# Empirical Observations of Class-level Encapsulation and Inheritance

Tim Menzies * Philip Haynes †

January 14, 1996

## Abstract

OO design theory promises numerous benefits from the use of inheritance and the information hiding properties of encapsulated classes. Such promises are commonly used to justify the switch to the OO paradigm from (e.g.) functional decomposition languages like C. In this paper, we audit these claims via an analysis of 2000 Smalltalk classes in 5 applications. We will find that, on average, applications use low levels of inheritance and information hiding at the class and class hierarchy level.
*KEYWORDS*: Smalltalk, encapsulation, inheritance, empirical studies.
*CATEGORY*: Research.
*TOPIC AREA*: Object testing and metrics.

## 1 Introduction

This paper argues that object-oriented applications do not make extensive use of inheritance, encapsulation and information hiding at the class and class hierarchy level. This is a surprising observation. OO design theorists (e.g. [3, 18]) promises numerous benefits from the use of inheritance and the information hiding properties of encapsulated classes (e.g. reduced error rates [12]). Such promises are commonly used to justify the switch to the OO paradigm from (e.g.) functional decomposition languages like C.

There are several consequences of this paper. OO designers seek to divide up applications into modules in order to sub-divide the tasks of design, coding and testing. Our results suggest that classes and class hierarchies are not the best units of division. In Section 7 we speculate that use cases or OO design patterns may be a better unit of division. More generally, we suggest that how we *actually* use OO languages is different to how OO design theorists tell us we *should* use OO languages. We echo a call by Fenton *et. al.* [4], for more empirical studies to produce an accurate picture of the practice of OO programming.

We will argue as follows. Section 2 describes in approximate terms how we will measure information hiding using *call graphs*. Section 3 describes certain low-level details of call graphs. This will lead to a more precise assessment criteria (see Section 3.5). Section 4 describes a simple parser that can implement our assessment criteria. This parser is used in section 4.1 to analyses 194,451 method calls between 1,643 Smalltalk classes from 5 applications. We find that, on average, classes make as much use of services defined in other classes as they do in their own hierarchy. In Section 5 we find that average number of message sends per class is rather small. The picture that we will paint is of lots of small classes talking as much to their distant neighbors as to themselves. As a side-effect of the information hiding study, we also have evidence relating to the relative use of inheritance. Section 6 reports that a surprisingly low-level of inheritance was seen in the studied systems. See Section 7 for a discussion of related work.

## 2 Assessing Information Hiding and Inheritance

Traditional object-oriented (OO) design theory states that an OO program is a set of classes in a hierarchy. Sub-classes refine the services inherited from their parent classes. In the standard view, which we do not agree with, each class is an independent encapsulated entity which hides its internal details from external clients. Application developers, it is claimed, can ignore the internal structure of a class. That is, classes are *encapsulated* independent computational units which hide

*Dept. of Software Development, Monash University, Caulfield, Melbourne, VIC., Australia, 3185; +61-3-9903-1022; timm@insect.sd.monash.edu.au
†OO Pty. Ltd., P.O. Box 1826, Nth. Sydney, NSW, +61-2-957-1092; p.haynes@oose.com.au

their internal structure (termed *information hiding*).

According to the standard view, there are many potential benefits of inheritance and encapsulation. Inheritance can be used to simplify the construction of new class. Encapsulation could encourage modular design. Well-design modules could be re-usable in multiple applications. Encapsulated modules could be understood and modified in isolation to the rest of the system, thus simplifying software maintenance. Errors in a well-encapsulated system should not propagate outside of encapsulation boundaries.

We will assess these claims using *calls graphs*. A *call graph* is a directed graph whose vertices represent basic data values and whose edges represent how those basic data values are passed to subroutines. For OO systems, a call graph vertex is a class. Each method of a class contains calls to methods defined in some other class. A call graph edge is added for each such call and reference.

If class encapsulation really promotes information hiding, then we should see some evidence of this in call graphs. More precisely, we should see relatively dense call graphs within classes or class hierarchy boundaries and relatively sparse call graphs between classes and class hierarchies. Further, once these call graphs are generated, they can also be used to assess levels of inheritance use.

