CLASS ENCAPSULATION AND OBJECT ENCAPSULATION
An Empirical Study

Janina Voigt, Warwick Irwin, Neville Churcher
Department of Computer Science and Software Engineering, University of Canterbury, Christchurch, New Zealand
jvo24@student.canterbury.ac.nz, warwick.irwin@canterbury.ac.nz, neville.churcher@canterbury.ac.nz

Keywords: Encapsulation; encapsulation boundary; OO design; information hiding.

Abstract: Two schools of thought underpin the way OO programming languages support encapsulation. Object encapsulation ensures that private members are accessible only within a single object. Class encapsulation allows private members to be accessed by other objects of the same class. This paper describes an empirical investigation into the way encapsulation is used in practice in class encapsulation languages C# and Java. We find arbitrary and inconsistent programming practices and suggest that object encapsulation is more intuitive and provides OO design advantages.

1 INTRODUCTION

'Programming is about managing complexity' (Eckel, 1998)[p6]. Complexity can be controlled by decomposing software into pieces with high *cohesion* and low *coupling* (Yourdon, 1979). The *information hiding* principle advocates encapsulation of implementation decisions so that they can be changed without impacting the rest of the program (Parnas, 1972). Only relatively stable features of a program should remain externally visible. The *Stable Abstractions Principle* reinforces this aspect of information hiding (Martin, 1997), and the fundamental Computer Science concept of *Abstract Data Types* applies the principle to data structures.

Encapsulation is the core programming language mechanism that supports these design principles, making it perhaps the most important semantic characteristic of programming languages. Because it is so fundamental, we might expect a clear consensus on how programming languages should support encapsulation, and on how encapsulation mechanisms should be employed. Among OO programmers, however, we have found a surprising lack of accord over even the most basic of encapsulation questions (Voigt, 2009). 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?

In languages such as Java, novices are commonly advised to declare attributes `private`, and to write *getters* and *setters*, which may be `public`. C# goes further and supports *property* syntax (Hejlsberg, 2008). This mechanical exposing of attributes through accessor methods undermines information hiding. A definitive characteristic of OO is the closeness of methods to the data on which those methods operate. The use of accessors is a sign that data is being manipulated outside its owner object. The *Tell, Don't Ask* principle (Hunt, 1998) 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. The *Law of Demeter* (Lieberherr, 1989) effectively prohibits the use of getters; an object may use only its own data, local variables and parameters.

Encapsulation is considerably more complex in the presence of inheritance. The use of `protected` access is controversial. Riel, as one of his 'golden rules for OO design', advises designers to 'avoid `protected` data', and instead make it `private` to its class (Riel, 1996). Holub states that '`protected` data is an abomination' (Holub, 2003). However, other OO cultures encourage or enforce the opposite rule. In Objective C the default access to attributes is `protected`. In Smalltalk inherited properties can't be hidden; all data is implicitly

`protected`. But Smalltalk programmers do not describe encapsulation in these terms. On the contrary, they describe data as private, but to an object, not a class.

2 OBJECT ENCAPSULATION AND CLASS ENCAPSULATION

Rogers states that encapsulation means only grouping of properties and that hiding is an orthogonal concept (Rogers, 2001). In this paper we use its conventional meaning of both grouping and hiding. A suitable definition is (Booch, 2007):

> The process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation.

In non-OO systems, the encapsulation boundary is normally around modules. OO languages variously place the boundary around objects or classes.

Smalltalk, arguably the archetypal OO language, uses object encapsulation, as Goldberg and Robson explain (Goldberg, 1989):

> 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.

