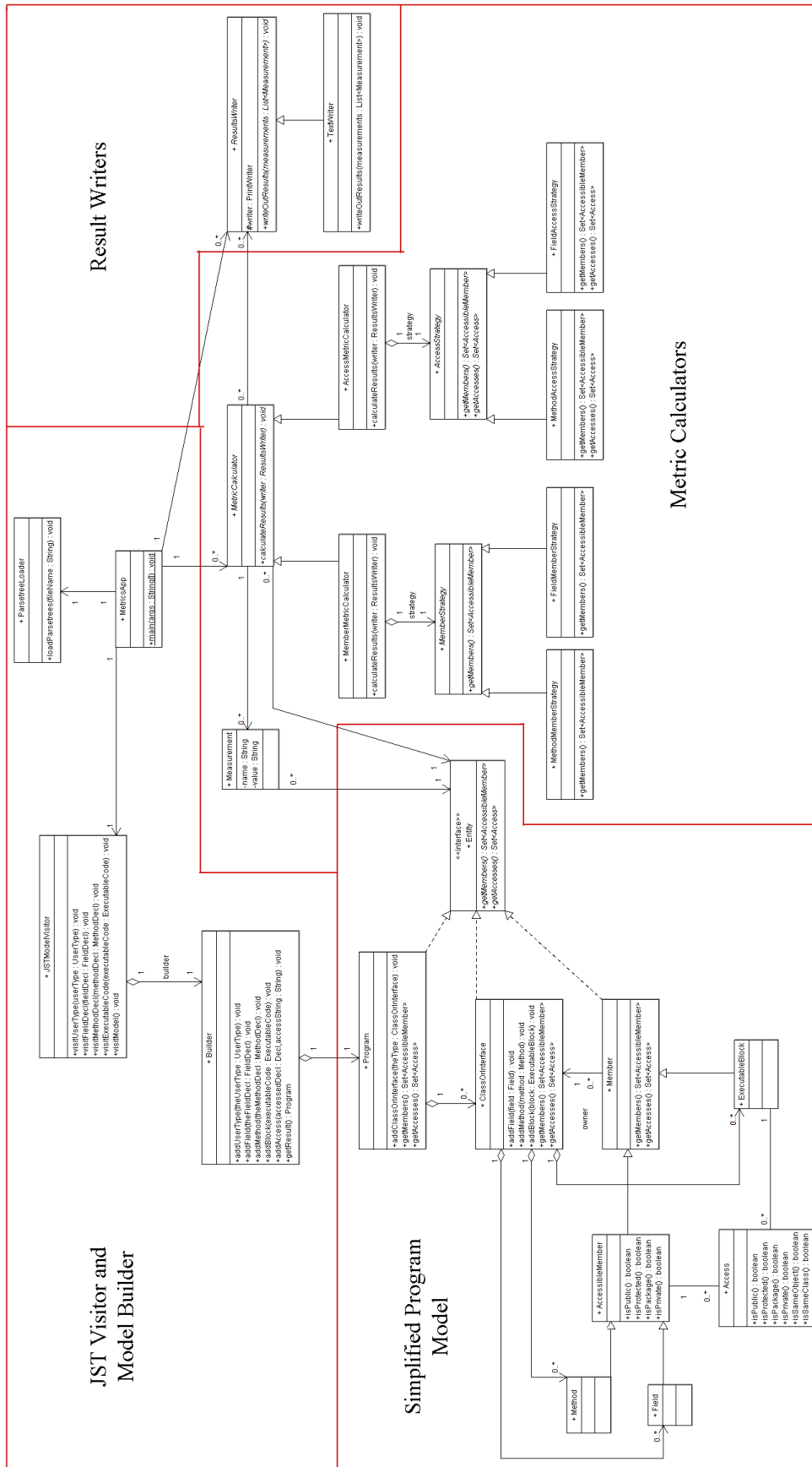


Figure 16: A UML class diagram of the analysis program

B Detailed data from encapsulation analysis of real-world and student programs

Table 6 shows detailed encapsulation data from corpus programs and Table 7 shows the same data for student programs. More detailed data, including data at program and class level was collected and analysed during the study but the aggregate data allows the most useful conclusions to be drawn and is therefore presented here. The correctness of this data was checked through careful testing of the encapsulation analysis program and by analysing some programs by hand and comparing the results. Note that the seemingly impossible accesses to private fields from a different class and a different object are caused by inner class objects accessing private fields in the outer object.

	Average	Minimum	Maximum	Standard Deviation
Total Fields	657.5882353	69	2159	506.9838392
Public Fields (%)	8.197895703	0	63.63636364	14.54170087
Protected Fields(%)	14.03413801	0	61.86440678	17.33452946
Package Fields (%)	17.47389307	0	69.66292135	17.33671453
Private Fields (%)	60.29407321	1.913875598	99.58333333	27.04349408
Total Accesses	3619.5	355	10818	2773.951107
Accesses per Field	5.811465856	3.3199446	17.867037	2.80383677
Public Accesses (%)	8.762200255	0	66.98679472	13.93046774
Protected Accesses (%)	18.04050757	0	85.24957936	20.02536543
Package Accesses (%)	14.64379262	0	67.86060019	14.59932799
Private Accesses (%)	58.55349956	0.600240096	100	27.62142468
Accesses from Outside Class (%)	14.68238602	0.3	39.85594238	10.50119523
Accesses from Outside Object (%)	10.79069496	1.971830986	40.21608643	9.543301389
Same object, same class accesses (%)	82.57180811	54.62703309	97.9909022	11.1100305
Different object, same class accesses (%)	2.745805868	0	24.52010507	4.47570256
Same object, subclass accesses (%)	6.637496932	0	31.09927089	7.692440057
Different object, subclass accesses (%)	0.20088857	0	1.800720288	0.383342642
Different object, different class accesses (%)	7.844000517	0.3	34.45378151	8.309757307
Public: same object, same class accesses (%)	3.412343394	0	29.65186074	5.792155646
Public: different object, same class (%)	0.197890683	0	3.00120048	0.574275215
Public: same object, subclass (%)	0.340648262	0	2.881152461	0.673609731
Public: different object, subclass (%)	0.086570335	0	1.800720288	0.316217561
Public: different object, different class (%)	4.72474758	0	29.65186074	7.782733305
Protected: same object, same class (%)	11.88831873	0	48.06763285	14.08346826
Protected: different object, same class (%)	0.378231752	0	7.599551318	1.302520678
Protected: same object, subclass (%)	5.115626232	0	31.09927089	6.989393512
Protected: different object, subclass (%)	0.063906302	0	0.536143465	0.121008951
Protected: different object, different class (%)	0.594424558	0	3.729669097	0.856950903
Package: same object, same class (%)	11.76731014	0	65.68247822	13.75361608
Package: different object, same class (%)	0.154109244	0	1.129943503	0.246958563
Package: same object, subclass (%)	1.181222438	0	22.96124031	4.249470393
Package: different object, subclass (%)	0.050411932	0	0.959532749	0.206230688
Package: different object, different class (%)	1.490738866	0	8.844388819	1.8813369
Private: same object, same class (%)	55.50383585	0.600240096	93.8	25.93713824
Private: different object, same class (%)	2.01557419	0	23.26732673	4.154713816
Private: same object, subclass (%)	0	0	0	0
Private: different object, subclass (%)	0	0	0	0
Private: different object, different class(%)	1.034089513	0	9.057713764	1.735243949
Minimum Accesses per Field	0	0	0	0
Maximum Accesses per Field	160.9117647	16	1265	236.9777342
Accesses to public from outside Class (%)	47.39112382	0	100	32.1540355
Accesses to public from outside Object (%)	44.4378246	0	97.77777778	31.00825426
Accesses to package from outside Class (%)	18.87645232	0	96.41119221	23.23446831
Accesses to package from outside Object (%)	14.74676691	0	57.17761557	16.14211148
Accesses to protected from Descendant (%)	26.27312714	0	74.41314554	23.07839286
Accesses to protected from outside Descendant (%)	5.288957804	0	77.77777778	13.30369305
Accesses to protected from outside Object (%)	7.856493349	0	77.77777778	14.03512035
Accesses to private from outside Object (%)	3.000757064	0	24.52609159	4.807652475

Table 6: Detailed encapsulation data from corpus programs

	Average	Minimum	Maximum	Standard Deviation
Total Fields	258.2727273	55	469	143.7463675
Public Fields (%)	4.421279512	0	28.33333333	8.075607552
Protected Fields(%)	2.401976308	0	13.43283582	3.821998813
Package Fields (%)	10.41026521	3.947368421	41.81818182	11.83483905
Private Fields (%)	82.76647897	58.18181818	92.85714286	12.33447433
Total Accesses	1125.363636	208	1973	648.945032
Accesses per Field	4.320690536	3.3	5.5164475	0.767121185
Public Accesses (%)	4.644821349	0	20.45454545	5.769406586
Protected Accesses (%)	2.960243149	0	16.91223575	4.917019284
Package Accesses (%)	9.613247897	0.192184497	39.90384615	11.81514243
Private Accesses (%)	82.7816876	60.09615385	94.45438283	11.04836448
Accesses from Outside Class (%)	4.797751407	1.308900524	13.13131313	3.216671771
Accesses from Outside Object (%)	6.030385652	2.356020942	15.4040404	3.85644733
Same object, same class accesses (%)	93.71653922	84.09090909	97.64397906	3.911795562
Different object, same class accesses (%)	1.485709372	0	4.383788255	1.408930136
Same object, subclass accesses (%)	0.253075127	0	1.366459627	0.474351313
Different object, subclass accesses (%)	0	0	0	0
Different object, different class accesses (%)	4.54467628	1.308900524	12.62626263	3.170021725
Public: same object, same class accesses (%)	2.747058871	0	14.39393939	4.298803229
Public: different object, same class (%)	0.342308126	0	2.777777778	0.831574064
Public: same object, subclass (%)	0	0	0	0
Public: different object, subclass (%)	0	0	0	0
Public: different object, different class (%)	1.555454353	0	4.135649297	1.591789624
Protected: same object, same class (%)	2.429210324	0	13.90134529	4.059632929
Protected: different object, same class (%)	0.035115342	0	0.284900285	0.088249521
Protected: same object, subclass (%)	0.207161445	0	1.366459627	0.471959573
Protected: different object, subclass (%)	0	0	0	0
Protected: different object, different class (%)	0.288756039	0	3.010890455	0.904192819
Package: same object, same class (%)	8.776544288	0.192184497	37.5	10.71418523
Package: different object, same class (%)	0	0	0	0
Package: same object, subclass (%)	0.045913682	0	0.505050505	0.152278457
Package: different object, subclass (%)	0	0	0	0
Package: different object, different class (%)	0.790789927	0	5.15970516	1.612482699
Private: same object, same class (%)	79.76372574	57.57575758	92.89529915	11.31663371
Private: different object, same class (%)	1.108285905	0	4.383788255	1.42155285
Private: same object, subclass (%)	0	0	0	0
Private: different object, subclass (%)	0	0	0	0
Private: different object, different class (%)	1.909675961	0	8.838383838	2.663793711
Minimum Accesses per Field	0	0	0	0
Maximum Accesses per Field	31.63636364	13	90	21.15784832
Accesses to public from outside Class (%)	36.71901881	0	87.71929825	27.40014164
Accesses to public from outside Object (%)	41.69717337	0	87.71929825	28.92524449
Accesses to package from outside Class (%)	5.297111346	0	23.33333333	8.095840239
Accesses to package from outside Object (%)	4.947460996	0	23.33333333	8.065046775
Accesses to protected from Descendant (%)	9.435483871	0	40	17.51276372
Accesses to protected from outside Descendant (%)	3.225378788	0	17.8030303	6.521649018
Accesses to protected from outside Object (%)	3.269250951	0	17.8030303	5.619484105
Accesses to private from outside Object (%)	1.264889298	0	5.110896818	1.605525835

Table 7: Detailed encapsulation data from student programs

C Paper: Intuitiveness of Class and Object Encapsulation

This is the paper we wrote about the results of the encapsulation survey and that has been accepted to ICITA '09 (International Conference on Information Technology and Applications).

Intuitiveness of Class and Object Encapsulation

Janina Voigt, Warwick Irwin, Neville Churcher

Abstract--Encapsulation is one of the most fundamental programming language mechanisms available to software developers for managing the complexity of software systems. One might therefore expect clear guidelines and consistent practices to be used in mature programming languages, and particularly in object oriented (OO) languages, with their rich support for encapsulation. However, the encapsulation practices employed by OO developers are surprisingly variable, even within a given OO language. Published advice on how best to use encapsulation is conflicting and little research has been done to determine what developers do in practice and why.

In this work, we focus on one aspect of encapsulation: the encapsulation boundary in OO systems. In the archetypal OO language Smalltalk, object data is private to an object. On the other hand, in statically typed OO languages such as C# and Java, object data is private to a class. This difference has broad implications for software design and maintenance, especially when inheritance is considered. Using a survey of both novice and experienced software developers, we show that the encapsulation boundary supported by mainstream statically typed languages does not coincide with the intuition of most developers.

Index Terms--Encapsulation, Encapsulation Boundary, OO design, Information Hiding

I. INTRODUCTION

Software systems are often large and very complex, making them difficult to comprehend and maintain. Programs commonly contain thousands or even millions of lines of code; far too many for any one person to understand. The difficulties that arise from the sheer size of software are compounded by the complexity that results from coupling between software components. “Programming is about managing complexity”, according to Bruce Eckel [3, page6]. Complexity in software systems leads to systems not meeting their specifications, suffering from quality problems, or even outright project failure.

A cardinal strategy used by software designers to control complexity is to decompose systems into loosely coupled components [20]. Parnas formulated this idea as the principle of Information Hiding [15], which encourages designers to hide implementation details so that the rest of the program cannot

depend on them. This reduces the cognitive load on developers because they can ignore hidden details when considering the services offered by a software component, and makes systems more amenable to change because hidden features can be modified without directly affecting client code.

Encapsulation is a programming language mechanism that enables Information Hiding, and so is arguably the most fundamental tool programmers have for managing complexity. Encapsulation mechanisms provide a way of establishing a boundary around a logical module of a system, and of hiding data and implementation details within that boundary to ensure that only the module that owns the information can access and modify it [1], [17].

OO systems provide several levels of encapsulation, usually in addition to conventional source code modules: Packages encapsulate classes, classes and objects encapsulate data and methods, and methods encapsulate algorithms. In this work we are interested in object and class level encapsulation of data, particularly in the presence of inheritance.

II. BACKGROUND

Many design heuristics, principles and guidelines have been proposed to help OO designers, including 61 “golden rules” of OO design introduced by Riel [16], heuristics by John Lakos [12], code smells by Fowler and Beck [4], and the advice of Ralph Johnson and Brian Foote [10]. Since encapsulation is such an important aspect of software design, many of these rules provide explicit or implicit guidance on how to practice encapsulation. Examples include the Separation of Concerns principle [2], the Law of Demeter [13], and several of Riel’s heuristics, such as “All data should be hidden within its class” [16]. Advice in this area sometimes conflicts, resulting in confusion on the part of designers, inconsistent code, and ultimately software that is harder to understand and maintain.

While there is universal acceptance of the value of encapsulating data to protect it from the rest of the system, there is no consensus among OO designers on where the encapsulation boundary should lie. Encapsulation is enforced in two main ways in modern OO programming languages; we will refer to them as *object encapsulation* and *class encapsulation*.

Object encapsulation is commonly used by dynamically typed languages, including Smalltalk [11] and Ruby [19]. In these languages, data is private to an object. This means that when an object contains data, only that object has the right to access and modify this data; it cannot be directly changed by any other object, regardless of that object’s class.