We operationalise our test for information hiding in Section 3.5 after reviewing the low-level details of call graphs.

# 3 Call Graphs: The Details

For a detailed picture of the patterns of calls in an OO system we can look at call-graph between all methods $M_j^i$ defined in a class $C^i$ which call some other class $X$. Consider Figure 1. The classes {OBJECT, SEQUENCE, LIST, STACKMACHINE, DEBUGGER} call the methods {`do:`, `reject:`, `do:`, `debug`, `executeMachineCode`}.

These calls can categorised as either message sends or message receives (see Sections 3.1 & 3.2). Apart from message sends and receives, classes can also access their own instance variables $I_v$, inherited instance variables $A_v$, or external variables in other hierarchies $E_v$. In some sense, such references are "calls" to a class. However, we argue below in Section 3.4 that for the purposes of assessing information hiding, the $I_v$, $E_v$, $A_v$ references can be ignored.

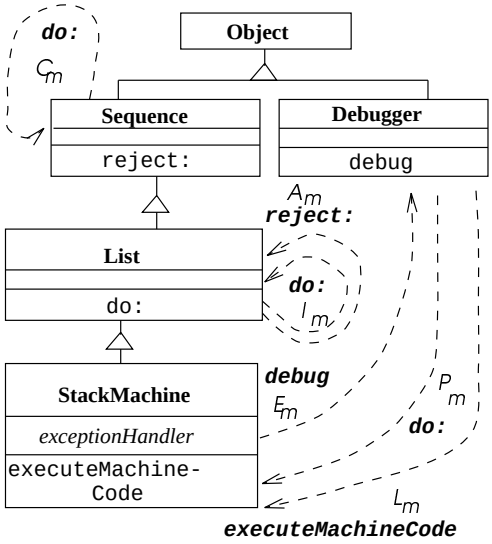


Figure 1: Message sends

## 3.1 Four Types of Message Sends

$I_m$ (*internal method call*): A method uses a method defined in its own class. For example, in Figure 1, a method in class LIST is sending the message `do:` to itself.

$A_m$ (*ancestor method call*): A method uses a method defined in one of its super-classes. For example, in Figure 1, a method in class LIST is sending the message `reject:` to itself. However, since this method is not defined in LIST, the message send ascends the hierarchy.

$E_m$ (*external method call*): i.e. a method uses a method defined in another hierarchy. For example, in Figure 1, a method in class STACKMACHINE is sending the message `debug:` to it's instance variable *exceptionHandler* which is an instance of the class DEBUGGER. This message therefore travels to another hierarchy. Note that in languages where all classes have a common root (e.g. the OBJECT class in Smalltalk), hierarchies are defined to start one level down from the common root. Figure 1 therefore shows two hierarchies: one of SEQUENCE and one for DEBUGGER.

$C_m$ (*child method calls*): A method calls a method defined in one of its sub-classes. For example, in Figure 1, suppose the SEQUENCE `reject:` method recursively iterates over all its contained variables using the `do:` method. In that case, SEQUENCE could be sending `do:` to an instance of its own subclass.

## 3.2 Two Types of Message Receives

$L_m$: A message is received that has a method name which is defined Locally in this class; i.e. the message is handled in this class. For example, in Figure 1, when DEBUGGER sends the message `executeMachineCode:` to STACKMACHINE, this message is caught and processed by the `executeMachineCode:` method defined in STACKMACHINE.

$P_m$: A message is received that has a method name which is not defined locally. Such messages are searched for in the Parent of this class. For example, in Figure 1, STACKMACHINE inherits `do:` from LIST. When DEBUGGER sends the message `do:` to STACKMACHINE, this message ascends the hierarchy.

## 3.3 Testing Information Hiding with Call Graphs

We would infer that information hiding is likely in applications with:

$$(E_m + E_v) < (I_m + I_v + A_m + A_v + C_m) \quad (1)$$

It could be argued that $P_m$ and $L_m$ should feature in Equation 1. We will argue below that this is not necessary (see Section 3.4.1).

