Transactions on Information and Communications Technologies vol 9, © 1994 WIT Press, www.witpress.com, ISSN 1743-3517

Complexity metrics for quality assessment of object-oriented design

T.P. Hopkins

Computer Science Department, University of Manchester, Oxford Road, Manchester, M13 9PL, UK

ABSTRACT

This paper considers the use of *interface complexity* metrics for the quantitative analysis of the quality of software design performed using object-oriented development methods. The techniques considered can be applied before coding is started, as part of the design review process, or when coding is complete. The use of complexity metrics for reusability assessment is also considered.

INTRODUCTION

Object-oriented analysis and design is relatively new, and many organisations are now attempting to introduce this style of development. Introducing a new development approach is fraught with risk, especially if the organisation or the individual developers have little experience with the new approach. One way of reducing this risk is to use design guidelines and metrics to assess the newly created designs. This paper considers the use of *interface complexity* measures for the assessment of the quality of object-oriented designs.

Why Quality Metrics

What we are really trying to do is get a measure of the difficulty a human has with understanding a particular software structure – this is a psychological question. The human may have to understand the software design for many reasons: to modify it, to implement it in a programming language, to devise tests, to find and fix bugs, or even to reuse the resulting design and code. In practice, however, we make these measurements in several steps: first we take a system design, program code, or something similar; also we devise some measures to be applied to our design or code. Clearly, these measures are based on our experience and intuitive understanding of what goes on, both in the resulting programs themselves and also in the whole program development process. Nevertheless, it should be stressed from the start that these measures mack a proper scientific basis. The measures can be very simple, such as counting the number of lines of code, or they might be more sophisticated models separately assessing many aspects.

We also must ascribe a 'meaning' to the measures – 'complexity', perhaps, or 'reusability'. We then compute these measures for the design or code to hand, and interpret the resulting numbers as some measure of 'quality': the likelyhood of bugs or design errors and the ease with which they might be expected to be found and fixed, the ease of reuse of some software, or even some abstract 'goodness of design'. Of course, we must attempt to verify our metrics, by (for example) comparing the predicted bug count with that actually found in practice; note that this is not possible in some cases. Despite any post-hoc verification, however, the lack of a physical (i.e. *scientific*) model of cause and effect means that the process lacks rigour. Nevertheless, there is ample evidence that such metrics can be useful, and therefore we should look to them for guidance in our software development process.

By software quality, we mean many related things. For example, high quality designs are easy to implement, and provide all the features required by the users. They are also *robust* in the face of changes in users' requirements, as well as changes in the underlying hardware, operating system, and even the programming language. High quality programs are bug-free, and when errors are discovered, they can be located and fixed readily without the introduction of further bugs. They are also robust in the face of design changes, and facilitate the introduction of new features. The object-oriented approach to software development is widely regarded as being sympathetic to more effective software reuse, so metrics which equate 'quality' with 'robustness' are of particular interest. We will discuss metrics for reusability measures later.

Kinds of Quality Metric

We can identify (at least) two useful kinds of quality metrics: *design* metrics, which allow a design's quality to be estimated before any code is produced; and *code* metrics which give an indication of the quality of the implementation. We might expect design metrics to allow us to estimate the design quality, in a very general sense, and to guide designers towards better designs. Having a framework in which to be able to compare alternative designs, and decide on the best of these, is useful, particularly when working in a style (such as the object-oriented approach) which might be unfamiliar. It is desirable to be able to identify complex parts of design, so that we can concentrate our design, coding and testing effort appropriately. Indeed, being able to identify areas which are under- (or perhaps over-) designed, or even under- or over-analysed, is helpful in allocating resources, and in deciding whether a design really is good enough, or stable enough, to be implemented. Thus, we can see that design metrics can assist the management control of development progress during the analysis and design phases, an area which is notoriously hard to control, and can allow quality assessment before any code is available.