Unlike Smalltalk, C++ augmented an existing language which used an encapsulation approach based on static modules, and so it is perhaps unsurprising that it (and subsequently Java and C#) use class encapsulation: 'Note that in C++, the class—not the individual object—is the unit of encapsulation' (Stroustrup, 1997)[p754]. Objects of the same class can access each other's `private` properties, as shown in the following Java code, adapted from Stroustrup's example:

```
public class Node {
  private int data;
  private Node next;
  public boolean find(int d) {
    for(Node n=this; n!=null; n=n.next)
      if(n.data==d) return true;
    return false;
  }
}
```

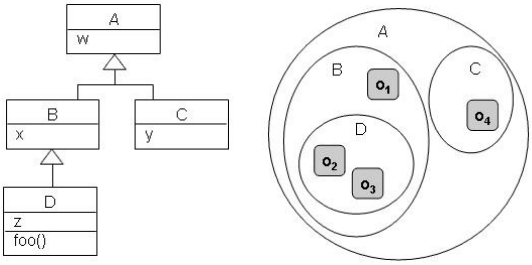
When inheritance is used, the class encapsulation boundary bisects objects, so that one part of an object cannot access other (inherited) parts of the same object. `Protected` access approximates object encapsulation within an object, but still allows objects of the same class to access each other's internals. Although Stroustrup introduced `protected` access (he credits Mark Linton as co-inventor), he is unconvinced about its merits (Stroustrup, 1994):

> 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 and object encapsulation has important consequences for software design, yet as far as we can tell has largely escaped the attention of software practitioners. Stroustrup, makes little of it. Snyder86 (Snyder, 1986) acknowledges the difference, but says 'we ignore this distinction in this paper as it does not affect our analysis'. We suggest that the validity of Snyder's argument—that inheritance is antithetical to encapsulation—hinges on which encapsulation boundary is assumed.

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



(a) An example class diagram (b) Classes and some instances

Figure 1: An encapsulation example.

main reasons he claims that C++ and Java 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 at it is there to keep certain things in.
>
> […]
>
> 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.

Object autonomy is indeed important. When class encapsulation is used to allow one object access to another, the accessed object cannot make use of inheritance to act in its own way. This prevents use of the open-closed principle (Meyer, 1988), dependency injection (Fowler, 2004) and reduces opportunities for software reuse. These are among the most influential OO design principles.

Figure 1a shows an example UML class diagram. The classes A to D represent the units of class encapsulation. Figure 1b shows the same classes, and a number of instances, o_1 to o_4, which represent the units of object encapsulation. Figure 2 contrasts the two encapsulation boundaries for the same example. Class encapsulation boundaries cut through objects, and also encompass portions of multiple objects.

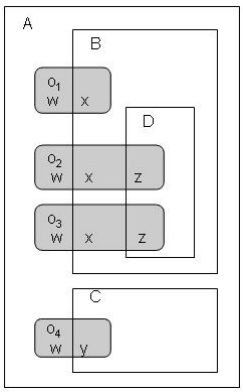


Figure 2: Different encapsulation boundaries.

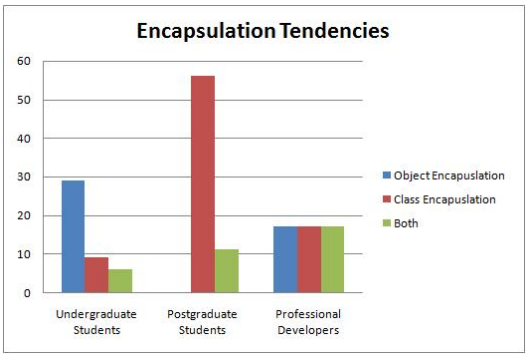


Figure 3: Survey results.

3 A PRELIMINARY SURVEY

In a recent paper (Voigt, 2009) we reported the results of a survey of the encapsulation preferences of programmers who use Java and/or C#. 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 embraced the class encapsulation offered by the programming languages. Professionals showed diverse preferences. All three populations exhibited a degree of confusion over encapsulation boundaries.

The survey 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. Coad and Yourdon (Coad, 1991) quote the 1986 Encyclopaedia Brittanica entry on Classification Theory (Brittanica, 1986):

> 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 is what makes object encapsulation intuitive; it echoes the boundaries between real-world objects.

4 MEASURING ENCAPSULATION PRACTICES IN JAVA

We wrote a static analysis tool to measure encapsulation in Java programs, to determine what developers do in practice, as opposed to what they say they do. We addressed 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 an overview of how rigorously data is hidden from the outside world.