## 3.4 Simplifications

It is a non-trivial problem to correctly characterise all method sends and receives into internal, ancestor, external, etc. For example, the source code of many OO languages may provide syntactic hints as to references to local or inherited instance variables. However, the situation is much more complicated when we consider references to instance variables in other hierarchies. If we could recognise `get` methods, then we could distinguish external method calls into (i) "real" method calls and (ii) "get" calls. However, in certain OO languages (e.g. Smalltalk) it is hard to automatically recognise a `get` method. A common construct is to extend a `get` method such that if a variable is not found, it is initialised. Is this still a `get` method? Or is it a "real" method call?

More generally, generating the correct call graph from OO source code is an unsolved problem. Murphy *et. al.* caution that in languages that support pointer to arbitrary constructs, then the problem is fundamentally intractable [14]. Different call graph generators tame this computational problem via a variety of heuristic design decisions. These heuristics alter the call graphs generated. For example, Murphy *et. al.* report significant differences in the graphs produced by different call graph generators [14].

Fortunately, for the purposes of assessing information hiding, two *simplification studies* (see Sections 3.4.1 & 3.4.2), suggests we can ignore many of the distinctions of Sections 3.1 & 3.2.

### 3.4.1 Simplification Study #1

A random selection of 29 classes from a Smalltalk application (see $\mathcal{APP}_v$ in Section 4.1) were randomly chosen using a spreadsheet's random number generator. Classes which were excessively large ($<$ 300 message sends), or those which had no source code were rejected from the study. The remaining 26 classes had 386 methods. Each message was manually inspected to see what messages it sent. Each message send was manually categorised into (i) a call back to this class or its parents (i.e. $P_m + L_m$); (ii) a call down the hierarchy (i.e. a $C_m$ reference) and (iii) a call to other hierarchies (i.e. $E_m$). Three observations were made:

- When messages arrive at an object, only 20% of them ascend the class hierarchy. That is 80% of messages are handled by the object that receives them. We will return to this observation later (see Section 6).
- No $C_m$ references were found; i.e. $C_m \approx 0$.
- The ratio of the $(P_m + L_m)$ to $E_m$ concurs with known ratios of $(A_m + I_m)$ to $E_m$ found in a study of 1,643 classes by the Haynes parser (see Section 4.1). Therefore, for the purposes of assessing information hiding, we can use $A_m$, $I_m$ and $E_m$ rather than $P_m$, $L_m$ and $E_m$.

### 3.4.2 Simplification Study #2

Elsewhere [7], we have assessed the importance of adding in references to local, inherited, and external instance variables (denoted $I_v, A_v, E_v$ respectively). In one variant of the Haynes parser, $I_v$ and $A_v$ counts were added to the $I_m$ and $A_m$ counts. This variant was compared to another version of the parser which ignored the $I_v$ and $A_b$ counts. The resulting numbers only differed in the second decimal place. Hence, we argue that we can ignore the variable references.

### 3.5 A Simpler Test for Information Hiding

The simplification studies suggest that, for our purposes, the values of $E_v$, $I_v$, $A_v$, $C_m$ are superfluous. Our test for information hiding (Equation 1) therefore reduces to:

$$E_m < I_m + A_m \quad (2)$$

For the purposes of comparison, $I_m$, $A_m$, and $E_m$ are normalised by expressing them as ratios of the total number of calls. Once normalised: $I_m + A_m + E_m = 1$. Suppose an application $\mathcal{APP}_x$ contains $N$ classes. For each class $C^w \in \mathcal{APP}_x$, we can compute $< i_w, e_w, a_w >$. Once this is known, we can compute the mean $< i', e', a' >$, standard deviation $< i_\mu, e_\mu, a_\mu >$ and standard error of the mean $< i_\sigma, e_\sigma, a_\sigma >$ [1] for $\mathcal{APP}_x$.