Similarly, we might expect code metrics to allow use to perform some quality assessments. We can measure some aspects of coding practices; for example, in the object-oriented approach, we can mark down implementations which make extensive use of 'case analysis' but mark up implementations which have many polymorphic methods. We can also estimate the testing requirements, and such metrics might assist us in devising a testing strategy. We might also be able to assess how reusable the code is.

However, with object-oriented methods, there is often much less distinction between the traditionally separate areas of 'analysis', 'design' and 'code'. Some writers (e.g. [1]) are now advocating more iterative development processes, with 'cyclic' or 'spiral' development models replacing the traditional 'waterfall' model. Since the principles of good (and bad) object-oriented design are very directly reflected in the resulting code, many of the metrics which have been suggested are difficult to classify as either 'design' or 'code'. Some work ostensibly on design quality metrics [2] suggest measures which require detailed information which is unlikely to be available until an implementation is near completion.

A poor object-oriented design may actually manifest itself as bad coding practices (such as case analysis). Some papers on design quality [3] activity encourage the identification of design flaws by inspecting the code. Indeed, some work on 'programming style' (e.g. [4, 5]) contain much advice which is arguably about good and bad design. Another difference from traditional software development is that, with object-oriented methods, there is often more effort in the analysis and design phases, and less in the coding phase – this is another reason why metrics to assist measurement of the progress of a design are important. With increased generality in design, and emphasis on both design and code reuse, the development process of object-oriented systems often simultaneously leads to more functionality and less code as the design quality is improved. Finally, the development of design *frameworks* [6] (also called design *clichés*) and the possible development of a marketplace in reusable object-oriented software components [7] may also make it increasingly commonplace to evaluate the design quality by assessing an actual implementation.

Object-oriented Design

The basic tenet of object-oriented design is that systems are defined and implemented as groups of cooperating, communicating *objects*. Each of these objects is an *instance* of a *class*, which describes the data which that object can hold, and the functions it can perform. Note that each individual object has a private data area, but shares a single version of the code. The private data, referenced by *instance variables* (also called 'attributes', 'data members' or 'slots'), are usually either simple types (string, integer and so on), or references (pointers) to other objects. Objects are usually regarded as encapsulating the data stored within them, not allowing access to internal data items from outside.

New objects can be created and destroyed at any time, and the relationships (maintained by references from instance variables) can change over time, as the system runs. Typically, we can view one part of object-oriented development as the (dynamic) construction of complex objects from less complex ones, in a 'compositional' or 'LEGO-block' programming style; this is one way of getting software reuse in an object-oriented system.

Classes define *methods* (also known as 'member functions', 'actions' or 'routines'), which are called or activated using 'messages' sent by other objects. Obviously, in order to call a method defined by the class of some object, a reference (often retained by an instance variable) to that object is required. Since, in many cases, an instance variable can refer to objects of different classes at different times, there is no static relationship between the name used when the member function is called and the body of code so activated: different classes can define a method of the same name but with different (but compatible) behaviour. This dynamic *polymorphic* behaviour allows object-oriented systems to be very flexible and general, and paves the way for more effective reuse.

Classes can also be related by *inheritance*, where a new class (a *subclass* or 'derived class') can inherit some of the behaviour of one or more other classes (*superclasses* or 'base classes'). Some classes will *abstract* (or 'deferred'), which cannot be instantiated and provide only common general behaviour, while *concrete* classes can be instantiated and refine or specialise the inherited behaviour. With careful use, this inheritance mechanism allows for common functionality to be put in one place, rather than being needlessly duplicated throughout the system, and provides another very powerful mechanism for software reuse. Note that class inheritance and object composition are complementary mechanisms, and some important design decisions are involved in deciding which to use in a particular case.