Characterising actual accesses to attributes is more complex. Our program accumulates the number of accesses to `public`, package, `protected` and `private` attributes 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 impossible to determine precisely whether a reference refers to the same object as the one doing the accessing. We employ a simple heuristic. 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. Otherwise, we check the syntax of the access code. If it is of the form *fieldName* (without a qualifier), *this.fieldName* or *super.fieldName*, the access comes from within the same object. If the access is of the form *qualifier.fieldName*, we presume the access comes from outside the object.

Our tool analyses the data it collects to determine the degree to which the two types of encapsulation are used in each program.

We implemented our tool using Java Symbol Table (JST) (Irwin, 2007, Irwin, 2003, Irwin, 2005) to model the semantic structure of a program, including 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 current version of the Qualitas Code Corpus produced by the University of Auckland as a repository of real-world software to be analysed (Qualitas_Research_Group, 2009). This version of the corpus contains 100 Java projects of diverse provenance, including some very well-known programs such as Eclipse and ANTLR.

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 (which are compatible with Java 1.6). A list of the 33 programs analysed can be found in (Voigt, 2010). We also analysed an additional 11 programs produced by groups of 6-7 second-year software engineering students as part of a semester-long project for real clients.

Our experiment has similarities to a recent empirical study that measured how often different access modifiers are used and how frequently fields

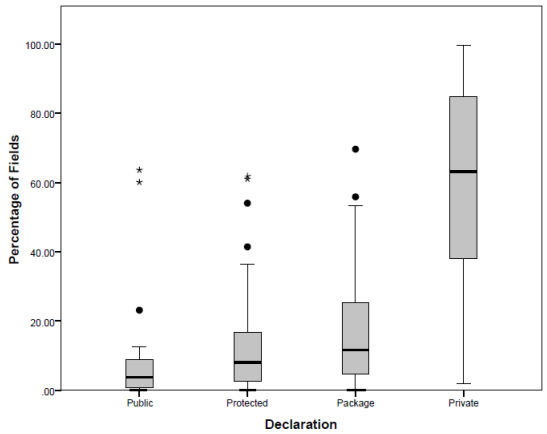


Figure 4: Use of protection levels in corpus programs.

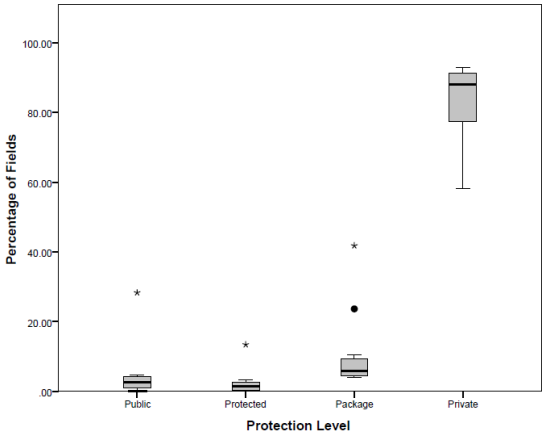


Figure 5: Use of protection levels in student programs.

are accessed in the Qualitas Code Corpus (Tempero, 2009). Tempero analysed all programs in the corpus, but considered only the level of exposure of fields, and implicitly assumed class encapsulation.

5 RESULTS AND ANALYSIS

The corpus programs contained 69-2159 attributes, and 355–10818 accesses. The student programs contained 55–469 attributes, and 208–973 accesses. Figure 4 and Figure 5 show the relative numbers of different protection levels used in corpus and students programs.

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 tend to be more tightly encapsulated.

It is interesting to note that students rarely declared `protected` attributes and that corpus programs tended to access `protected` attributes somewhat more frequently than other types.

The graphs also show highly diverse encapsulation practices in the corpus programs, particularly for `private` data which spans a range from virtually no use to almost exclusive use.