In contrast, today’s dominant statically typed OO programming languages, including Java [5], C# [6] and

Janina Voigt is a postgraduate student at the University of Canterbury, Christchurch, New Zealand (e-mail: jvo24@ student.canterbury.ac.nz).

Warwick Irwin and Neville Churcher are both with the University of Canterbury, Christchurch, NZ (e-mail: warwick.irwin@canterbury.ac.nz, neville.churcher@canterbury.ac.nz).

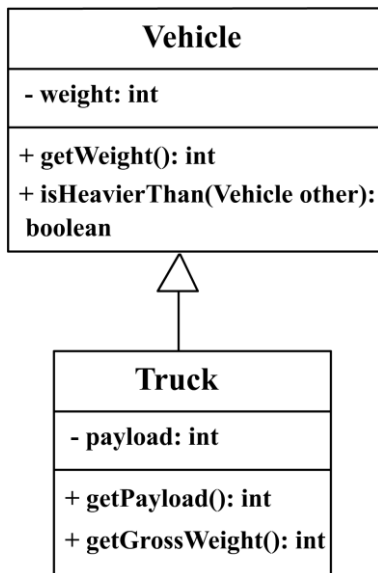


Fig. 1. A class view

C++[18], enforce class encapsulation: the most tightly protected data is private not to an object but to a class. By placing the encapsulation boundary around classes, these languages allow objects of the same class to access each others' private data.

This difference becomes more important when inheritance is considered. When using object encapsulation, a single object can access all of that object's data, regardless of which class it was inherited from. When using class encapsulation, objects cannot access private members inherited from a superclass, even though they are part of the same object. The class encapsulation boundary cuts the object, making the inherited part inaccessible to the derived part.

The protected access mechanism provided by many current OO programming languages allows an object of a subclass to access protected parts inherited from a superclass. This enables an approximation of object encapsulation, because if all inherited members are protected an object is able to access all of its contents. Nevertheless, this is still a variation of class encapsulation rather than true object encapsulation, because other objects of the same class can also access the object's contents. In some programming languages such as Java, protected access also confers access rights on all objects of other classes in the same package.

The two different types of encapsulation represent very different design philosophies. They are best explained using an example. Vehicles have a weight and can compare their weight to that of another Vehicle using the method `isHeavierThan(Vehicle other)`. The class also defines a simple accessor method (or getter) for the weight field. Trucks inherit from Vehicles and add their own field `payload`, which describes the maximum load a truck is allowed to carry. In addition to this field, the Truck class also defines a getter method for the `payload` field and a method called `getGrossWeight()` that can calculate the total

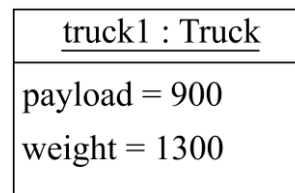


Fig. 2. An object view

weight of a Truck: its net weight plus its payload.

Fig. 1 shows a class encapsulation view of the example. Vehicle and Truck are different classes, each with their own sets of attributes and methods. These classes are the building blocks of the program when it is constructed. Class encapsulation reflects a designer's mindset oriented around static, compile-time concepts. According to this mindset, it makes little sense to allow classes to access other classes' private members.

Fig. 2 shows the object encapsulation view. In this paradigm, the Truck object is a single entity, in part defined by the Truck class and in part by Vehicle. This mindset is oriented around the runtime concept of objects. For this way of thinking, it does not make sense for a Truck to be able to access only a part of itself.

Both types of encapsulation have advantages and disadvantages. We have observed that many students who use Java are surprised when they find out that objects of the same class can access each others' private data. Many students seem to find object encapsulation intuitively correct, and assume that programming languages support it.

However, while object encapsulation may be more intuitive, Snyder (and others) argue that it removes all advantages to be gained from the use of encapsulation [17]. He suggests that by allowing access to data in another class—which may have been written by a different developer—the maintainability of the system is compromised. The reason he gives is that, should the other developer wish to change the internal data representation for that class, the subclass accessing that data will also be affected.

Similar arguments can be made in defence of object encapsulation, however. The extensibility and reusability of a software system might be enhanced by allowing objects of subclasses to access inherited implementation details; this gives them freedom to override or reuse existing members in ways that would be prevented by class encapsulation. Again, this is effectively a philosophical choice: object encapsulation is a permissive approach intended to maximise subclass freedoms, while class encapsulation favours tighter control. An issue that may influence a designer's choice here is the question of whether inheritance is innately such a strong dependency that it does not make sense to try to isolate a subclass from changes to its superclass.

As a first step in investigating whether our speculations on the encapsulation boundary are valid, we decided to conduct a survey to clarify how programmers use encapsulation in the

simple `Vehicle` scenario described above.

III. SURVEY

We designed a survey to investigate how novice and experienced software developers practise encapsulation. This section describes the goals we were trying to achieve, the survey participants and the tasks in the survey.

A. Goals

From personal experience and from working closely with computer science students, we suspect that many students learning object oriented programming using a programming language like Java or C++ tend to assume that `private` data is private to an object and are surprised and in some cases shocked when they learn that it is instead private to a class. Many of them feel uncomfortable when accessing the `private` data in another object of the same class. It seems that this conflicts with their intuitive expectation of where the encapsulation boundary should lie.

However, over time many appear to adapt to some degree to the tools a programming language provides them. They start to access `private` data from another object of the same class on occasion, particularly in places like the `equals()` method in Java, despite the fact that it conflicts with their intuition. We have heard people justify these decisions by saying that accessing `private` data is more efficient and quicker to code.

We decided to conduct a formal survey involving a number of novice and experienced programmers to clarify their encapsulation practices. We surveyed 34 undergraduate students, 9 postgraduate students and 12 professional developers about their practices of encapsulation. We expected to show that, while professional developers have adapted to the class encapsulation mechanism provided by most modern programming languages, object encapsulation makes intuitively more sense to novice programmers.

B. Participants

The survey was conducted with both undergraduate and postgraduate university students and professional software developers. The undergraduates were volunteers from two computer science courses at the University of Canterbury. The first course was a second-year course about computational theory. The students in the course had just completed their first year of computer science, including an introduction to Java but had relatively little experience, having not yet completed a programming project other than the usual small CS1 and CS2 assignments. The second class we surveyed was a third-year software engineering course. These students had all completed a second-year software engineering course which included a group project in Java where they developed software for a real client over a period of 6 weeks. We also surveyed postgraduate students who all had a substantial amount of experience using Java.

In addition to surveying students, we surveyed 12 professional software developers who routinely used C# as part of their work. They were likely to be far more proficient

programmers than undergraduate students and more aware of OO design principles, having programmed professionally from anywhere between 2 and 20 years.

C. Task

We carefully designed the survey to allow us to infer the encapsulation practices and principles of participants rather than asking them directly. We did not want participants to over-think their replies but rather to act as they would when programming. The two main parts of the survey can be seen in Fig. 3. The corresponding class diagram is the one already presented in Fig. 1.

The survey consisted of two main questions that were designed to exemplify the difference between object and class encapsulation. For each question, we presented a small class containing a few fields and methods. We then asked developers to complete a new method by choosing between three alternatives. Each alternative completed the method in a way that achieved similar functionality. However, the difference between the options was that some used getters to access fields while others accessed data directly.

For each of the two questions, we asked developers to rank the three options from 'best' to 'worst' and to explain the ranking they decided on.

The first question focused on the question of whether an object should be able to access the `private` data of another object of the same class; this is allowed in class encapsulation but not in object encapsulation. Subjects ranked, in order of preference, three given options for the completion of the `isHeavierThan(Vehicle other)` method which compared the weight of one `Vehicle` object to that of another `Vehicle`.

The second question focused on the question of whether an object should be able to access inherited `private` data; this is allowed in object encapsulation but not in class encapsulation.

Subjects ranked three given options to complete a method called `getGrossWeight()`, which returned the sum of the truck's weight (inherited from `Vehicle`) and payload (locally defined in `Truck`). Again, the only difference between the options was that some used getters while others accessed variables directly.

Many programmers will automatically invoke getters, if they exist, rather than accessing fields directly, and this convention might overpower any preference for a particular encapsulation boundary. Similarly, if programmers automatically access fields directly whenever possible, this convention may dominate any single encapsulation boundary. Both questions asked subjects to rank the alternatives, rather than simply pick a favourite, so that in a number of cases we were able to determine their encapsulation boundary preferences, even if they always favoured getters or direct access. In other cases, their comments gave clues about their way of thinking.

In addition to the two main questions, we included two very simple coding exercises—asking participants to write a `toString()` method for the `Vehicle` and `Truck` classes—to test the competence of the participants. These

Consider the following class Vehicle:

```
public class Vehicle {
    private int weight;

    public int getWeight() {
        return weight;
    }

    public boolean isHeavierThan(Vehicle other) {
        //Something goes here
    }
}
```

Now we want to complete the code for the `isHeavierThan(Vehicle other)` method of the Vehicle class.

Here are several ways in which we could complete this method:

Option 1

```
public boolean isHeavierThan(Vehicle other) {
    return other.weight < this.weight;
}
```

Option 2

```
public boolean isHeavierThan(Vehicle other) {
    return other.getWeight() < this.weight;
}
```

Option 3

```
public boolean isHeavierThan(Vehicle other) {
    return other.getWeight() < this.getWeight();
}
```

We will now extend our design by adding a second class, `Truck`, which is a subclass of `Vehicle`. The `Truck` class contains a field `payload` storing the maximum load the truck is allowed to carry.

```
public class Truck extends Vehicle {
    private int payload;

    public int getPayload() {
        return payload;
    }

    public int getGrossWeight() {
        //Something goes here
    }
}
```

Now we want to complete the code in the `getGrossWeight()` method of the `Truck` class. This method is supposed to calculate the gross weight of the truck, i.e. the weight of the truck plus the maximum load it can carry. Here are several different ways in which this method could be written:

Option 1

```
public int getGrossWeight() {
    return weight + payload;
}
```

Option 2

```
public int getGrossWeight() {
    return getWeight() + payload;
}
```

Option 3

```
public int getGrossWeight() {
    return getWeight() + getPayload();
}
```

Fig. 3. Main parts of the Encapsulation Questionnaire

questions enabled us to eliminate participants who did not have enough basic programming knowledge to competently complete the questionnaire. On the basis of these two questions, we eliminated two of the responses.

For the professional software developers, we also included a question about their previous programming experience, including their first programming language and the amount of time they had used C# or VB.NET. We translated the questionnaire from Java to C#, making sure that it was semantically identical to the Java questionnaire.

IV. RESULTS

We classified respondents by the encapsulation practices they preferred, as shown in Fig. 4. The results from students provide evidence to support our theory that novice programmers tend to find object encapsulation much more intuitive than class encapsulation. The undergraduate students could be divided into four major groups:

- Those who preferred using getters rather than accessing data directly. (No single encapsulation

boundary.)

- Those who practised object encapsulation.
- Those who preferred accessing data directly rather than using getters. (No single encapsulation boundary.)
- Those who did not mind whether getters were used or data was accessed directly as long as the approach used was consistent. (No single encapsulation boundary.)

More than half of the students (58 %) preferred using getters to accessing data directly. This is not surprising since they have been taught in class that getters make a system more maintainable. They commented that using getters was better style, safer and made the system more maintainable, and also said that getter methods encapsulate private data.

The second largest group (24%), practised object encapsulation. They preferred to access private data in a superclass directly, but did not want to access private data from another object of the same class. From their comments, it was evident that they truly believed that this was what Java allowed. They often commented that accessing private data in another

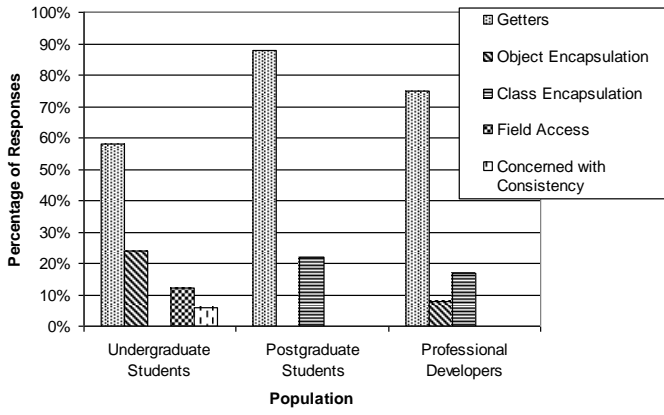


Fig. 4. Classification of survey responses

object of the same class was not possible.

The remaining groups were both small, with about 12 percent of students preferring to access data directly rather than using getters. They usually commented that this was more efficient. The last group of students (6%) were simply concerned with keeping the coding approach as consistent as possible.

Notably, there were no students who practised pure class encapsulation; that is, no one accessed the private data in another object of the same class but not the private data in a superclass.

Many of the students used getters to access data and therefore used neither object nor class encapsulation explicitly. However, in a number of cases their rankings still showed which they would chose if getters were unavailable, and their comments provided clarification of their reasoning. Using this information, we could classify respondents by whether they showed object or class encapsulation tendencies or both. The results of this additional analysis can be seen in Fig. 5.

For undergraduate students, two responses clearly showed partial object encapsulation thinking, with students commenting (incorrectly) that it was not possible to access a private field of another object of the same class. Another three responses showed a partial tendency toward class encapsulation, showing that these students were aware of Java's approach to encapsulation; they commented that a private field in a superclass could not be accessed directly. This is not a surprising response since they have been taught this in class. The remaining two students occupied an uneasy middle ground, showing tendencies towards both types of encapsulation, and were clearly confused about what Java allows.

We wanted to compare the way novice programmers think to how more experienced and professional software engineers think about encapsulation. We surveyed nine postgraduate students all of whom were very proficient in Java, and twelve professional software developers who were experienced .NET developers.

Interestingly, we found that none of the postgraduate students were using object encapsulation, but two used class encapsulation (22%). They commented that accessing the fields of another object of the same class directly was simple and valid

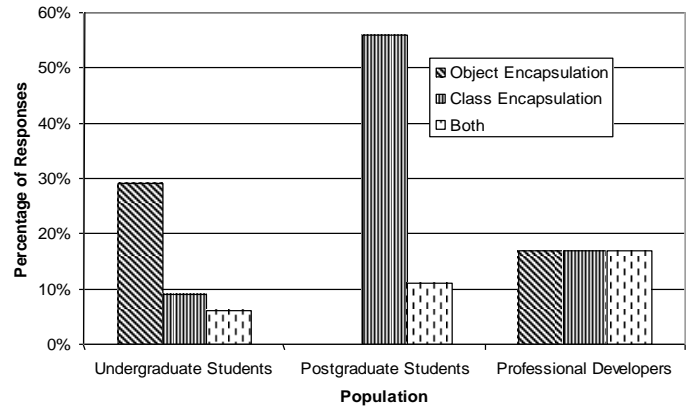


Fig. 5. Encapsulation tendencies for the three populations

while accessing the private fields in a superclass was not allowed. This clearly shows that they think differently from novice programmers. The remaining postgraduates (88%) preferred always using getters to support encapsulation.

We again had a closer look at the responses of the postgraduate students who used getters to see if we could infer more about their way of thinking. Three of the seven respondents who used getters showed definite class encapsulation tendencies, while another one showed tendencies both ways, and appeared to be confused about encapsulation in Java.

We saw a similar effect when we surveyed twelve professional .NET developers. The largest group (75%) again liked to always use getters. Two respondents (17%) used pure class encapsulation, clearly demonstrating that they were aware of what was valid in C#. Both commented that accessing private fields in a superclass was not valid and would not compile. One developer with 5 years experience using C# still believed that object encapsulation was correct.

A closer look at the responses of developers who used getters showed that even some professional developers are not completely comfortable with encapsulation in C#. One developer showed object encapsulation tendencies stating that accessing private fields of another object of the same class would cause a compile-time error. Two more developers showed both object and class encapsulation tendencies in their survey and appeared generally unsure about what was allowed and what was not.