Classes are used in two different ways: firstly, as 'templates' for the instantiation of objects which are then parameterised (by setting attributes) and 'connected up' in relationships with other objects, and secondly as the basis for defining new classes by inheritance. Classes therefore have two different interfaces, corresponding to these different uses: the *public* (or 'client' or 'use') interface, for general use, and the *protected* (or 'subclass' or 'inheritance') interface, for use only in subclasses. Also, we will expect to find some code in 'private' methods, as well as ordinary procedures and functions, which are not accessible through either of the interfaces. Some methods may be in both the public and protected interfaces. Note that the distinction between these different interfaces is similar to that embodied in some languages, notably C++ [8]. Also, note that the instance variables may, or may not, be visible in either of these interfaces; one common situation is that instance variables are not directly visible through the public interface, but can be accessed through the protected interface.

Already, a great many different object-oriented analysis and design methods have been espoused. This plethora of approaches do have a fair amount in common (which is to be expected, since they are all supposed to be 'object-oriented'); here, we are only interested in the outcome of the design method. In this paper, we will assume that certain information is available from the results of object-oriented analysis and design, regardless of the exact approach used; for example, Booch [9] advocates the use of class and method templates which capture much of the information required.

The outcome of a design process will include descriptions of classes, outlining their functionality, describing their instance variables, and defining the methods in both the public and protected interfaces. Also, there will be a definition of the inter-class relationships, especially inheritance, but might also include other kinds of 'use' information. There will also be some description of inter-object relationships, especially the part-whole and cooperative object structures which will be formed. Ideally, the design information should be captured in some form of machine representation (e.g. a CASE tool), so that (amongst other reasons) the design metric computations can be done automatically.

Eventually, the design will be coding in some programming language, ideally an object-oriented language. The resulting source code should be accessible by some kind of tool (such as a CASE tool or a program development environment) which can also be used to extract information to allow design metrics to be computed. Much of the design information required for quality assessment is available from the source code of many object-oriented programming languages, so that design quality metrics can also be used to assess some aspects of the code quality, including code from prototypes, frameworks and bought-in software components.

COMPLEXITY METRICS FOR OBJECT SYSTEMS

Complexity metrics for software systems have been known for a long time [10]. It is natural from the object-oriented viewpoint to concentrate on the interfaces to classes; in particular, the two different interfaces identified earlier. We can argue that some measure of the complexity of the interface can provide important information on the ease of understanding of the class interface (which is needed for reuse, maintenance, rework and redevelopment), as well as information on testing strategy (from a 'black-box' perspective). Furthermore, it might be reasonable to assume that the complexity of the interfaces will gave some idea of how difficult the class will be to design (correctly) and implement.

We expect to perform much application development by *composition* – creating instances of existing classes and 'assembling' them to form some more complex software structure; in this case, we will be using the public interface discussed previously. We also will use inheritance to add new refined subclasses, using the protected interface. Since these two kinds of design activity are different, it seems sensible separately to estimate the complexity of the public and subclass interfaces.

Method Complexity

The smallest unit usually identified by the design process are the methods associated with a class. We can estimate the *interface complexity* for a single method IC_{meth} as follows:

$$IC_{meth} = N_{return-classes} + \sum_{i=1}^{N_{args}} N_{arg-classes}(i)$$

where $N_{return-classes}$ is the number of different classes (or types) possibly returned from this method, N_{args} is the number of arguments to the method, and $N_{arg-classes}(i)$ is the number of different classes (or types) for each of the arguments i. This implies that a method is regarded as more complex if it has a large number of arguments, or if the arguments can be of many different kinds, or if the result returned to the caller can be of many different kinds.

For example (using C++ syntax), the method:

```
int length()
```

defined in class `String` takes no arguments and always answers with an integer value, has complexity `1`. As another example (using Smalltalk[11] syntax), the method:

```
scaleBy: factor
```

defined in class Point always returns an instance of class Point, but has an argument which can be an instance of one of six classes (either Point or the Number subclasses LargeNegativeInteger, LargePositiveInteger, SmallInteger, Float or Fraction), has complexity 7.