## 4 The Haynes Parser

While sophisticated schemes exist for detailed type inferencing (e.g. [1, 2, 6, 15]) we have found that, for generating ratios of $I_m$ to $A_m$ to $E_m$, a simple "brute-force" approach suffices. The Haynes parser [8] first collects lists of method names that are only defined in one class. Such unique names can be used to quickly determine the class type $X$ which is sent a message from a method. Our experience with Smalltalk code suggests that 40% of the method references can be resolved at this point.

Next, the program parses for the special special Smalltalk variables *self* and *super* in order to detect calls back into the current hierarchy. This step resolves a further 30% of the method references.

Next, the parser uses a hand-built library of commonly-used method names to resolve certain common calls [2]. This step resolved a further 10 to 20% of the message sends.

Lastly, the parser assigns the class type $X$ for any remaining unresolved calls heuristically using the distributions computed from the above techniques. Experimentally, we cannot detect any statistical differences between such automatically generated call graphs and call graphs built manually by programmers reading the code [8].

[1] Recall that $\sigma = \frac{\mu}{\sqrt{N-1}}$.

[2] For example, any call to `*` is taken to be a call to the most general numeric class NUMBER; all calls to conditions (e.g. `ifTrue:`, `ifFalse:` are taken to be a call to the most general conditional class BOOLEAN.

### 4.1 Analysis of Five Smalltalk Applications

Figure 2 shows the ratios of calls in different directions for 1,643 classes and 194,451 method calls generated from five applications using a Haynes parser that ignores variable references. The applications were:

- $\mathcal{APP}_f$: *FinApp* is an anonymous commercial applications for the financial market.
- $\mathcal{APP}_e$: *Envy* is a source code control system from Object Technology International.
- $\mathcal{APP}_v$: *VisualAge* is a general visual development tool with built-in database hooks from IBM. *VisualAge* is built on top of *IBM Smalltalk*.
- $\mathcal{APP}_i$: *IBM Smalltalk*.
- $\mathcal{APP}_m$: *Metric* is the source code of the Haynes parser.

The standard deviations shown in Figure 2 are very large. However, the sample sizes are large enough for the numbers to still be meaningful. Applying a two-tailed t-test, we explored the null hypothesis that $\mathcal{APP}_\alpha.x' = \mathcal{APP}_\beta.x'$ for $\alpha, \beta \in \{f, e, v, i, m\}$ and $x' \in \{i, e, p\}$. If we could reject the null hypothesis that *any* of $i', e', p'$ was the same for two applications, then we rejected the hypothesis that the mean standard deviation of the two applications was the same. In all cases, we could reject the null hypothesis at the 0.05 level of significance.

Figure 2 tells us that the mean values for $I_m, E_m, A_m$ are 0.38, 0.5, and 0.12 respectively. Applying Equation 2, this experiment does not confirm that encapsulation is likely in the studied applications (since $E_m = (I_m + A_m)$). On the contrary, it suggest that inter-hierarchy behaviour in the surveyed systems is just as common as intra-hierarchy behaviour.

## 5 Effective Class Size

The Haynes parser also lets us find the average size of a class. We define the *size of a class* to be the sum of the message sends of its own methods. We prefer message sends to lines of code since we have found that there is less variance in the correlation between effort and size using message sends as a measure of software size than lines of code [9]. Metrics based on message sends are more uniform over

| $\mathcal{APP}$ | $N$ | $i'$ | $i_\mu$ | $i_\sigma$ | $e'$ | $e_\mu$ | $e_\sigma$ | $a'$ | $a_\mu$ | $a_\sigma$ |
|---|---|---|---|---|---|---|---|---|---|---|
| f:FinApp | 313 | 0.44 | 0.29 | 0.02 | 0.47 | 0.26 | 0.01 | 0.09 | 0.10 | 0.01 |
| e:Envy | 123 | 0.28 | 0.18 | 0.02 | 0.57 | 0.22 | 0.02 | 0.16 | 0.16 | 0.01 |
| v:VisualAge | 527 | 0.31 | 0.18 | 0.01 | 0.49 | 0.19 | 0.01 | 0.20 | 0.13 | 0.01 |
| i:IBM Smalltalk | 666 | 0.56 | 0.36 | 0.01 | 0.37 | 0.33 | 0.01 | 0.07 | 0.12 | 0.00 |
| m:Metric | 14 | 0.31 | 0.26 | 0.07 | 0.60 | 0.24 | 0.07 | 0.08 | 0.11 | 0.03 |
| a:All | 1643 (total) | 0.38 (mean $I_m$) | - | - | 0.50 (mean $E_m$) | - | - | 0.12 (mean $A_m$) | - | - |