V. DISCUSSION

The results from the student survey clearly support our contention that novice programmers find object encapsulation more intuitive than class encapsulation. More than a quarter of the students we surveyed with as much as two years of programming experience still believed that Java effectively supports object encapsulation. In addition, no students were comfortable using what Java provides: pure class encapsulation. Some students showed signs of being aware of encapsulation mechanisms in Java but no one wanted to use them.

This result has important implications because it shows that novice programmers are uncomfortable with the encapsulation mechanisms provided by many modern programming languages, including Java and C#. Object encapsulation, not class encapsulation, appears to make sense to them.

Even some postgraduate students and professional software engineers, all of whom were proficient in either Java or C#, showed signs of unease and confusion about the encapsulation mechanisms provided. Some did not appear to be entirely sure about what was allowed and what was not despite years of programming experience and the very basic nature of the exercises.

However, there was a clear sign that a number of them had adapted to what the programming language they were using provided them with, because around 20 percent used class encapsulation.

Overall, we believe that our survey shows that class encapsulation as provided by many modern programming languages is not what novice programmers expect and can confuse even experienced developers.

VI. CONCLUSION AND FUTURE WORK

We have conducted a survey amongst both novice and experienced programmers to determine how they practice encapsulation. We were particularly interested to find out if they preferred object or class encapsulation.

The difference between object and class encapsulation is that the encapsulation boundary is in a different place, making a different set of operations legal and illegal. Most modern programming languages like Java and C# use class encapsulation, while some languages like Ruby, Smalltalk and Java Script use object encapsulation.

Overall, our survey found that the class encapsulation mechanism provided by most of today's mainstream programming languages is unintuitive for novice programmers. While over time programmers appear to adapt to what the language allows them to do, there is still confusion amongst some experienced programmers as to what is allowed and what is not. We therefore argue that class encapsulation as provided by C# or Java is not what programmers intuitively expect or want.

This work is only the first step in our investigation into encapsulation practices. The results of this survey have provided us with useful insights into the issues surrounding encapsulation practices which will inform the next phase of our research. Because encapsulation is so fundamental, the lack of consistency uncovered by the survey has far-reaching consequences that impact on virtually all other principles and guidelines of OO design. We are currently developing tools to perform a quantitative analysis of encapsulation practices by extracting relevant data from the Qualitas Code Corpus of Java programs from the University of Auckland [14]. We have previously developed a very accurate semantic model for Java called Java Symbol Table (JST) [7] – [9] which we plan to use in the analysis of these programs. The aim will be to determine

whether object or class encapsulation is mostly used in real software and whether direct accesses of data are common. We expect the results to provide useful material for users and designers of OO languages.

REFERENCES

- [1] A. Cohen, "Data abstraction, data encapsulation and object-oriented programming," *ACM SIGPLAN Notices*, vol. 19, pp. 31–35, 1984.
- [2] E. Dijkstra, "On the role of scientific thought," in *Selected Writings on Computing: A Personal Perspective*, pp. 60–66, 1982.
- [3] B. Eckel and Z. Sysop, *Thinking in Java*. Upper Saddle River, NJ: Prentice Hall, 1998.
- [4] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.
- [5] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996.
- [6] A. Hejlsberg, M. Torgersen, S. Wiltamuth, and P. Golde, *The C# Programming Language*. Addison-Wesley Professional, 2008.
- [7] W. Irwin, "Understanding and improving object-oriented software through static software analysis", PhD thesis, University of Canterbury, 2007.
- [8] W. Irwin and N. Churcher, "Object oriented metrics: Precision tools and configurable visualisations," *METRICS2003: 9th IEEE Symposium on Software Metrics*, Press, pp. 112–123, 2003.
- [9] W. Irwin, C. Cook, and N. Churcher, "Parsing and semantic modelling for software engineering applications," *ASWEC '05: Proceedings of the 2005 Australian Conference on Software Engineering*, Washington, DC, USA: IEEE Computer Society, pp. 180–189, 2005.
- [10] R. Johnson and B. Foote, "Designing reusable classes," *Journal of Object-Oriented Programming*, vol. 1, pp. 22–35, 1988.
- [11] A. Kay, "The early history of Smalltalk," in *History of Programming Languages*, New York, NY, USA: ACM, 1996, pp. 511–598.
- [12] J. Lakos, *Large-Scale C++ Software Design*. Addison-Wesley, 1996.
- [13] K. Lieberherr and I. Holland, "Assuring good style for object-oriented programs," *IEEE Software*, pp. 38–48, 1989.
- [14] H. Melton and E. Tempero, "The CRSS metric for package design quality," *ACSC '07: Proceedings of the thirtieth Australasian conference on Computer science*, Darlinghurst, Australia: Australian Computer Society, Inc., pp. 201–210, 2007.
- [15] D. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, pp. 1053–1058, 1972.
- [16] A. Riel, *Object-Oriented Design Heuristics*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996.
- [17] A. Snyder, "Encapsulation and inheritance in object-oriented programming languages," *OOPLSA '86: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, New York, NY, USA: ACM, pp. 38–45, 1986.
- [18] B. Stroustrup, *The C++ Programming Language, Third Edition*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.
- [19] D. Thomas and A. Hunt, *Programming Ruby: The Pragmatic Programmer's Guide*. Addison-Wesley, 2001.
- [20] E. Yourdon and L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1979.

D Paper: Incest - Is it ok if you use protection?

This is the paper that we wrote about the results of the encapsulation analysis of real-world software and submitted to ASWEC 2010 (Australian Software Engineering Conference).

Incest: Is it OK, if you use protection?

Janina Voigt, Warwick Irwin, Neville Churcher
Department of Computer Science and Software Engineering
University of Canterbury
Christchurch, New Zealand
neville.churcher@canterbury.ac.nz

Abstract—This paper addresses a fundamental division in the way OO programming languages support encapsulation, and describes an empirical investigation into the way encapsulation is used in practice. The title - for which the authors apologise - refers to the encapsulation system in mainstream OO languages such as C# and Java, in which one object is allowed to touch another object's `private` parts if the objects are siblings (instances of the same class). We call this class encapsulation to differentiate it from object encapsulation, in which `private` data is accessible only within a single object. The use of protected data is disparaged in the class encapsulation paradigm and enforced in the object encapsulation paradigm. We find that current programming practice is arbitrary and inconsistent in languages that support class encapsulation. Developers appear confused about how best to employ the mechanisms offered by the languages.

Keywords-encapsulation; encapsulation boundary; OO design; information hiding;

I. INTRODUCTION

'Programming is about managing complexity', as Eckel succinctly states [1, page 6]. The most elemental means of managing complexity is decomposition; we divide programs into pieces and sub-pieces, and strive to keep them as independent as possible. Yourdan and Constantine's *cohesion* and *coupling* formalise these two aspects of decomposition [2].

Parnas provides guidance on how to decompose software, by *information hiding* [3], which advocates encapsulation of implementation decisions so that they cannot be subject to coupling from outside, and are free to change without impacting the rest of the program. Only relatively stable features of a program should remain externally visible. The *Stable Abstractions Principle* reinforces this aspect of information hiding [4], and the fundamental Computer Science concept of *Abstract Data Types* applies the principle to data structures.

These concepts were formulated during the ascendancy of procedural programming and structured software engineering. Object Oriented (OO) programming continued the progression of support for decomposition and introduced more sophisticated mechanisms for structuring programs in ways that allowed implementation details to be hidden, yet remain available for use. Inheritance and polymorphism, in particular, added a level of indirection between the caller of

a function and its implementation. In procedural systems, the caller does not know how a function is implemented; in OO systems, the caller does not even know which method it is calling.

Encapsulation is the core programming language mechanism that underpins this pre-eminent family of design principles, making it perhaps the most important semantic characteristic of programming languages. Because it is so fundamental, we might expect clear guidance on how to employ the encapsulation mechanisms provided by programming languages. We might also expect that as languages mature, they will converge on encapsulation mechanisms that encourage or enforce recommended practice. Among OO programmers, however, we have found a surprising lack of consensus over even the most basic of encapsulation questions [5]. What level of protection (`private`, `package`, etc) should be used for attributes? Should accessors be provided? If they are, should they also be used by an object to access its own data? Should subclasses call them? These are questions that must be answered by even novice programmers, yet experts don't agree on the answers.

In languages such as Java, beginners are commonly advised to declare attributes `private`, and to write getters and setters, which may be `public`. C# goes a step further and supports transparent use of getters and setters using the *property*¹ syntax [6]. But mechanically exposing attributes through accessor methods undermines information hiding; it is hiding of only the most superficial kind. A definitive characteristic of OO is the closeness of methods to the data on which those methods operate; a class defines a set of objects and the operations those objects can perform. The use of accessors is a sign that data is being manipulated in places other than within its owner object. Recognising this, the *Tell, Don't Ask* principle [7] advises designers to avoid using getters and operating on the returned data, and instead to instruct the object that already contains the data to do the work. A more extreme view is expressed in the *Law of Demeter* [8], which effectively prohibits the use of getters; an object may use only its own data, local variables and parameters.

¹The C# term *property* differs from the established OO usage we employ elsewhere in this paper to mean a member of a class.

Encapsulation is considerably more complex in the presence of inheritance. Should subclasses be allowed access to inherited attributes? In most statically typed OO languages, `protected` access provides a mechanism which makes this possible. In C# and C++ [9], for example, `protected` properties are accessible within the declaring class and all its subclasses. In Java, `protected` access also exposes properties to the declaring package; there is no mechanism to allow access only to subclasses.

Although these languages support it, the use of `protected` access is controversial. Riel, as one of his 61 ‘golden rules for OO design’, advises designers to ‘avoid `protected` data’, and instead make it `private` to its class [10]. Holub more emphatically states that ‘`protected` data is an abomination’ [11]. But despite the conviction of some commentators, this view is not universally accepted. Indeed, other OO cultures encourage or enforce the opposite rule. In Objective C, for example, the default access to attributes is `protected`. In Smalltalk there is no other choice; the only access mechanism for data is equivalent to `protected`, as the language does not provide a means of hiding inherited properties. But Smalltalk programmers do not describe encapsulation in these terms. On the contrary, they describe data as `private`, but it is `private` to an object, not a class.

A substantial lore of design advice is available to OO practitioners. We have already mentioned Riel’s 61 rules, but many others exist. We are advised to program to the interface, not the implementation, avoid inheritance for implementation, favour composition over inheritance, apply the Liskov Substitution Principle [12], [13], separate concerns [14], keep related code and data together [10], detect code smells [15], apply design patterns [16], and much more. The nature of design means that these maxims will inevitably exert conflicting pressures on designs, and that is as it should be. However, the conflicting advice about encapsulation is of a different nature; it results from a lack of consensus about how to structure software, rather than being a useful indicator of opposing forces. This is of some concern because encapsulation is such an elemental part of programming; how much value can we ascribe to more sophisticated layers of design advice if they are based on ill-considered foundations?

II. OBJECT ENCAPSULATION AND CLASS ENCAPSULATION

The field of software engineering suffers from a tendency for terminology that was once precise to accumulate shades of meaning. Even the term encapsulation is not immune. In [17], for instance, Rogers suggests that encapsulation means only grouping of properties, and that hiding is an orthogonal concept. We disagree; encapsulation is a mechanism that implies hiding. A more conventional definition is provided by Snyder [18]:

Encapsulation is a technique for minimizing interdependencies among separately-written modules by defining strict external interfaces. The external interface of a module serves as a contract between the module and its clients, and thus between the designer of the module and other designers.

This definition implies hiding: that which is not exposed in the interface is hidden, and cannot be the subject of a dependency from outside. Snyder places the encapsulation boundary around modules, but the definition of module in an OO context is not entirely clear. Our approach here is to work the other way: to recognise first the encapsulation boundaries supported by OO languages, and so demark the units in which hiding occurs.

In the archetypal OO language Smalltalk, everything is an object. Each object is encompassed by an encapsulation boundary. An object has access to all its properties and to no others. Goldberg and Robson explain the modularity of Smalltalk like this [19]:

The set of messages to which an object can respond is called its interface with the rest of the system. The only way to interact with an object is through its interface. A crucial property of an object is that its `private` memory can be manipulated only by its own operations. A crucial property of messages is that they are the only way to invoke an object’s operations. These properties insure that the implementation of one object cannot depend on the internal details of other objects, only on the messages to which they respond. Messages insure the modularity of the system because they specify the type of operation desired, but not how that operation should be accomplished.

This is *object encapsulation*. In object encapsulation, there is only one type of attribute access: attributes are (implicitly) `private` to their object.

Unlike Smalltalk, C++ augmented an existing language which already used an encapsulation approach based on static modules, and so it is perhaps unsurprising that C++ placed the encapsulation boundary around classes. In [9], Stroustrup makes this explicit: ‘Note that in C++, the class-not the individual object-is the unit of encapsulation.’ Elsewhere, Stroustrup comments that Smalltalk had little influence over the design of C++ [20].

The term *class encapsulation* describes the encapsulation boundary introduced in C++, and since adopted (with modifications) in Java, C# and other languages. Because it places the encapsulation boundary around a class, class encapsulation allows incest: two objects of the same class can access each other’s `private` properties. When inheritance is used, the encapsulation boundary cuts through objects, so that one part of an object can’t access other (inherited) parts of the

same object.

Early versions of C++ supported only `public` and `private` class members. Later versions of C++ introduced `protected` access, which allowed subclasses to access inherited properties, in order to mitigate the effects of partitioning objects according to class boundaries. Although `protected` allows an approximation of object encapsulation, it is not the same thing; incest can still occur (hence the deplorable title of this paper).

Although Stroustrup introduced `protected` access (he credits Mark Linton as co-inventor), he is tepid at best about its merits. In [21] he writes:

The alternative to `protected` data was claimed to be unacceptable inefficiency, unmanageable proliferation of inline interface functions, or `public` data. `Protected` data, and in general, `protected` members seemed the lesser evil. Also, languages claimed ‘pure’ such as Smalltalk supported this - rather weak - notion of protection over the - stronger - C++ notion of `private`. I had written code where data was declared `public` simply to be usable from derived classes. These were good arguments and essentially the ones that convinced me to allow `protected` members. However, I regard ‘good arguments’ with a high degree of suspicion when discussing programming. There seem to be ‘good arguments’ for every possible language feature and every possible use of it. In retrospect, I think that `protected` is a case where ‘good arguments’ and fashion overcame my better judgement and my rules of thumb for accepting new features.

A similar distaste for `protected` attributes is evident in much advice available to OO programmers, and in the casual way it is supported in Java, where it also opens properties to `package` access.

The difference between class encapsulation and object encapsulation is elemental and potentially has far-reaching consequences for software design, yet as far as we can tell, has largely escaped the attention of software practitioners. The term object-oriented is used without qualification to describe both approaches. Stroustrup, although he mentions the boundary, makes little of it.

The same, however, cannot be said of Alan Kay. In his keynote speech to OOPSLA97, Kay famously said ‘I made up the term object-oriented, and I can tell you I did not have C++ in mind’. We interpret subsequent passages of Kay’s speech to mean that the lack of object encapsulation is one of the main reasons he claims that C++ and Java - and by extension C# and others - are not legitimate OO languages. Kay contrasts a mechanical analogy of software with a cellular analogy:

If you take things like clocks, they don’t scale by a factor of a hundred very well. Take things like

cells, they not only scale by factors of a hundred, but by factors of a trillion, and the question is, how do they do it, and how might we adapt this idea for building complex systems? Okay, this is the simple one. This is the one, by the way, that C++ has still not figured out, though.

You must, must, must not let the interior of any one of these things be a factor in the computation of the whole. [...] The cell membrane is there to keep most things out, as much as it is there to keep certain things in.