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

- Unsurprisingly, `public` attributes were used heavily from outside their declaring class (corpus 57.5%, student 40.8%), and were also used heavily internally.
- In Java, `protected` gives access to subclasses and classes in the same package. In the corpus subclass access was much more common (27.9%) than same-package access (5.6%). The remaining accesses were from within the declaring class. However in student programs, out of the eight that used `protected`, two used it as package access, four as `private` access, and two as subclass access. This suggests considerable 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%). `Package` is the default protection level, so is used when developers forget to specify tighter access. Six out of 11 student programs 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. Almost all corpus and student programs made some use of this, but the average percentage of accesses to `private` attributes from other objects was very low (3.0% and 1.3% respectively).

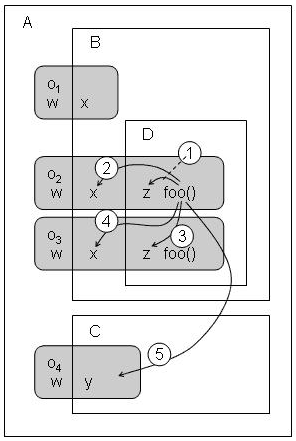


Figure 6: An overview of access categories.

Figure 6 shows the main categories of access we measured. Table 1 shows 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. This category does not reveal anything about encapsulation boundary preferences as it crosses no encapsulation boundaries. Categories 4 and 5 similarly do not indicate any encapsulation boundary preference as the accesses 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 might be explained by the fact that the scenario in the survey was 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.

Table 1: Percentage of accesses by category.

	Category of Access	Corpus programs	Student programs
1	Same object, same class	82.6 %	93.7 %
2	Same object, superclass	6.6 %	0.3 %
3	Different object, same class	2.7 %	1.5 %
4	Different object, superclass	0.2 %	0.0 %
5	Different object, different class	7.8 %	4.5 %

Figure 7 andFigure 8 show the relative use of object and class encapsulation in each of the programs analysed. Clearly, object encapsulation is preferred by the majority of corpus programs. Notably, there is only a single program that clearly uses class encapsulation (JFreeChart) but a number of programs use mostly object encapsulation or a mixture of the two encapsulation approaches. The majority of student programs use class encapsulation rather than object encapsulation. This is consistent with the observation that students generally avoided the `protected` access level, which can be used to support object encapsulation in languages like Java.

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 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: FitJava. This indicates not only that the encapsulation is very loose and 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.

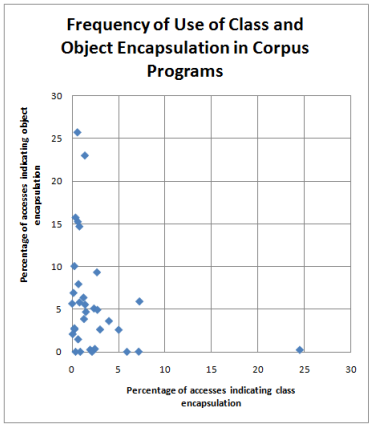


Figure 7: Corpus use of object and class encapsulation.

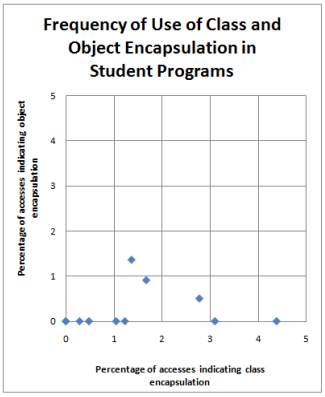


Figure 8: Student use of object and class encapsulation.

6 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 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, approximately 30% of accesses to `protected` attributes came from subclasses.

The dominant 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 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, it is restrictive 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, which supports neither object nor class encapsulation. It appears that in many cases omission of the access modifier is 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 causes confusion among students.

Our ongoing work includes investigating the use of protection levels on methods in Java to broaden our understanding of encapsulation practices beyond

attributes. We are also extending our dataset of programs to include all 100 programs in the Qualitas Code Corpus.

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.

7 REFERENCES

Booch, G., Maksimchuk, R., Engle, M., Young, B., Conallen, J., Houston, K. 2007. *Object Oriented Design with Applications*, Addison Wesley Professional.