Figure 2: Mean call directions in five applications

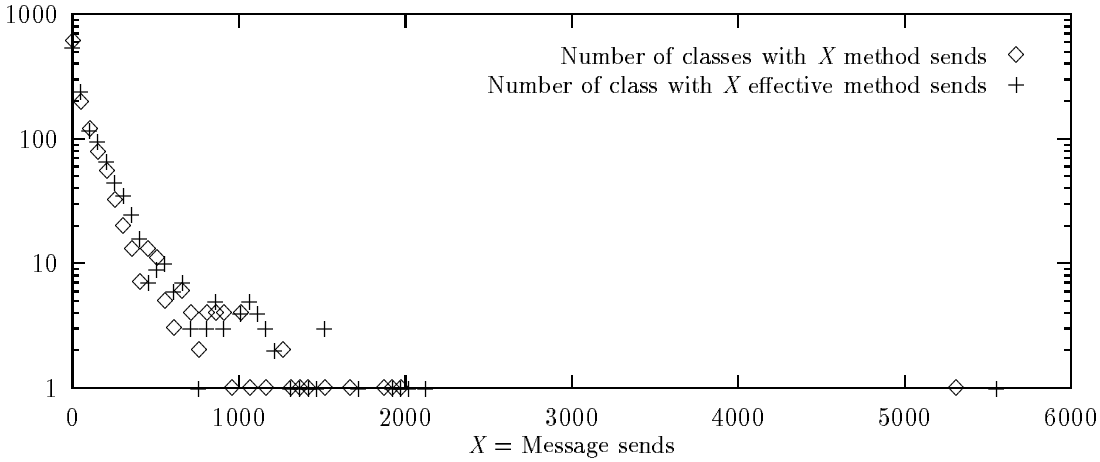


Figure 3: Message sends.

different OO languages that other measures. For example, based on an analysis of around 1 million lines of source (including several Smalltalks and a range of C++ class libraries), we have argued elsewhere [9] that for these languages, the following relationship holds. If $MS$ be the number of message sends and $LOC$ be the source lines of code, then:

$$LOC = \frac{MS + 9.6025}{1.153} \quad (3)$$

Our definition of class size ignores the methods used in parent classes. We define the *effective size of a class* to be the size of a class plus the sum of the messages sends in the methods called in its parent classes. To compute the effective size, we added a small report generator to the internals of the Haynes parser. For each class, we returned the message sends in the union of the methods generating the $A_m$ references.

The number of message sends per class and effective message sends per class are shown in Figure 3. The mean class sizes and effective class sizes for $\mathcal{APP}_v$ and $\mathcal{APP}_i$ are shown in Figure 4. In Figure 4:

- Mean lines of code for the class and effective class size are calculated by applying Equation 3 to the message sends figures.
- Figure 5 shows the distribution of number of methods per class in Digitalk's Smalltalk/V for Windows, version 2[3]. The mean of these distributions is 34. In Figure 4, the mean lines of code per method is calculated by dividing lines of code per class by 34.

In summary, the average method is small (about 4 lines) and the average class contains a small number (about 34) of small methods. In order

[3]Note that for this analysis, we have excluded methods inherited from the Smalltalk root class OBJECT since OBJECT stores what is the same for all classes.

| | *Class size* | *Effective class size* |
|---|---|---|
| Message sends | 129 | 154 |
| Lines of code | 120 | 142 |
| Mean lines of code per method | 3.5 | 4.2 |