Kay goes on to say:

The realization here [...] is that once you have encapsulated, in such a way that there is an interface between the inside and the outside, it is possible to make an object act like anything. The reason is simply this, that what you have encapsulated is a computer. You have done a powerful thing in computer science, which is to take the powerful thing you’re working on, and not lose it by partitioning up your design space. This is the bug in data and procedure languages. I think this is the most pernicious thing about languages like C++ and Java, that they think they’re helping the programmer by looking as much like the old thing as possible, but in fact they are hurting the programmer terribly by making it difficult for the programmer to understand what’s really powerful about this new metaphor.

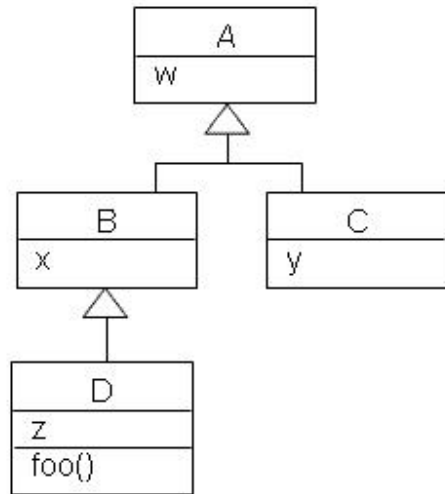
Kay’s distress is understandable. The analogy with cellular organisms was a main feature of his vision of computing from its earliest days [22]. The languages that now wear the OO mantle support the superficial form of his work, but not the vision, and few commentators seem to have noticed.

Figure 1a shows an example UML class diagram. The classes A to D represent the main units of class encapsulation. Figure 1b shows the same classes, and a number of instances, o_1 to o_4 , that have been created. These instances represent the units of object encapsulation. Classes are shown in this diagram as a way of grouping objects but not as encapsulation boundaries.

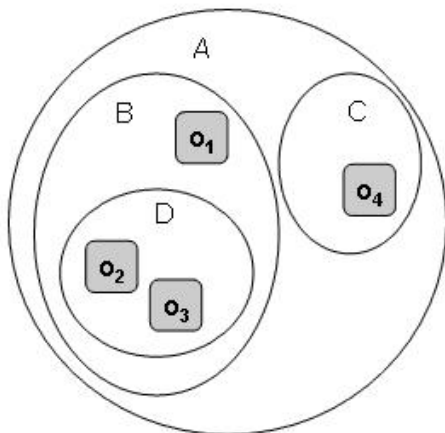
Figure 2 contrasts the two encapsulation boundaries for our example. From the point of view of object encapsulation, the class encapsulation boundaries cut through objects, meaning that one part of an object may be unable to access another part of the same object. From the point of view of class encapsulation, it makes sense for one object to access data of another object in the same class.

III. A PRELIMINARY SURVEY

In a recent paper [5], we reported the results of a survey of the encapsulation preferences of programmers who use Java and/or C#. The survey was designed to show whether



(a) An example UML class diagram



(b) Classes A to D and some instances of these classes

Figure 1. An encapsulation example

the programmers tended to use object encapsulation, class encapsulation, or a mixture of both. The subjects were drawn from three populations: undergraduate students, postgraduate students, and professional developers. As can be seen in Figure 3, we found different tendencies in the three populations. Novice programmers showed a preference for object encapsulation, despite having been taught that data is `private` to a class in Java. In contrast, most postgraduates had embraced the class encapsulation mechanism offered by the programming languages. Professionals showed diverse preferences. All three populations exhibited a degree of confusion over the languages' encapsulation boundary.

The survey revealed significant discord over the use of encapsulation in Java and C#, and confirmed our expectation that novice programmers find object encapsulation more intuitive. This is not greatly surprising, given the parallels between OO and real-world classification. In [23], Coad and

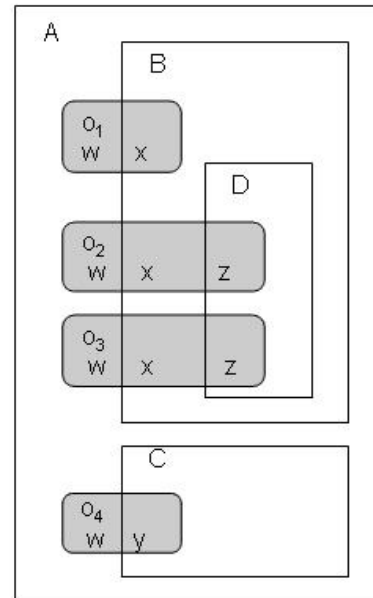


Figure 2. A view of different encapsulation boundaries

Yourdon quote the 1986 Encyclopaedia Britannica entry on Classification Theory [24]:

In apprehending the real world, [people] constantly employ three methods of organisation, which pervade all of their thinking:

- 1) the differentiation of experience into particular objects and their attributes – eg. when they distinguish between a tree and its size or spatial relations to other objects.
- 2) the distinction between whole objects and their component parts – eg. when they contrast a tree with its component branches, and
- 3) the formation of and the distinction between different classes of objects – eg. when they form the class of all trees and the class of all stones and distinguish between them.

This description, which was presumably written by authors with no knowledge of programming, is immediately recognisable by OO programmers. This is not a coincidence, of course: OO simply imports this extant way of organising systems into programming languages. This is what makes object encapsulation intuitive; it echoes the boundaries between real-world objects.

IV. MEASURING ENCAPSULATION PRACTICES IN JAVA

Following the survey, we decided to investigate encapsulation practices in real software to determine what developers do in practice (as opposed to what they say they do when surveyed). To this end, we wrote a static analysis tool to measure encapsulation in Java programs.

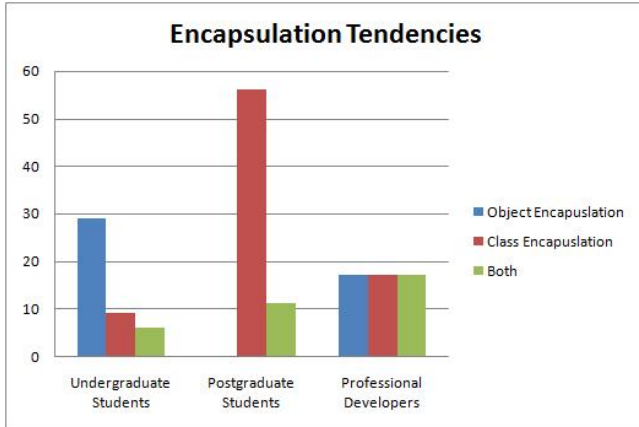


Figure 3. Survey results

We carefully considered what data to collect in order to relate empirical results to the survey results. We chose to focus our attention on encapsulation of data in the first instance; we will expand the study to include method encapsulation in the near future. Encapsulation of data is more emphatically stressed by OO design guidelines and more readily grasped by programmers, so it is likely to support more definitive conclusions.

Our tool measures two aspects of a program: the levels of protection accorded to attributes, and the ways in which attributes are actually accessed. (This allows us to tell, for example, if an attribute has been given wider scope than is used in practice, such as when a package-accessible attribute is only ever used locally in its class.)

To characterise protection levels, we count the number of attributes in a program with `public`, `package`, `protected` and `private` access. These numbers give a good overview of how rigorously data is hidden from the outside world.

Characterising actual accesses to attributes is a little more complex. Our program accumulates the number of accesses to `public`, `package`, `protected` and `private` attributes. It also counts the number of accesses that originate inside and outside the object that contains the attribute, and the number of accesses that originate inside or outside the class that defines the attribute. This allows us to count the number of accesses that cross both types of encapsulation boundary.

Accesses from outside a class that defines an attribute are easy to find. However, a more sophisticated approach is required to determine if an access comes from outside an object. Because we perform static analysis of source code, objects - which are a runtime concept - do not yet exist and it is virtually impossible to determine precisely whether a reference refers to the same object as the one doing the accessing. We instead employ a simple heuristic

which will work correctly in the vast majority of cases. If an access originates in a class that is neither the class that defines the attribute, nor a subclass of the class that defines the attribute, then the access must come from outside the object. If, however, the access originates in the same class or a subclass, the access may or may not come from the same object. If this is the case, we check exactly how the access was done in code. If the access is of the form *fieldName* (without a qualifier), the access comes from within the same object. Similarly, if the access is of the form *this.fieldName* or *super.fieldName*, the access comes from within the same object. On the other hand, if the access is of the form *qualifier.fieldName*, we presume the access comes from outside the object.

While this strategy works well for the vast majority of cases, there are exceptions. The most common occurs with inner classes, which can access attributes in the outer class. This syntax appears to be an access from within an object when it is really an access from a different (inner) object. However, we decided that this situation was sufficiently rare (and also outside our simple formulation of the concept of an object encapsulation boundary) that our straightforward heuristic would provide an acceptable approximation for a first study. We plan to refine this aspect of our instrument in later versions.

Our program has the ability to analyse the data it has collected and report on what encapsulation the program uses: object or class encapsulation. In many cases, we expect that a mixture of the two will be used, and in such cases our program reports to what degree the two types of encapsulation are used.

We implemented our program using Java Symbol Table (JST), a semantic model of Java developed by Irwin and Churcher [25]–[27], to extract information about Java programs. JST constructs a model of the semantic structure of a program, representing concepts such as packages, classes, methods, constructors, parameters, attributes and local variables. The relationships between these entities are also captured by the model.

We used the latest version of the Qualitas Code Corpus produced by the University of Auckland as a repository of real-world software to be analysed [28]. This version of the corpus contains 100 Java projects of diverse provenance, including some very well-known programs such as Eclipse and ANTLR.

The Java version for which these programs were written varies from Java 1.1 to Java 1.6. Our Java parser is generated from the grammar for version 1.6. In many cases, Java is sufficiently backwards-compatible that our parser and JST can handle older source code, but some programs contain syntax that has been made illegal by changes in the Java language. The most common syntax error - which prevents about 20 programs from compiling - is caused by the introduction of `enum` as a keyword. Early versions of Java provided the

unfortunately named `Enumerator` which was later deprecated in favour of `Iterator`, and local variables of the `Enumerator` type were commonly named `enum`, leading to the name clash. The late introduction of the `assert` keyword produced a similarly widespread problem. We chose to exclude these programs from our experiments because they are no longer correct Java.

For this experiment, we chose to analyse 34 programs from the corpus; these were the ones for which the complete source code could be processed by the current version of our tools without difficulty. We also analysed an additional 11 student programs to see if there was a similar difference between students' and professionals' encapsulation practices to that found by the survey. The student programs were each produced by a group of 6-7 second-year software engineering students as part of a semester-long project for real clients.

V. RESULTS AND ANALYSIS

In the 34 corpus programs, the number of attributes ranged from 69 to 2159, while the number of accesses to attributes ranged from 355 to 10818. In the 11 student programs, the number of attributes ranged from 55 to 469, while the number of accesses to attributes ranged from 208 to 1973.

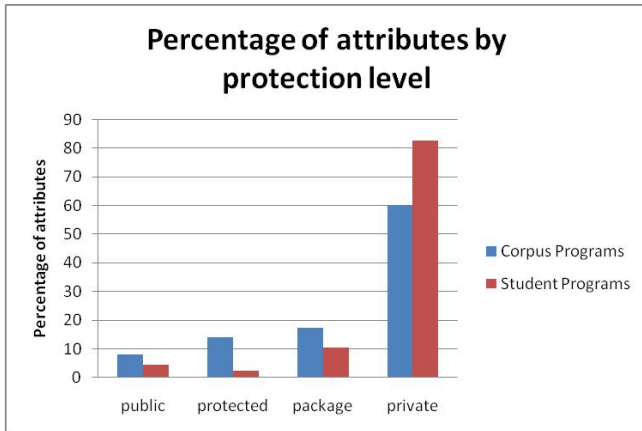


Figure 4. Use of protection levels in real and student programs

Figure 4 shows the relative numbers of different protection levels used in corpus and students programs; Figure 5 shows the relative numbers of accesses to attributes with those different protection levels. Unsurprisingly, the two graphs have very similar shapes.

Clearly, `private` is the most frequently declared and the most heavily accessed protection level. This tendency is more pronounced in student programs than in corpus programs, where 40% of attributes are not `private`. This suggests that student programs are more tightly encapsulated on average.

It is also interesting to note that students rarely declared `protected` attributes and that corpus programs tended to

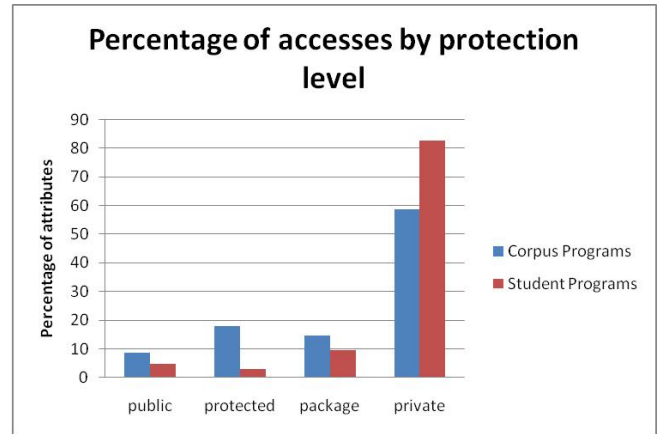


Figure 5. Accesses in real and student programs by protection level

access `protected` attributes somewhat more frequently than other types.

	Average (%)	Minimum (%)	Maximum (%)	Standard Deviation
Public	8.8	0	63.6	13.9
Protected	18.0	0	61.9	20.0
Package	14.6	0	69.7	14.6
Private	58.6	1.9	99.6	27.6

Table I
PROTECTION LEVEL STATISTICS FOR DECLARATIONS IN CORPUS PROGRAMS

	Average (%)	Minimum (%)	Maximum (%)	Standard Deviation
Public	4.6	0	20.5	5.8
Protected	3.0	0	13.4	4.9
Package	9.6	0.2	42	11.8
Private	82.8	60.1	94.5	11.0

Table II
PROTECTION LEVEL STATISTICS FOR DECLARATIONS IN STUDENT PROGRAMS

The data shown in the above figures is aggregated over all systems, but does not show the variations between programs. We found a great diversity of encapsulation practices in the corpus programs but relatively consistent practices in the student programs, as can be seen in Table I and Table II. This no doubt reflects the greater variety of domains, purposes and scales of the corpus programs as well as the diversity of the developers.

A notable characteristic of this data is the very high standard deviations of the corpus programs' use of protection levels. `Private` data in particular spans a range from virtually no use to almost exclusive use. This is evidence that encapsulation practices in industry are inconsistent.

We found similar levels of variation in the number of accesses to attributes. Of particular note was:

- Public attributes unsurprisingly were used quite heavily from outside the class that declared them (57.5% for corpus programs and 40.8% for student programs), and were also used heavily internally.
- In Java, the `protected` access modifier gives access rights to subclasses, as well as to other classes in the same package. We found that in corpus programs subclass access was used much more common (27.9%) than same-package access (5.6%). The remaining accesses were from within the declaring class. Student programs, however, revealed a different picture. Out of the eight programs that used the `protected` protection level, two used it as package access, four used it as `private` access, and two used it as subclass access. This suggests a considerable degree of confusion among students regarding Java's `protected` access mechanism.
- Package attributes were much less commonly accessed from outside the class in which they were declared (averaging 20.1%). In Java, package access is the default protection level, and it seems likely that this level of access has been granted in many cases by developers forgetting to specify tighter access. This is the case in student programs in particular, where 6 out of 11 systems never accessed package attributes from outside the declaring class.
- Because Java uses class encapsulation, `private` attributes can be accessed from other objects of the same class. We found that almost all corpus and student programs made some use of this; however, the average percentage of accesses to `private` attributes from other objects was very low (3.0% and 1.3% respectively).