Brittanica, E. 1986. *Encyclopaedia Brittanica*, Chicago, IL, USA, Encyclopaedia Brittanica.

Coad, P., Yourdon, E. 1991. *Object Oriented Design*, Upper Saddle River, NJ, USA, Yourdon Press.

Eckel, B., Sysop, Z. F. 1998. *Thinking in Java*, Upper Saddle River, NJ, USA, Prentice Hall PTR.

Fowler, M. 2004. *Inversion of Control Containers and the Dependency Injection Pattern* [Online]. [Accessed Jan 2010].

Goldberg, A., Robson, D. 1989. *Smalltalk-80: The Language*, Boston, MA, USA, Addison Wesley Longman.

Hejlsberg, A., Torgersen, M., Wiltamuth, S., Golde, P. 2008. *The C# Programming Language*, Addison Wesley Professional.

Holub, A. 2003. Why Extends Is Evil. *Java World.*

Hunt, A., Thomas, D. 1998. *Tell, Don't Ask* [Online]. Available: http://www.pragprog.com/articles/tell-dont-ask [Accessed].

Irwin, W. 2007. *Understanding and Improving Object-Oriented Software through Static Software Analysis.* Ph.D., University of Canterbury.

Irwin, W., Churcher, N. I. 2003. Object Oriented Metrics: Precision Tools and Configurable Visualisations. *In:* METRICS2003: 9th IEEE Symposium on Software Metrics, Sep 2003 Sydney, Australia. 112-123.

Irwin, W., Cook, C., Churcher, N. I. 2005. Parsing and Semantic Modelling for Software Engineering Applications. *In:* Strooper, P., ed. Australian Software Engineering Conference, Mar 2005 Brisbane, Australia. IEEE Press, 180-189.

Lieberherr, K., Holland, I. 1989. Assuring Good Style for Object-Oriented Programs. *IEEE Software,* 6**,** 38-48.

Martin, R. C. 1997. Stability. *C++ Report.*

Meyer, B. 1988. *Object-Oriented Software Construction*, New York, Prentice-Hall.

Parnas, D. L. 1972. On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM,* 15**,** 1053 - 1058.

Qualitas_Research_Group. 2009. *Qualitas Corpus Version 20090202* [Online]. Available: http://www.cs.auckland.ac.nz/~ewan/corpus [Accessed 2009].

Riel, A. J. 1996. *Object-Oriented Design Heuristics*, Reading, Mass., Addison-Wesley.

Rogers, P. 2001. Encapsulation Is Not Information Hiding. *Java World.* http://www.javaworld/javaworld/jw-05-2001/jw-0518-encapsulation.html

Snyder, A. Year. Encapsulation and Inheritance in Object-Oriented Programming Languages. *In:* Object-oriented programming systems, languages and applications, 1986. 38-45.

Stroustrup, B. 1994. *The Design and Evolution of C++*, ACM Press/Addison-Wesley Publishing Co.

Stroustrup, B. 1997. *The C++ Programming Language*, Boston, MA, USA, Addison Wesley Longman.

Tempero, E. D. Year. How Fields Are Used in Java: An Empirical Study. *In:* Australian Software Engineering Conference, Apr 2009 2009 Gold Coast, Australia. IEEE Computer Society, 91-100.

Voigt, J., Irwin, W., Churcher, N. I. 2009. Intuitiveness of Class and Object Encapsulation. *6th International Conference on Information Technology and Applications.* Hanoi, Vietnam.

Voigt, J., Irwin, W., Churcher, N. I. 2010. *Technical Report Tr-Cosc 01/10: List of Qualitas Code Corpus Programs Used for Encapsulation Research* [Online]. Christchurch, New Zealand: University of Canterbury. Available: http://www.cosc.canterbury.ac.nz/research/reports/TechReps/2010/tr_1001.pdf [Accessed].

Yourdon, E., Constantine, L. 1979. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Englewood Cliffs, N.J., Prentice Hall.