Figure 4: Mean class and method sizes

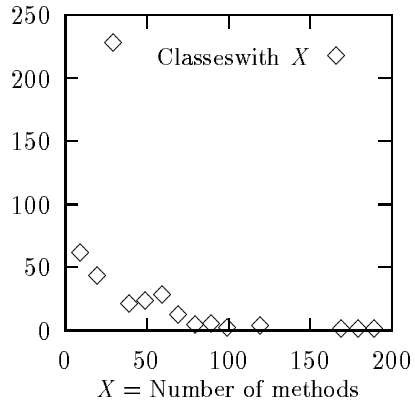


Figure 5: Class size (number of methods).

to perform any non-trivial task, such an average class would have to collaborate extensively with other classes. Extensive intra-hierarchy collaboration may not violate the principles of information hiding since it could be argued that it is reasonable that sibling sub-classes know something about each other. However, recall the results of Section 4.1: at least half the message sends are across class hierarchy boundaries. The classes studied here exhibit extensive inter-hierarchy collaboration. In such an architecture, we doubt that the processing of any particular class can be considered without respect to its connections to classes in different hierarchies.

# 6 Use of Inheritance

In Smalltalk/V for Windows (version 2.0), the average class contains $N$ methods can access $2 * N$ methods from its parents. However, based on three different measures, we say that child classes use only a small subset of the methods which they potentially could inherit from their parents:

- Figure 2 shows that in all the surveyed applications inheritance accounts for only 10% to 20% of all method calls.
- Figure 4 shows that that the mean effective class size (154 message sends) is only 19% bigger than the the mean class size (129 message sends); i.e. classes make far use of their own services than their parent's services.
- Simplification study #1 tells us that only 20% of the messages received by a class are referred up the hierarchy.

# 7 Related Work

We believe that the growing use of *use cases* [10] is further evidence for our observation that real-world OO programs make limited use of information hiding. A class is not an independent design concept in its own right. A class's public interface is insufficient information to understand it. Use cases are also required to manage class collaboration knowledge in OO designs. A use case is an informal technique for requirements capture. For example, Rumbaugh argues that use cases are the basis of defining functional requirements, deriving objects, allocating functions to objects, and designing the interface [17]. Use cases are becoming a dominant theme in OO design and have been adopted by all the major methodologies.

Our work also gives some support for the OO design patterns paradigm [5]. This call graph research suggests that the natural unit of division in a OO system is not the class or the class hierarchy. OO systems design and testing should focus on groups of classes that collaborate to achieve some goal. These groups of classes may collaborate extensively across class hierarchy boundaries. This inter-hierarchy collaboration fits the patterns research better than the traditional view of classes as independent processing units.

One explanation for the relatively low values of $A_m$ in Figure 2 is that Smalltalk is a single parent inheritance language. In a single parent OO language, if a class requires the services of more than one other class, it must reference an instance of the other class via an $E_m$ call. However, we note that Phipps [16] has made a similar observation about low-levels of inheritance use in C++, a multiple-inheritance language.

One drawback with our analysis is that it ignores the runtime behaviour of an OO system. Call graphs are static. They describe the possible paths that could be taken by a running OO system. If a running system only ever uses some subset of these paths, then we should base our information hiding/

inheritance usage conclusions on those portions of the program actually exercised at runtime. We are currently exploring metering the runtime of Smalltalk systems (i.e. in the manner of [1, 6]) order to confirm/refute the above static analysis.

Murphy *et. al.* use call graphs to make value statements about an OO design [13]. In their *software reflexion model*, call graphs between high-level software modules can be generated and compared to the collaborations stated in the design documents. However, there is very little other related work that uses metrics to actively explore the utility of a programming paradigm. Call graph research typically focuses on runtime optimisation. Grove *et. al.* seek classes that are highly connected in the call graph [6] Such classes are candidates for optimisation. Agesen *et. al.* [1] review different call graph generation techniques, again with a view to optimisation.

More generally, measurements of software are rarely used to critically assess programming paradigms (a point expanded on by [4]). We have argued elsewhere that we need more active experimentation in software engineering [11]. More precisely, we should declare an active hypothesis (e.g. OO applications make extensive use of information hiding) then define a test which has the potential to refute that hypothesis (e.g. Equation 1). Pre-experimental intuitions, however convincing they may appear, do not always hold true in the real world.