	Category of Access	Percentage
1	Same object, same class	82.6
2	Same object, superclass	6.6
3	Different object, same class	2.7
4	Different object, superclass	0.2
5	Different object, different class	7.8

Table III
PERCENTAGE OF ACCESSES BY CATEGORY IN CORPUS PROGRAMS

	Category of Access	Percentage
1	Same object, same class	93.7
2	Same object, superclass	0.3
3	Different object, same class	1.5
4	Different object, superclass	0.0
5	Different object, different class	4.5

Table IV
PERCENTAGE OF ACCESSES BY CATEGORY IN STUDENT PROGRAMS

Figure 6 shows the main categories of access we measured. Table III and Table IV show what percentage of all

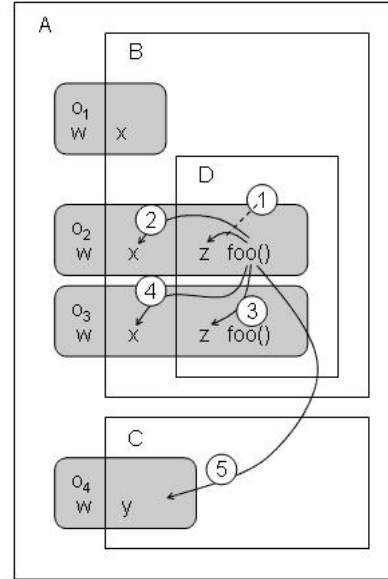


Figure 6. An overview of access categories

accesses belonged to each category in the corpus and student programs.

Unsurprisingly, in both populations the dominant category of access is within the same object and class. This category does not reveal anything about encapsulation boundary preferences as it crosses no encapsulation boundaries. Categories 4 and 5 similarly do not yield useful information about encapsulation boundary preference as the accesses in these categories cross both kinds of boundary. Interestingly, category 4 accesses are much less frequent than category 5, suggesting that developers are more averse to accessing superclass data than data in a completely unrelated class.

Category 2 accesses cross the class boundary but not the object boundary, indicating the use of object encapsulation. Category 3 is the inverse, indicating the use of class encapsulation.

In corpus programs, when an encapsulation boundary preference is evident, object encapsulation (6.6%) is used more than twice as much as class encapsulation (2.7%). This is consistent with our expectations and earlier survey results showing that object encapsulation is more intuitive.

In student programs, the number of accesses in Category 2 and Category 3 suggest the opposite result: class encapsulation appears to be preferred. This conflicts with our findings from the survey where object encapsulation was overwhelmingly preferred by students. This difference could be explained by the fact that the scenario in the survey was a lot simpler, the students' programs show evidence of general confusion about encapsulation mechanisms in Java, and the total number of accesses in Category 2 and Category 3 was very low.

The great majority of accesses in both populations either cross no encapsulation boundary or both kinds, which confirms our earlier observations that current encapsulation practice is inconsistent. No systems measured used either type of encapsulation exclusively, although some showed a strong preference for one or the other. The percentage of accesses crossing a class encapsulation boundary ranged from 0.3% to 39.9% for corpus programs and 1.3% to 6.1% for student programs. The percentage of accesses crossing an object encapsulation boundary ranged from 2.0% to 40.2% for corpus programs and 3.8% to 15.4% for student programs.

Some corpus programs were notable for having large numbers of accesses from outside both the class and the object. For example, the highest number of class and object boundary crossings (39.9% and 40.2%) occurred in one program. This indicates not only that the data is barely protected and encapsulation is very loose, but also that the data is poorly distributed amongst the classes because the program's behaviour is not located with the data on which it acts.

When `protected` access is used, it tends to be used for object encapsulation. For both populations, the access to `protected` attributes from outside the object was less common (8.3% for corpus programs and 4.5% for student programs) than for attributes with other protection levels.

VI. CONCLUSIONS AND FUTURE WORK

Encapsulation is a fundamental mechanism for controlling complexity in programs. However, the units of encapsulation employed by OO programming languages differ; some languages place the encapsulation boundary around classes, and some around objects.

We measured many Java programs and found incoherent encapsulation practices, not only between programs but within programs. Neither class nor object encapsulation was practised consistently in any of the programs. It also appears that common advice to make attributes `private` was not consistently followed; on average 58%, and in the worst case 1.9%, of attributes were `private`. Student programs were more tightly and more consistently encapsulated, no doubt because students had recently been instructed to program this way. Even so, students broke the rule in 17% of declarations.

Protection levels of attributes can be used to enforce particular encapsulation practices but even in the absence of these protections the same encapsulation boundaries can be respected by simply choosing not to access the attributes. We found, however, that accessible attributes did tend to be accessed. For example, approximately half the accesses to `public` attributes came from outside the declaring class. Similarly, accesses to `protected` attributes came from subclasses approximately 30% of the time.

The dominant encapsulation practice is to access attributes from within the object and class that declares it. Here,

the encapsulation boundary is effectively the intersection between the object and class boundary. We refer to this as *Intersection Encapsulation*. Intersection encapsulation is likely to be a strategy adopted in response to the general confusion around encapsulation boundaries. It represents the common ground between developer's intuition and the programming language mechanism for encapsulation. It is safe because it crosses no boundaries. However, this is a restrictive approach because it provides minimal access to attributes, which in turn may lead to heavier use of accessors and mutators, and hence an effective weakening of encapsulation.

In those programs which did exhibit a preference for object or class encapsulation, the majority of programs tended towards object encapsulation. In corpus programs, 6.6% of accesses showed an object encapsulation tendency, while only 2.7% showed a class encapsulation tendency. This adds weight to the findings of our previous study.

Class encapsulation allows objects of the same class to access each other's `private` attributes, but this ability is rarely used in practice. In corpus programs only 3% of accesses to `private` fields come from outside the object. This suggests that there is a certain level of uneasiness among developers regarding this access mechanism.

Java defaults to `package` access. This default supports neither object nor class encapsulation. It appears that in many cases omission of the access modifier is in fact unintended, particularly in student programs.

In C++ and C# it is possible to grant access to subclasses exclusively but in Java `protected` access also grants `package` access. In practice, however, `protected` attributes tend not to be accessed outside the class hierarchy and Java's `protected` mechanism appears to cause confusion among students.

In the near future, we intend to investigate the use of protection levels on methods in Java to broaden our understanding of encapsulation practices beyond attributes.

We are working on tools to automatically tighten encapsulation so that a consistent encapsulation policy will be applied throughout a program. These policies include object encapsulation, class encapsulation and intersection encapsulation.

In the absence of these tools, however, developers could significantly improve the quality of their programs by being more aware of their encapsulation practices and consciously choosing which boundary to apply.

We would hope that in the future programming languages will be designed to more closely match the expectations of programmers. This may be a relatively simple change to existing languages such as Java. For example, Java could be made to support object encapsulation by eliminating syntax that accesses a field of any object other than this, and removing the class encapsulation access levels.

REFERENCES

- [1] B. Eckel and Z. F. Sysop, "Thinking in Java," *PTR, Upper Saddle River, NJ*, 1998.
- [2] E. Yourdon and L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1979.
- [3] D. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.
- [4] R. Martin, "Stabilitythe Stable Dependencies and Stable Abstractions Principles," *C++ Report*, vol. 9, no. 2, February 1997.
- [5] J. Voigt, W. Irwin, and N. Churcher, "Intuitiveness of class and object encapsulation," *ICITA 2009*.
- [6] A. Hejlsberg, M. Torgersen, S. Wiltamuth, and P. Golde, *The C# programming language*. Addison-Wesley Professional, 2008.
- [7] "Tell don't ask," <http://c2.com/cgi/wiki?TellDontAsk>, 2008.
- [8] K. Lieberherr and I. Holland, "Assuring good style for object-oriented programs," *IEEE Software*, pp. 38 – 48, 1989.
- [9] B. Stroustrup, *The C++ programming language, third edition*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.
- [10] A. Riel, *Object-Oriented Design Heuristics*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996.
- [11] A. Holub, "Why extends is evil," *Java World*, 2003.
- [12] B. Liskov, "Data abstraction and hierarchy," *ACM SIGPLAN Notices*, pp. 17 – 34, May 1987.
- [13] R. Martin, "The Liskov Substitution Principle," *C++ Report*, vol. 8, no. 3, pp. 16 – 17, 20 – 23, 1996.
- [14] E. Dijkstra, "On the role of scientific thought," *Selected Writings on Computing: A Personal Perspective*, pp. 60 – 66, 1982.
- [15] M. Fowler, *Refactoring: improving the design of existing code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.
- [16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
- [17] P. Rogers, "Encapsulation is not information hiding," *Java World*, May 2001.
- [18] A. Snyder, "Encapsulation and inheritance in object-oriented programming languages," *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pp. 38–45, 1986.
- [19] A. Goldberg and D. Robson, *Smalltalk-80: The Language*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989.
- [20] B. Stroustrup, "Bjarne Stroustrup's FAQ," http://www.research.att.com/~bs/bs_faq.html, 2009.
- [21] —, *The design and evolution of C++*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1994.
- [22] A. Kay, "The early history of Smalltalk," pp. 511–598, 1996.
- [23] P. Coad and E. Yourdon, *Object-oriented design*. Upper Saddle River, NJ, USA: Yourdon Press, 1991.
- [24] I. Encyclopaedia Britannica, *Encyclopaedia Britannica*, Chicago, Illinois, 1986.
- [25] W. Irwin, "Understanding and improving object-oriented software through static software analysis," Ph.D. dissertation, University of Canterbury, 2007.
- [26] W. Irwin and N. Churcher, "Object oriented metrics: Precision tools and configurable visualisations," in *METRICS2003: 9th IEEE Symposium on Software Metrics, IEEE*, pp. 112–123, 2003.
- [27] W. Irwin, C. Cook, and N. Churcher, "Parsing and semantic modelling for software engineering applications," *ASWEC '05: Proceedings of the 2005 Australian conference on Software Engineering*, pp. 180–189, 2005.
- [28] Q. R. Group, "Qualitas corpus version 20090202," <http://www.cs.auckland.ac.nz/~ewan/corpus>, February 2009.

E A Guide to Using Java Symbol Table to Analyse Software

This is the guide we wrote about using JST to analyse a program in order to clarify the structure of JST for future researchers.

A Guide to Using Java Symbol Table to Analyse Software

Janina Voigt

July 11, 2009

Overview of JST

Java Symbol Table (JST) is a semantic model for Java, developed by Irwin and Churcher [2, 3, 4]. It constructs a model of a Java program in memory, capturing various semantic concepts. This includes concepts such as packages, classes, methods, constructors, parameters, fields and local variables. The relationships between these entities are also represented by the model.

JST is a much richer model than other existing Java semantic models like `Javasrc`. These models often only include simple relationships between entities such as method invocation and commonly struggle to resolve polymorphic and inherited method calls, leading to an inaccurate model. By using JST, you can get a very accurate model of your program, allowing you to analyse it very carefully.

JST currently accepts valid source code written in any Java version up to Java 1.6.

A UML class diagram giving an overview of the structure of JST can be seen in Figure 1.

Despite the size and complexity of JST, information can be extracted from JST quite easily by ‘walking’ the semantic model. This can be done using a Visitor design pattern [1] as demonstrated in previous work [5, 6]. This article explains how to write a simple visitor to extract information from the latest version of JST (for programs using up to Java 1.6).

Running JST

In order to be able to create a JST model of your program in memory, you first need to parse the Java files into parsetrees that JST can understand. You can do this by running a parser created using `Yakyacc` [?] over your program. This parser reads in the Java files and writes the resulting parsetrees out to XML. If there are no parse errors, you can run JST giving it the location of the XML files containing the parsetrees for your program. This will build a model of your program in memory, allowing you to manipulate and analyse it.

The Visitor Design Pattern

We recommend that you use a visitor design pattern to visit the different parts of the JST model. This is quite easy to do by extending an existing JST visitor as explained below.

The visitor design pattern has a number of advantages, including that it cleanly separates the JST model code from your visitor logic. Your visitor program should only need to go through the public interface of the model, thus making it easy to upgrade your program when a new version of JST comes out.

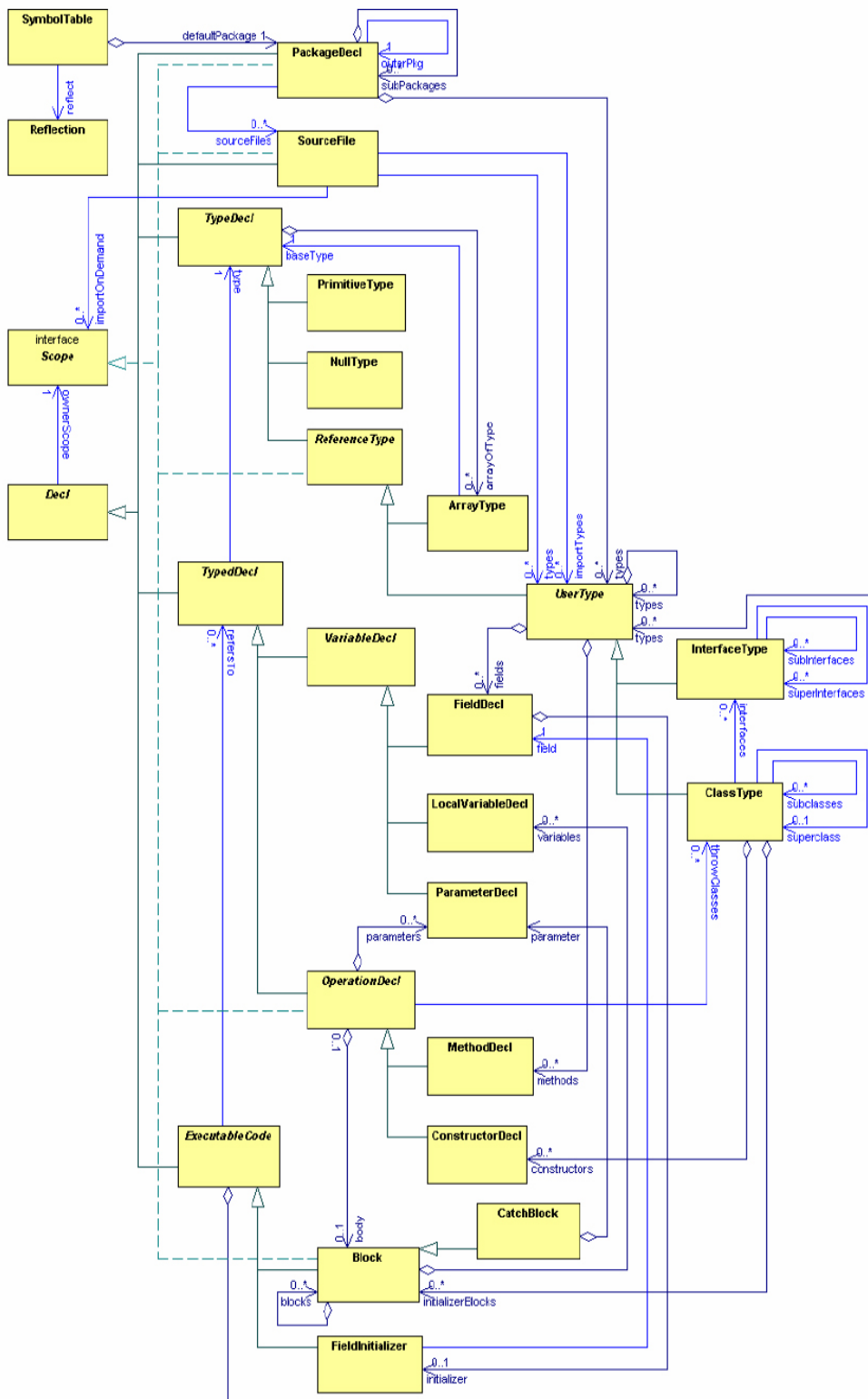


Figure 1: A UML class diagram of JST

JST Visitors

JST already contains a very simple visitor which walks the entire semantic model in a logical order. This visitor does not implement any specific operations and only contains the code for visiting the model. By subclassing this visitor, you can override the operations that deal with parts of the model you are interested in without having to worry about the rest. This makes writing visitors for JST very fast and simple.

The basic visitor is called `CompositionVisitor` and can be found in the package `jst.symtab.visitor`. It contains different methods for visiting each relevant part of the semantic model. Do not change the code in this class unless you know what you are doing as it may alter or break the visiting process. Instead, subclass this class to create your own visitor.

Once you have subclassed `CompositionVisitor`, you have to decide which parts of the model you want to visit. This will decide which methods you need to override in your visitor. To understand exactly which Java concept each of the parts of JST represents, a thorough knowledge of Java is required. This particularly applies if you are dealing with Java generics which are quite complicated.

The different parts of the model which can be visited using your visitor are:

- `Decl` represents any kind of declaration and is an ancestor of all other classes described below. Each `Decl` has a name and a scope.
- `PackageDecl` represents a package declaration. A package may have sub-packages (of type `PackageDecl`) and contains source files (of type `SourceFile`).
- `SourceFile` represents an entire Java source file which may contain any number of classes or interfaces. A source file knows about the types that have been imported (of type `UserType`) and resolves these imports so that the correct types can be found.
- `TypeDecl` is a superclass for all declarations which represent a type such as classes, interfaces, arrays and primitive types.
- `PrimitiveType` represents a primitive type such as `int`, `float`, `boolean` or `double`.
- `NullType` simply represents the type of the keyword `null`.
- `Actual Type` is a type which can be used as a type argument for Java generics. This includes reference types such as class types, interface types and array types as well as wildcard types. For example, it is possible in Java to declare a set of students (`Set<Student>`) but not to declare a set of ints (`Set<int>`) because `int` is a primitive type, not a reference type or wildcard.
- `ReferenceType` represents a type which is stored and passed by reference by Java, such as class types and array types. Most types, that is subclasses of `TypeDecl`, are also reference types, except for `PrimitiveType` and `NullType`.
- `ArrayType` represents an array type, such as an array of strings.
- `UserType` represents any class or interface, which may be part of a library or declared by a user. It contains fields (of type `FieldDecl`) and methods (of type `MethodDecl`) and may contain other types (of type `UserType`), including anonymous types.

- `InterfaceType` represents an interface. Interfaces know which other interfaces (of type `InterfaceType`) they implement.
- `GenericInterfaceType` represents an interface type which has some type parameters that are provided when an object of that class is created. For example, `Comparable<T>` is a generic interface with type parameter `T`.
- `ParameterizedInterfaceType` represents the type that results when instantiating a generic interface. For example, `Comparable<T>` is a generic interface and `Comparable<String>` is the parameterized interface type that results from providing `String` as a type parameter.
- `ClassType` represents any class, whether library classes or classes declared by the user. Classes know their superclass (of type `ClassType`) and the interfaces (of type `InterfaceType`) they implement. They also contain constructors (of type `ConstructorDecl`).
- `GenericClassType` represents a class type which has some type parameters that are provided when an object of that class is created. An example of a generic class is `HashSet<T>` which has a type parameter `T`.
- `ParameterizedClassType` represents the type that results from constructing an object of a generic class. For example, `HashSet<T>` is a generic class and `HashSet<String>` is the parameterized class type that results from providing `String` as a type parameter.
- `TypedDecl` is a superclass for declarations which have a type, such as variable and field declarations. As opposed to `TypeDecl`, these declarations don't declare a particular type but have a particular type.
- `VariableDecl` represents a variable declaration, such as a field, local variable or parameter declaration.
- `FieldDecl` represents a field declaration. A field may have a field initialiser (of type `FieldInitializer`) which represents the expression used to initialise the field when it is declared.
- `LocalVariableDecl` represents a local variable declaration.
- `ParameterDecl` represents a parameter declaration.
- `OperationDecl` represents an operation declaration, such as a method or a constructor declaration. An operation has a set of parameters (of type `ParameterDecl`), a body (of type `Block`) and a set of exceptions that it may throw (of type `ClassType`).
- `MethodDecl` represents a method declaration and also contains some information about the methods overridden and hidden by this method (of type `MethodDecl`).
- `ConstructorDecl` represents a constructor declaration.
- `ExecutableCode` represents sections of code which can be executed, such as method blocks; that is the code that is actually inside a method. An executable piece of code knows about the other parts of the model it refers to (of type `TypedDecl`) such as variables or methods.

- **Block** represents a block of code; that is the code between a pair of matching braces. This includes whole methods, if-statements and loops. A block can have other blocks (of type **Block**) and local variable declarations (of type **LocalVariableDecl**) inside of it.
- **CatchBlock** represents a catch block that is used to catch exceptions. A catch block contains a parameter declaration (of type **ParameterDecl**).
- **FieldInitializer** represents a field initialiser; that is the expression that is used to initialise the value of a field when it is first declared.
- **WildcardType** represents a wildcard for Java generics. Using generics, we can declare a set of students (`Set<Student>`) where `Student` is a reference type. Alternatively, we can declare a set that can hold students and objects of student subclasses (`Set<? extends Student>`). The type `?` extends `Student` is a wildcard type that matches `Student` or any descendants of `Student`. We say that the wildcard type has an upper bound of type `Student`.
- **TypeParameter** can be used in a generic class to describe the parts of the class (including variable types or method return types) that are not yet known until an object of the class is created. For example, the generic class `HashSet<T>` uses a type parameter `T` to define some of its parameters and return types. The actual type of `T` is not known until a `HashSet` is created. If we then create a `HashSet<String>`, `String` is substituted for `T`. `T` is the type parameter.

When you have decided which parts of the model you want to visit, you need to override the corresponding methods in your visitor. Make sure that you call the corresponding method in `CompositionVisitor` at some point in the overridden method, as there may be important code to visit other parts of the model in `CompositionVisitor`.

When writing the code to visit each part of the model, it is a good idea to have a look at the public interface of the corresponding JST class to see what methods you can use. Unfortunately, there are far too many methods to describe them all here but finding out what you can and need to do is usually relatively easy.

References

- [1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
- [2] Warwick Irwin. *Understanding and Improving Object-Oriented Software Through Static Software Analysis*. PhD thesis, University of Canterbury, 2007.
- [3] Warwick Irwin and Neville Churcher. Object oriented metrics: Precision tools and configurable visualisations. In *in METRICS2003: 9th IEEE Symposium on Software Metrics, IEEE*, pages 112–123. Press, 2003.
- [4] Warwick Irwin, Carl Cook, and Neville Churcher. Parsing and semantic modelling for software engineering applications. In *ASWEC '05: Proceedings of the 2005 Australian conference on Software Engineering*, pages 180–189, Washington, DC, USA, 2005. IEEE Computer Society.

F A Guide to Using JST Parse Trees for Code Generation

This is the guide we wrote about using JST parse trees which can be used for code generation. It is intended to allow future researchers to more easily get started using JST parse trees.

A Guide to Using JST Parse Trees for Code Generation

Janina Voigt

November 3, 2009

Overview of JST Parse trees

Java Symbol Table is a Java semantic model, representing a variety of semantic Java concepts including classes and methods. It reads a program in form of parse trees. These parse trees can be easily generated from .java files by using a Java parser generated using the yakyacc parser generator [1]. Such a parser will output the resulting parse trees in the form of an .xml file which JST can understand and read.

By simply giving the location of these XML files to JST as a command line argument, you can ask JST to read the parse trees and build a semantic model of the program in memory [1, 2, 3].

JST is a semantic model, meaning that its in-memory model of a program consists of all the important semantic Java concepts like classes and methods, making it easy to analyse the structure of the program. Each of these semantic concepts knows about its source; that is the part of the parse tree from which it was constructed. This makes it easy to find the correspondence between parse trees and JST model entities.

Most of the time when using JST you will not need to use parse trees, meaning that you will not need to understand their structure. However, parse trees can be very useful for code generation. It is relatively easy to analyse the structure of a program using JST and decide on things that should be changed or refactored. These changes can then be made in the parse trees before the parse trees are written back out to file. In this way, refactoring tools can be built relatively easily. In order to be able to do this, you need a good understanding of the structure of the parse trees which will be explained in this guide.

1 Java Grammar, XML files and Parse Trees

Both the structure of the XML files containing the parse trees and the in-memory parse trees are closely based on the Java grammar. This grammar specifies exactly what is legal in Java. There are some mistakes in the original grammar provided by Sun so we have modified it to create a working parser.

The grammar specifies what is valid syntax in Java. It is made up of nonterminals and terminals. Nonterminals appear between angle brackets ($\langle i \rangle$) and contain other terminals or nonterminals. Terminals do not have brackets around them and do not contain any other symbols. Terminals in java include keywords such as `int` or `static`, symbols such as `(` or `;` and identifiers.

The Java grammar uses production rules to specify what other symbols nonterminals contains. The nonterminal in question appears on the left-hand side of the production rule,

followed by a ::= sign and the nonterminals and terminals it contains. If there are several alternatives, they are separated by an | symbol. The options for a single nonterminal are terminated by a semicolon to signal that the rules for this nonterminal are finished. And example of a production rule is shown below:

```
<PrimitiveType>
::= <NumericType>
| boolean
;
```

This production rule states that a PrimitiveType contains either a NumericType or the terminal boolean. NumericType in turn can be either a IntegralType or a FloatingPointType and so on.

When a nonterminal contains several terminals and nonterminals, the order in which they are specified in the grammar needs to be adhered to in the code. A production rule for FieldDeclaration can be seen below:

```
<FieldDeclaration>
::= <FieldModifiers>? <Type> <VariableDeclarators> ";"
;
```

This production rule states that when declaring a field we need to specify its modifiers first, followed by its type, its declaration and finally a semicolon. We cannot specify the type before the field modifiers because the order specified in the grammar must be adhered to.

The ? symbol can be used to specify that some parts of the production rule are optional. For example in a field declaration field modifiers are optional and may be left off. However, both the type, variable declaration and semicolon are compulsory and cannot be omitted.

While the Java grammar seems very large at first sight, it is easy to understand if you understand the examples above.

The structure of the XML files and parse trees is based closely on the Java grammar. Each nonterminal is represented in the XML file by an XML element whose type is the name of the nonterminal in the grammar. A terminal is represented by an XML element of type token which contains the actual symbol represented by the terminal.

Let us consider the simple field declaration `private int i;`. From the grammar, we know that this is represented by the nonterminal FieldDeclaration. In the XML file, we can easily find an element of type FieldDeclaration which contains a number of other elements:

```
<FieldDeclaration>
  <FieldModifiers>
    <FieldModifier>
      <token column="2" id="private" line="5">private</token>
      <token column="9" id="WHITESPACE" line="5"> </token>
    </FieldModifier>
  </FieldModifiers>
  <Type>
    <PrimitiveType>
      <NumericType>
        <IntegralType>
```

```

    <token column="10" id="int" line="5">int</token>
    <token column="13" id="WHITESPACE" line="5"> </token>
  </IntegralType>
</NumericType>
</PrimitiveType>
</Type>
<VariableDeclarators>
  <VariableDeclarator>
    <VariableDeclaratorId>
      <token column="14" id="identifier" line="5">i</token>
    </VariableDeclaratorId>
  </VariableDeclarator>
</VariableDeclarators>
<token column="15" id=";" line="5">;</token>
</FieldDeclaration>

```

Clearly, this FieldDeclaration element corresponds to the field that we just created. From the grammar we know that a FieldDeclaration should contain a FieldModifiers child, a Type, a VariableDeclarators child and a semicolon. We can see that in the XML file, the FieldDeclaration element clearly contains these four children. A semicolon is a terminal so it is represented by a token XML element which contains the ; symbol. The other children are all nonterminals. In the grammar, we can easily look up what those nonterminals contain to verify that this XML file is correct.

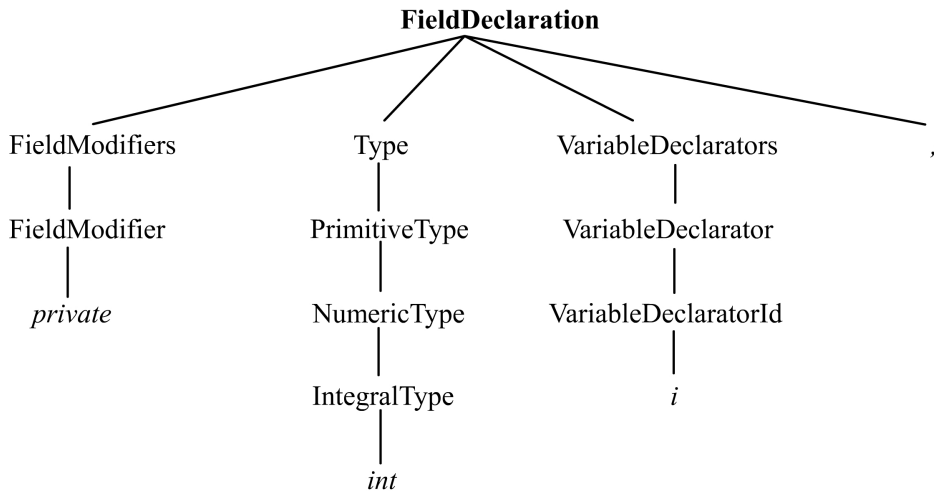


Figure 1: The parse tree of the field declaration.

This XML file can be read in by JST and a parse tree can be constructed in memory. The structure of this parse tree is essentially identical to that of the XML file. A diagram showing the structure of the in-memory parse tree can be seen in Figure 1. From this, it becomes clear that the information in the parse tree is identical to that in the XML file; only the manner this information is represented in is slightly different. Clearly, it is very easy to determine what a program's parse tree will look like by looking at the XML file. We can also easily

derive the parse tree structure by looking at the Java grammar. This is very useful because it means that you can determine exactly what changes you want to make to the parse tree and how you need to make these changes by simply looking at the XML files and the Java grammar.

We will now look more closely at the important classes that you need to know about in order to work with JST parse trees. A UML class diagram showing the most important parse tree classes can be seen in Figure 2. The interface `SyntaxTree` represents the most abstract parse tree concept: any parse tree node including terminals and nonterminals. The `SyntaxTreeBranch` and `SyntaxTreeLeaf` interfaces represent an inner node and a leaf node respectively. An inner parse tree node is a nonterminal; it has at least one child. A leaf node on the other hand represents a terminal because it has no children.

The `Token` class which represents a terminal or token in the parse tree implements the `SyntaxTreeLeaf` interface. When the parse trees are constructed from the XML files, the XML elements of type token are turned into `Token` objects. The type of the token is also recorded, which corresponds to the `id` attribute of the token element in the XML file. `ValueToken` is a subclass of `Token`. It represents a particular kind of token that contains a string value. Some tokens, like tokens of type *semicolon*, do not require a string value to be stored; it is obvious from their type what string they represent. However, tokens like identifiers and whitespace need to store more than just their type and also include their actual string value. In the case of the identifier token in the example above, its type would be *identifier* while its string value would be *i*. Therefore, this token would be turned into a `ValueToken` object.

Nonterminals, or branch nodes, are represented by either a `ParseTreeBranch` or a `ASTList`. These two representations only differ in the way they represent a list of children, with `ParseTreeBranch` using a recursive representation and `ASTList` using a list representation. However, in most cases, you should not have to worry about these different representations. The recursive representation used by `ParseTreeBranch` is used by JST so you do not have to worry about the alternative representation. The next section will look more closely at the recursive representation and the implications for updating the parse trees.