# 8 Conclusion

Based on an analysis of call graphs generated from Smalltalk systems, we have reported two counter-intuitive observations:

1. Low-levels of inheritance in Smalltalk systems
2. Low usages of information hiding in Smalltalk systems at the class and class hierarchy level.

Results from other languages [16] and the growing usage of use cases in OO design theory make us suspect that this result is applicable to other OO languages.

The picture that we have painted is of classes talking as much to their distant neighbors as to themselves. The reason for this is that, as we have seen, classes are very small computational units. When executing, classes have to use services defined outside of themselves. We have shown here that classes access services outside their own hierarchy at least 50% of the time. More generally, when we divide a domain model into lots of small parts (classes), we require significant amounts of "glue" to make them all fit together again.

# 9 Acknowledgments

James Noble defined the $P_m$ and $L_m$ directions. Further, his comments on $C_m$ prompted simplification study #1.

# References

[1] O. Agesen and U. Holzle. Type Feedback vs Concrete Type Inference: A Comparison of Optimisation Techniques for OO Languages. In *OOPSLA '95*, pages 91–107, 1995.

[2] O. Agesen, J. Palsberg, and M. Schwartzbach. Analysis of Objects with Dynamic and Multiple Inheritance. In *ECOOP'93, Seventh European Conference on Object-Oriented Programming*, pages 329–349. Springer-Verlag, 1993.

[3] G. Booch. *Object-Oriented Design with Applications (second edition).* Benjamin/ Cummings, 1994.

[4] N. Fenton, S.L. Pfleeger, and R.L. Glass. Science and Substance: A Challenge to Software Engineers. *IEEE Software*, pages 86–95, July 1994.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[6] D. Grove, J. Dean, C. Garnett, and C. Chambers. Profile Guided Reciever Class Prediction. In *OOPSLA '95*, pages 108–123, 1995.

[7] P. Haynes, T. Menzies, and R.F. Cohen. Visualisations of Large Object-Oriented Systems. Technical Report TR95-4, Department of Software Development, Monash University, 1995. To appear in the book **Software Visualisations**, Eades. P. (ed).

[8] P. Haynes and T.J. Menzies. The Effects of Class Coupling on Class Size in Smalltalk Systems. In *Tools '94*, pages 121–129. Prentice Hall, 1994.

[9] P. Haynes, T.Menzies, and G. Phipps. Using The Size of Classes and Methods as the Basis for Early Effort Prediction; Empirical Observations, Initial Application; A Practitioners Experience Report. In *OOPSLA Workshop on OO Process and Metrics for Effort Estimation*, 1995.

[10] I. Jacobson and M. Christerson. A Growing Consensus on Use Cases. *JOOP*, pages 15–19, 1995.

[11] T.J. Menzies and P. Haynes. The Methodologies of Methodologies; or, Evaluating Current Methodologies: Why and How. In *Tools Pacific '94*, pages 83–92. Prentice-Hall, 1994.

[12] B Meyer. *Object-oriented Software Construction.* Prentice-Hall,Hemel Hemstead, 1988.

[13] G.C. Murphy and D. Notkin. Software Reflexion Models: Bridging the Gap Between Source and High-Level Models. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE '95)*, 1995.

[14] G.C. Murphy, D. Notkin, and E.S.C. Lan. An Empirical Study of Static Call Graph Extractors. Technical Report TR95-8-01, Department of Computer Science & Engineering, University of Washington, 1995.

[15] J. Palsberg and M. Schwartzbach. Object-Oriented Type Inference. In *Proc. OOPSLA'91, ACM SIGPLAN Sixth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 146–161, 1991.

[16] G. Phipps. The Structure of Large C++ Programs. In *Tools Pacific 18*, pages 253–264, 1995.

[17] J. Rumbaugh. Getting Started: Using Use Cases to Capture Requirements. *JOOP*, pages 8–23, 1994.

[18] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

Some of the Haynes & Menzies papers can be obtained from *http://www.sd.monash.edu.au/ ~timm/pub/ docs/papersonly.html.*