Each part of the JST semantic model of the program, including classes, methods, fields and variables, contains a reference to the part of the parse tree from which it was constructed. This is known as the source of the JST entity and can be extracted using the `getSource()` method. This parse tree part is always a branch rather than a leaf node. For example, a JST `FieldDecl`'s source will point to the `FieldDeclaration` node in the parse tree. From there, it is easy to extract and change the field modifiers, field type and variable declaration. The source of a JST entity is of type `JstParseTreeBranch`, a subclass of `ParseTreeBranch`.

2 Code Generation

Using JST you can analyse a program and then modify it. You can modify the parts of the parse tree corresponding to the parts of program you want to change. After changing the parse trees, you can use them to create `.java` files.

2.1 Modifying the Parse Tree

Once you have decided what part of the program you want to modify, you need to look at the parse tree structure to see what part of the parse tree you need to change and what specific

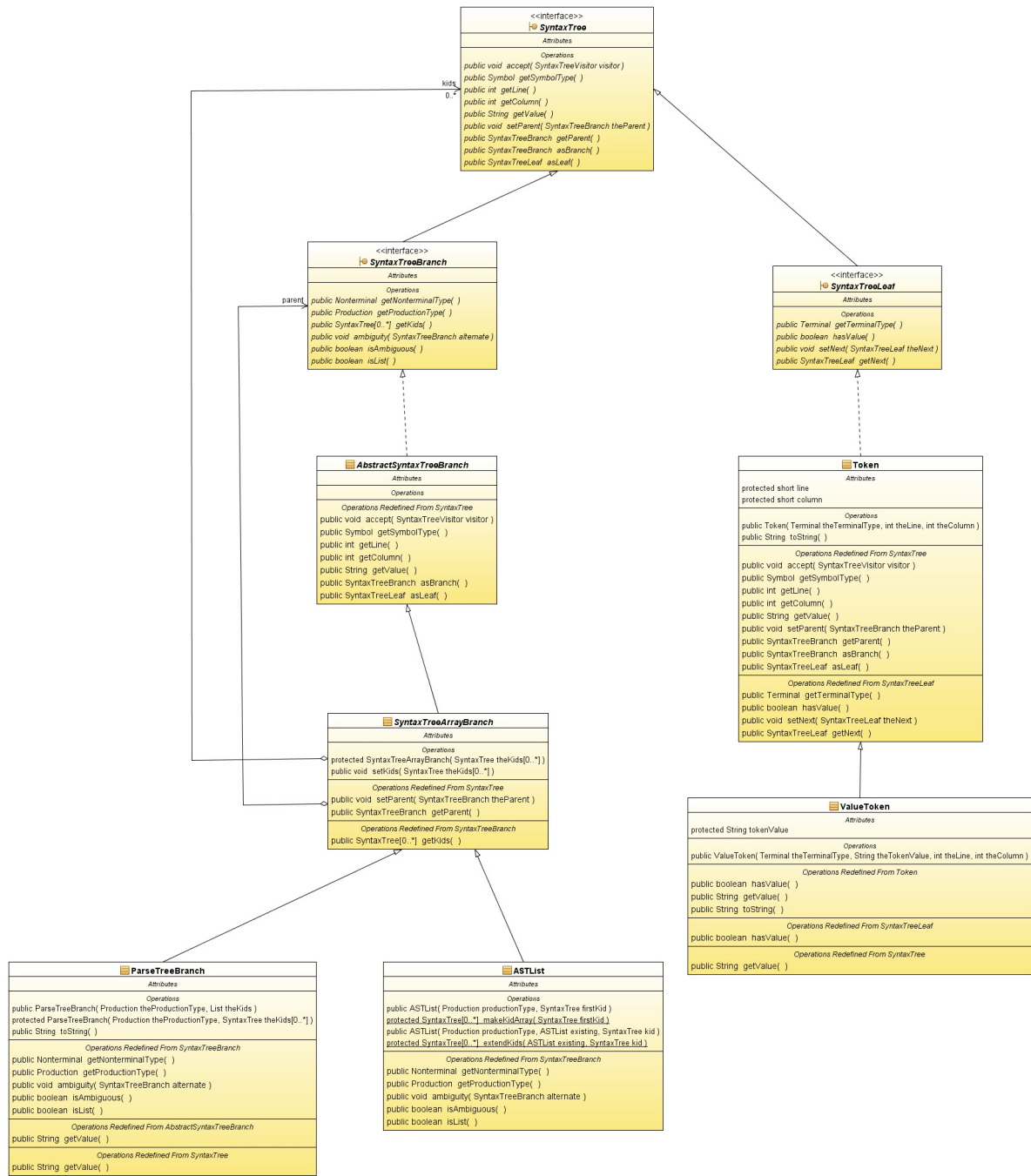


Figure 2: UML class diagram showing the main parse tree classes

changes you need to make. It is very important to check that the changes you make are legal because otherwise the parse trees will become unusable and the generated code may be invalid. The easiest way to check that the changes you are making are legal is to check the Java grammar and remember the correspondence between parse trees and the grammar.

One particular issue to be aware of when modifying parse trees is the recursive way in which some parts are represented. A `FieldDeclaration` contains `FieldModifiers`. Looking at the Java grammar, we can see that the production rule for `FieldModifiers` is:

```
<FieldModifiers>
 ::= <FieldModifier>
 | <FieldModifiers> <FieldModifier>
 ;
```

We can see that a `FieldModifiers` node can either contain a single `FieldModifier`, or another `FieldModifiers` node followed by a `FieldModifier`. For example, Figure 3 shows how four `FieldModifier` nodes are chained together in the parse tree.

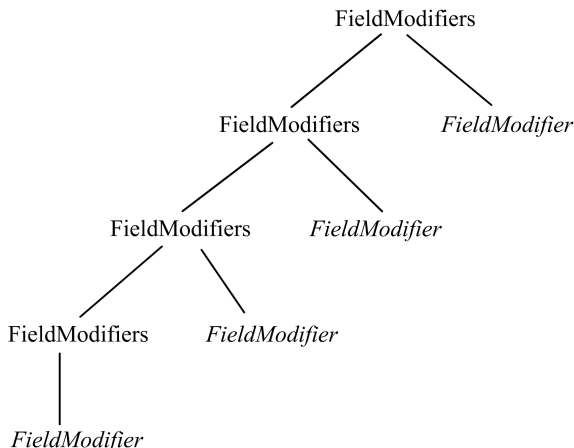


Figure 3: Four `FieldModifier` nodes chained together in a parse tree

When adding a `FieldModifier` node we are therefore required to add another `FieldModifiers` node and insert it in the right position in the parse tree. Similarly, when removing a `FieldModifier` node, we will also have to remove a `FieldModifiers` node. This is easy enough to do as long as you remember the recursive structure of the grammar and the parse tree.

The `SyntaxTree` interface, which is implemented by all nodes in the parse tree, specifies a few very useful methods. The `getParent()` method can be used to move upwards through the tree. This method is implemented by both leaf nodes and inner nodes. Inner nodes also contain a `getKids()` method as defined in the `SyntaxTreeArrayBranch` which you can use to find the child nodes of an inner tree node. This method returns an array of all child nodes, which may be either leaf nodes or inner nodes. It is also very useful when trying to determine the number of children a node has.

In addition to the methods that can be used to navigate the parse tree, there is also a method to determine the type of a node. Both inner and leaf nodes have this method which is called `getSymbolType()`. The type of a node is equal to the type of the XML element it

was constructed from. For example, in the parse tree in Figure 3 the root node has the type *FieldDeclaration*.

2.2 Generating Code

Each parse tree node has a `getValue()` method which returns the concatenated values of all of its children. If you call the `getValue()` method of the root of the parse tree, it will return all tokens in the parse tree concatenated together in their original order. This is an easy way to generate code. After changing the parse tree, you can thus use the `getValue()` method to create .java files.

However, this method has a couple of problems. Firstly, only a single space is added between tokens, meaning that the resulting code will look a little like this:

```
public boolean equals ( Object other ) {
```

which may not be desirable.

The second problem is more serious. The `getValue()` method only includes the tokens that are directly connected to the parse tree. This includes all tokens which are required by the grammar, but excludes both whitespace and comments. Therefore, they will not be reached by the `getValue()` method and thus the final output will contain no newlines, tabs or any other whitespace other than the standard single space inserted automatically between tokens by the `getValue()` method. Comments are also not included and will not appear in the generated code. This is a significant issue since comments are very important in real-world software. If the refactoring tools created using these parse trees will ever need to be used in reality, removing comments is not acceptable.

Fortunately, there is another way to generate code from the parse trees. The whitespace and comment tokens are created even though they are not directly connected to parts of the parse tree. However, they are connected to other tokens. All tokens are chained together using the `next` field in the `Token` class. In this way, the order they were in originally in the Java file is preserved. This means that to generate code, you simply need to find the first token in the parse tree and follow the token chain, printing the contents of each token. The first token can easily be found by continually going down the left branch of the tree until a leaf is reached.

In the current version of the parse trees, comment tokens are not created and added to the token chain. While the whitespace tokens are added to the token chain, they only produce a single space when printed out rather than tabs or new lines as they should. However, these shortcomings are likely to be fixed in future versions of parse trees and therefore writing your code so that it can handle the updated versions is a good idea.

Another issue with using the token chain to generate code is that there may be several repeated tokens in ambiguous sections of the parse tree. For example, whenever you use a class name, it is not clear from the grammar if the entity you are referring to is a class or an interface. Therefore, two versions of the parse tree are created, one representing the class variant and the other representing the interface variant. As a result, the token containing the class name you are using is repeated several times in the token chain. When following the token chain to generate the code, you need to check if you are in an ambiguous section of the parse tree. If you are, you need to take steps to ensure that duplicate tokens are skipped. You can determine if you are in an ambiguous section by looking up the parse tree. If any

G Human Ethics Application for the Encapsulation Surveys

This section contains all the information about the encapsulation survey. It includes the consent and information form given to respondents before the survey as well as the survey used for students (in Java) and professional software developers (in C#). In addition, the original low risk ethics application and the human ethics committee approval for the survey are included.

Janina Voigt, Dr. Warwick Irwin and Dr. Neville Churcher
Department of Computer Science and Software Engineering
University of Canterbury
Email: jvo24@student.canterbury.ac.nz

Date: 27/4/09

Encapsulation in Object Oriented Systems

You are invited to participate in a research project about encapsulation in object oriented systems.

The aim of this experiment is to find out how people practise encapsulation when writing software and what they see as good and bad practice. We are not trying to assess your programming skills or knowledge of computer science in any way.

For this experiment, we will ask you to write a couple of methods for a given scenario containing one or two classes. We will also ask you to rank several different ways of writing a method, from what you think is best to what you think is worst.

The results of the experiment may be published, but you may be assured of the complete confidentiality of data gathered in this investigation: the identity of participants will not be made public. To ensure anonymity and confidentiality, the survey you will be asked to fill in is anonymous.

The project is being carried out as part of an Honours project by Janina Voigt under the supervision of Dr. Warwick Irwin (03 364 2987 ext 8225) and Dr. Neville Churcher (03 364 2987 ext 6352). They will be pleased to discuss any concerns you may have about participation in the project.

The project has been reviewed and approved by the University of Canterbury Human Ethics Committee.

Janina Voigt, Dr. Warwick Irwin and Dr. Neville Churcher
Department of Computer Science and Software Engineering
University of Canterbury
Email: jvo24@student.canterbury.ac.nz

Date: 27/4/09

Consent Form

Encapsulation in Object Oriented Systems

I have read and understood the description of the above-named project. On this basis I agree to participate in the project, and I consent to publication of the results of the project with the understanding that anonymity will be preserved.

I understand also that I may at any time withdraw from the project, including withdrawal of any information I have provided.

I note that the project has been reviewed and approved by the University of Canterbury Human Ethics Committee.

Name (please print): _____

Signature: _____

Date: _____

Encapsulation in Object Oriented Systems

Previous Programming Experience

1) How long ago did you first learn to program? _____

2) How familiar are you with OO? _____

3) When did you first learn OO? _____

4) What programming language(s) have you used? _____

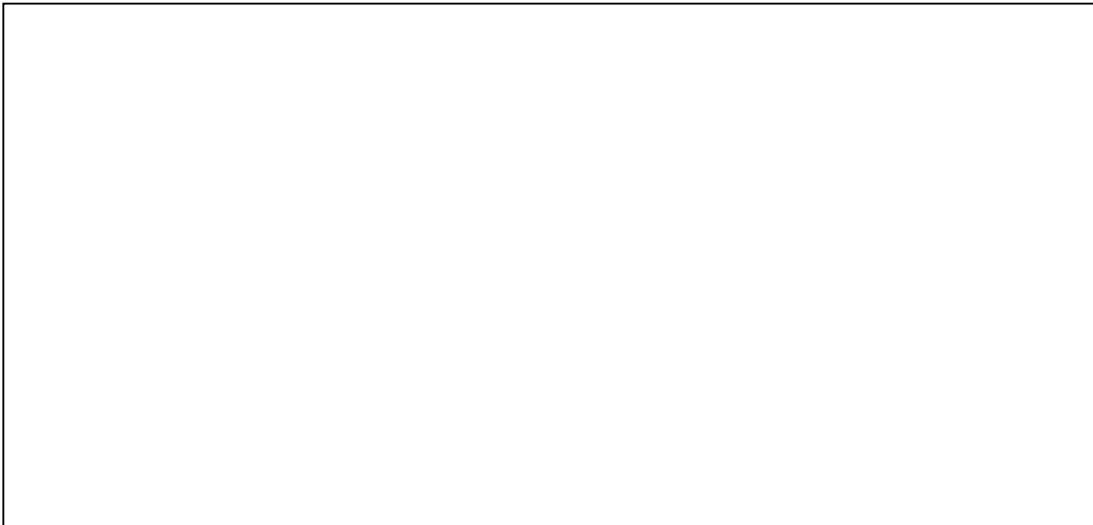
5) What programming language do you feel most proficient in? _____

6) How much experience do you have using C#? _____

Consider the following class Vehicle:

```
class Vehicle {  
    private int weight;  
  
    public int GetWeight() {  
        return weight;  
    }  
  
    public bool IsHeavierThan(Vehicle other) {  
        //Something goes here  
    }  
}
```

7) Write a ToString() method for the Vehicle class that returns "Vehicle weighs *weight* kg", where weight is the Vehicle's weight.



Now we want to complete the code for the `IsHeavierThan(Vehicle other)` method of the `Vehicle` class.

Here are several ways in which we could complete this method:

Option 1

```
public bool IsHeavierThan(Vehicle other) {  
    return this.weight > other.weight;  
}
```

Option 2

```
public bool IsHeavierThan(Vehicle other) {  
    return this.weight > other.GetWeight();  
}
```

Option 3

```
public bool IsHeavierThan(Vehicle other) {  
    return this.GetWeight() > other.GetWeight();  
}
```

8) Which of the above options do you consider the best? Rank the three options, from best to worst.

1 _____

2 _____

3 _____

Give reasons for the way you ranked the options:

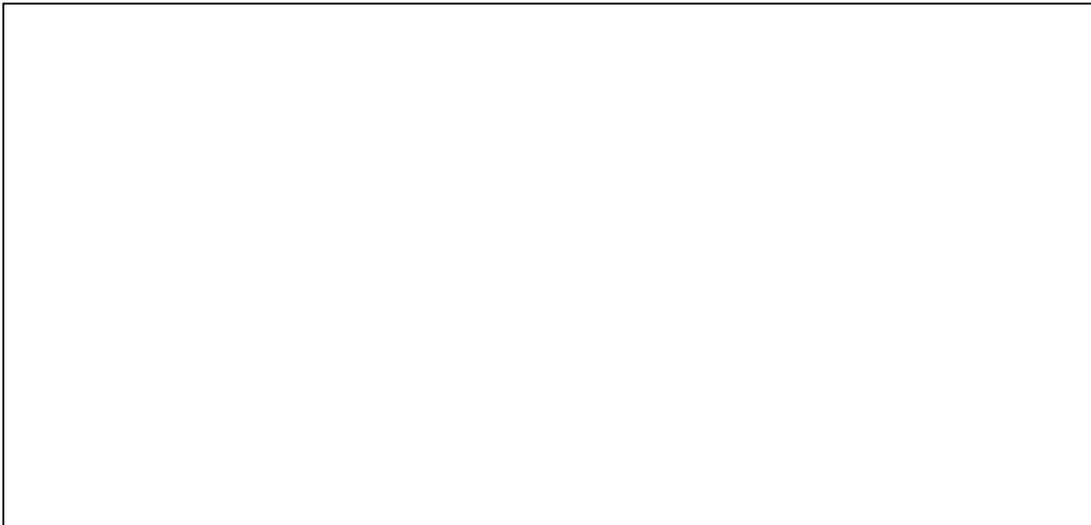
We will now extend our design by adding a second class, `Truck`, which is a subclass of `Vehicle`. The `Truck` class contains a field *payload* storing the maximum load the truck is allowed to carry.

```
class Truck:Vehicle {
    private int payload;

    public int GetPayload() {
        return payload;
    }

    public int GetGrossWeight() {
        //Something goes here
    }
}
```

9) Write a `ToString()` method for the `Truck` class that returns “Truck weighs *weight* kg and can carry up to *payload* kg”, where *weight* is the `Truck`’s weight and *payload* is the maximum load it can carry.



Now we want to complete the code in the `GetGrossWeight()` method of the `Truck` class. This method is supposed to calculate the gross weight of the truck, i.e. the weight of the truck plus the maximum load it can carry. Here are several different ways in which this method could be written:

Option 1

```
public int GetGrossWeight() {  
    return weight + payload;  
}
```

Option 2

```
public int GetGrossWeight() {  
    return GetWeight() + payload;  
}
```

Option 3

```
public int GetGrossWeight() {  
    return GetWeight() + GetPayload();  
}
```

10) Which of the above options do you consider the best? Rank the three options, from best to worst.

1 _____

2 _____

3 _____

Give reasons for the way you ranked the options:

Encapsulation in Object Oriented Systems

Previous Programming Experience

1) What was your first programming language? _____

2) What other programming language(s) have you used (if any) ? _____

3) What programming language do you feel most proficient in? _____

4) How much experience do you have using Java? _____

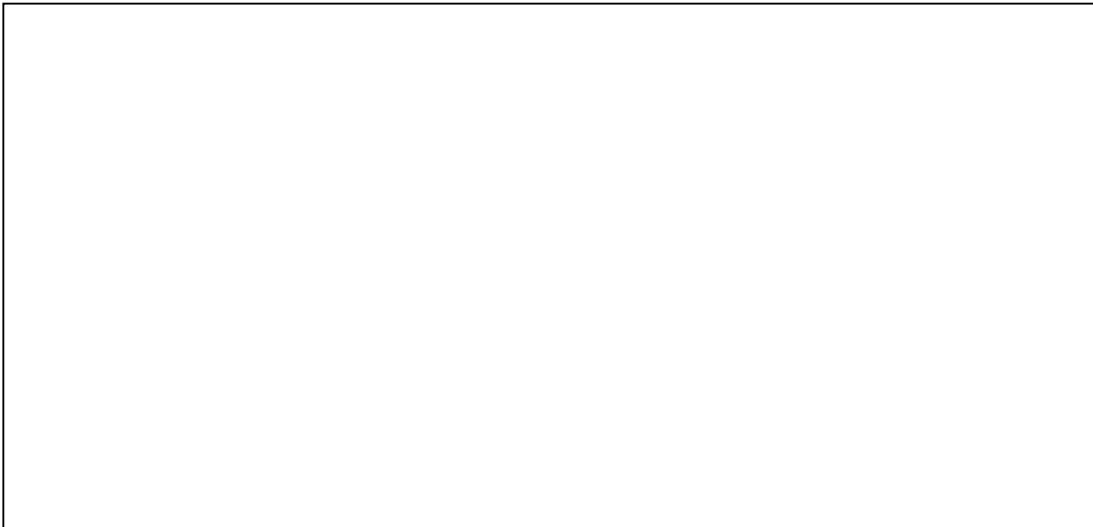
Consider the following class Vehicle:

```
public class Vehicle {
    private int weight;

    public int getWeight() {
        return weight;
    }

    public boolean isHeavierThan(Vehicle other) {
        //Something goes here
    }
}
```

5) Write a `toString()` method for the `Vehicle` class that returns “Vehicle weighs *weight* kg”, where *weight* is the Vehicle’s weight.



Now we want to complete the code for the `isHeavierThan(Vehicle other)` method of the `Vehicle` class.

Here are several ways in which we could complete this method:

Option 1

```
public boolean isHeavierThan(Vehicle other) {  
    return this.weight > other.weight;  
}
```

Option 2

```
public boolean isHeavierThan(Vehicle other) {  
    return this.weight > other.getWeight();  
}
```

Option 3

```
public boolean isHeavierThan(Vehicle other) {  
    return this.getWeight() > other.getWeight();  
}
```

6) Which of the above options do you consider the best? Rank the three options, from best to worst.

1 _____

2 _____

3 _____

Give reasons for the way you ranked the options:

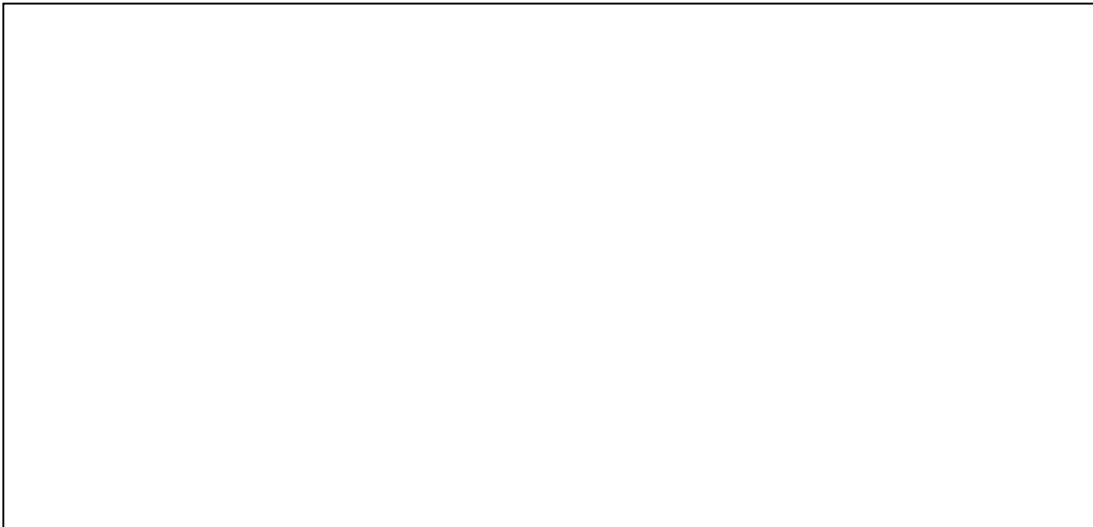
We will now extend our design by adding a second class, `Truck`, which is a subclass of `Vehicle`. The `Truck` class contains a field *payload* storing the maximum load the truck is allowed to carry.

```
public class Truck extends Vehicle {
    private int payload;

    public int getPayload() {
        return payload;
    }

    public int getGrossWeight() {
        //Something goes here
    }
}
```

7) Write a `toString()` method for the `Truck` class that returns “Truck weighs *weight* kg and can carry up to *payload* kg”, where *weight* is the `Truck`’s weight and *payload* is the maximum load it can carry.



Now we want to complete the code in the `getGrossWeight()` method of the `Truck` class. This method is supposed to calculate the gross weight of the truck, i.e. the weight of the truck plus the maximum load it can carry. Here are several different ways in which this method could be written:

Option 1

```
public int getGrossWeight() {  
    return weight + payload;  
}
```

Option 2

```
public int getGrossWeight() {  
    return getWeight() + payload;  
}
```

Option 3

```
public int getGrossWeight() {  
    return getWeight() + getPayload();  
}
```

8) Which of the above options do you consider the best? Rank the three options, from best to worst.

1 _____

2 _____

3 _____

Give reasons for the way you ranked the options:

**UNIVERSITY OF CANTERBURY
LOW RISK APPLICATION FORM**

(For research proposals which are not considered in full by the University Human Ethics Committee)

FOR STUDENT RESEARCH UP TO AND INCLUDING MASTERS LEVEL

**ETHICAL APPROVAL OF LOW RISK RESEARCH INVOLVING HUMAN PARTICIPANTS REVIEWED
BY DEPARTMENTS**

PLEASE read the important notes appended to this form before completing the sections below

- 1 RESEARCHER'S NAME:** Janina Voigt
- 2 NAME OF DEPARTMENT OR SCHOOL:** Department of Computer Science and Software Engineering
- 3 EMAIL ADDRESS:** jvo24@student.canterbury.ac.nz
- 4 TITLE OF PROJECT:** Encapsulation in Object Oriented Systems
- 5 PROJECTED START DATE OF PROJECT:** 1/5/09
- 6 STAFF MEMBER/SUPERVISOR RESPONSIBLE FOR PROJECT:** Dr. Warwick Irwin and Dr. Neville Churcher
- 7 NAMES OF OTHER PARTICIPATING STAFF AND STUDENTS:** None
- 8 STATUS OF RESEARCH:** (e.g. class project, thesis) Honours Project
- 9 BRIEF DESCRIPTION OF THE PROJECT:**
Please give a brief summary (approx. 300 words) of the nature of the proposal in lay language, including the aims/objectives/hypotheses of the project, rationale, participant description, and procedures/methods of the project:-

This project aims to identify how software engineers practise encapsulation. Encapsulation refers to the practice of hiding information and data within components of the system. It is an important tool used by software developers to manage the complexity of software and to avoid errors and mistakes in code.

We are planning to conduct a short survey to find out how software engineering students think about and use encapsulation when programming. In the survey, the students will be asked to write a small amount of code. They will also be asked to rank a number of different pieces of code, from best to worst. It is hoped this will give us an indication of how they think about encapsulation and how they use it.

The survey will be given to second and third year software engineering students during their lab or tutorial times.

10 WHY IS THIS A LOW RISK APPLICATION?

Description should include issues raised in the Low Risk Checklist

Please give details of any ethical issues which were identified during the consideration of the proposal and the way in which these issues were dealt with or resolved. :-

This research is conducted as part of an Honours project. It is low risk because it does not involve threat, deception, invasion of privacy, mental, physical or cultural risk or stress, and does not involve gathering personal information of sensitive nature about or from individuals.

The students who will take part in the experiment will not be asked to perform any tasks that could be harmful to them and no personal information will be gathered about them.

The experiment will be performed as a part of a lab or tutorial session. The survey should take up to 15 minutes to complete and will thus not interfere significantly with their normal class time.

Participation in the experiment is completely voluntary and no incentives or inducements will be given. The project and its goals will be explained at the start of the session to give students the opportunity to give informed consent. Students will also be given the opportunity to pull out of the experiment at any time during or after the session. This is done to ensure that students do not feel stressed about their performance. There is no deception involved in this project since students will be told up-front about the experiment and its goals.

Please ensure that Section A, B and C below are all completed

APPLICANT'S SIGNATURE:

Date

A SUPERVISOR DECLARATION:

- 1 I have made the applicant fully aware of the need for and the requirement of seeking HEC approval for research involving human participants.
- 2 I have ensured the applicant is conversant with the procedures involved in making such an application.
- 3 In addition to this form the applicant has individually filled in the full application form which has been reviewed by me.

SIGNED (Supervisor):

Date

B SUPPORTED BY THE DEPARTMENTAL/SCHOOL RESEARCH COMMITTEE:

Name **Signature:** **Date**

C APPROVED BY HEAD OF DEPARTMENT/SCHOOL:

Name **Signature:** **Date**

ACTION TAKEN BY HUMAN ETHICS COMMITTEE:

- Added to Low Risk Reporting Database Referred to University of Canterbury HEC
- Referred to another Ethics Committee - Please specify:

.....

REVIEWED BY:..... **Date**

Please attach copies of any Information Sheet and/or Consent Form

Forward two copies to:

**The Secretary
Human Ethics Committee
Level 6, Registry Building**

**All queries will be forwarded to the applicant within 7 days
Please include a copy of this form as an appendix in your thesis or course work**

NOTES CONCERNING LOW RISK REPORTING SHEETS

1. This form should **only be used** for proposals which are **Low Risk** as defined in the University of Canterbury Human Ethics Committee Principles and Guidelines policy document, and which may therefore be properly considered and approved at departmental level under Section 5 of that document;
2. Low Risk applications are:
 - a Masters theses where the projects do not raise any issue of deception, threat, invasion of privacy, mental, physical or cultural risk or stress, and do not involve gathering personal information of a sensitive nature about or from individuals.
 - b Masters level supervised projects undertaken as part of specific course requirements where the projects do not raise any issue of deception, threat, invasion of privacy, mental, physical or cultural risk or stress, and do not involve gathering personal information of sensitive nature about or from individuals.
 - c Undergraduate and Honours class research projects which do not raise any issue of deception, threat, invasion of privacy, mental, physical or cultural risk or stress, and do not involve gathering personal information of sensitive nature about or from individuals, but do not have blanket approval as specified in Section 4 of the Principles and Guidelines.
3. No research can be counted as low risk if it involves:
 - (i) invasive physical procedures or potential for physical harm
 - (ii) procedures which might cause mental/emotional stress or distress, moral or cultural offence
 - (iii) personal or sensitive issues
 - (iv) vulnerable groups
 - (v) Tangata Whenua
 - (vi) cross cultural research
 - (vii) investigation of illegal behaviour(s)
 - (viii) invasion of privacy
 - (ix) collection of information that might be disadvantageous to the participant
 - (x) use of information already collected that is not in the public arena which might be disadvantageous to the participant
 - (xi) use of information already collected which was collected under agreement of confidentiality
 - (xii) participants who are unable to give informed consent
 - (xiii) conflict of interest e.g. the researcher is also the lecturer, teacher, treatment-provider, colleague or employer of the research participants, or there is any other power relationship between the researcher and the research participants.
 - (xiv) deception
 - (xv) audio or visual recording without consent
 - (xvi) withholding benefits from "control" groups
 - (xvii) inducements
 - (xviii) risks to the researcher

This list is not definitive but is intended to sensitise the researcher to the types of issues to be considered. Low risk research would involve the same risk as might be encountered in normal daily life.

4. Responsibility

Supervisors are responsible for:

- (i) Theses where the projects do not raise any issues listed below.
- (ii) Masters level supervised projects undertaken as part of specific course requirements where the projects do not raise any issue.
- (iii) Undergraduate and Honours class research projects which do not raise any issue listed but do not have blanket approval as specified in the Principles and Guidelines.

HODs are responsible for:

- (i) Giving final approval for the low risk application.
- (ii) Ensuring a copy of all applications are kept on file in the Department/School.

Ref: HEC 2009/LR/17

11 June 2009

Janina Voigt
Department of Computer Science & Software Engineering
UNIVERSITY OF CANTERBURY

Dear Janina

Thank you for forwarding to the Human Ethics Committee a copy of the low risk application you have recently made for your research proposal "Encapsulation in object oriented systems".

I am pleased to advise that this application has been reviewed and I confirm support of the Department's approval for this project.

With best wishes for your project.

Yours sincerely

Dr Michael Grimshaw
Chair, Human Ethics Committee