We can already see some evidence that low complexity can be associated with good design. For example, it is considered [3] that methods should have a small number of arguments, which can be of a wide range of classes. If the scaleBy: method in class Point was replaced by:

```
scaleByX: xFactor y: yFactor
```

where xFactor and yFactor can be of any of the five number classes mentioned above, then the complexity measure would be 11 in this case.

We can also estimate the interface complexity for a single instance variable IC_{iv} with both read and modify permission as:

$$IC_{iv} = 2 \times N_{classes}$$

where $N_{classes}$ is the number of different classes (or types) to which an instance variable can refer. Note that this is equivalent to two *accessor* methods, one of which simply returns the value of the instance variable, and the other which sets the variable.

For example, if a class `Person` defines an instance variable `name` which will always be of type `String`, then equivalently we might provide two methods `getName` and `setName` to return and set the instance variable respectively. The complexity of each of these two methods is 1, so the total complexity is 2. Of course, if strict encapsulation is implied, then instance variables are not available in the public interface, but are often accessible through the protected interface.

If the instance variable is read-only, then the interface complexity is reduced:

$$IC_{iv} = N_{classes}$$

This is equivalent to a single accessor method, which simply returns the value of the instance variable.

Class Interface Complexity

Given a metric for the complexity for individual methods, it is a reasonable step to compute an overall complexity for each of the two interfaces to a class. For example, we can compute the complexity of the public interface IC_{pub} by summing the complexity of each method which appears in that interface:

$$IC_{pub} = \sum_{i=1}^{N_{pub}} IC_{meth-pub}(i)$$

where N_{pub} is the number of methods in the public interface and $IC_{meth-pub}(i)$ is the complexity of each of these methods. Note that this metric attempts to give a measure of the client interface complexity which is independent of the exact nature of the interface; for example, a change which reduces the number of arguments to methods or reduces the number of methods, but increases the number of different kinds of arguments might result in approximately the same measure.

We can use the same approach to computing the complexity due to methods and instance variables in the protected interface. Note that it is rare for the public interface to include any instance variables, while it is much more common for instance variables defined in a superclass to be made available to subclasses. In this case, therefore, the protected interface complexity IC_{prot} includes the complexity of the instance variables:

$$IC_{prot} = \sum_{i=1}^{N_{prot}} IC_{meth-prot}(i) + \sum_{j=1}^{I_{prot}} IC_{iv}(j)$$

where N_{prot} is the number of methods in the protected interface and $IC_{meth-prot}$ is the complexity of each of these methods; I_{prot} is the number of instance variables in the protected interface and $IC_{iv}(j)$ is the complexity associated with each of these variables.

Having got a complexity measure for both of the interfaces to a class, we might want to combine these to form a single complexity measure. However, adding together the two numbers may be inadequate. This is because of the nature of the subclass interface: it is an interface explicitly provided so that subclasses can modify the operation in ways unforseen by the original designer. This arbitrary extensibility makes the subclass interface very difficult (possibly impossible) to test, and it is probably fair to weight the subclass interface rather more heavily; perhaps a weighting factor of five might be appropriate. This quite arbitrary factor will of course have to be verified. So, a measure of complexity for a single class IC_{class} (considered *individually*), is:

$$IC_{class} = IC_{pub} + (weight \times IC_{prot})$$

where *weight* is the weighting factor applied to the protected interface.

Note that any method which appears in both interfaces is counted twice. Such a method may be defined so that a different implementation *must* be provided in subclasses (a *deferred* or *pure virtual* function or a subclassResponsibility method), or may be only *optionally* re-implemented by subclasses.

Implications of Inheritance and Polymorphism

Now that we have a measure of class complexity, there is a temptation to estimate the overall complexity of an object-oriented design by simply summing the complexity of each class. This approach is inappropriate, since classes are coupled one to another by inheritance. For example, public methods in a class are usually all inherited by subclasses; therefore, the *true* public interface complexity of a class should include all of the methods inherited, as well as the ones directly implemented. However, note that methods which appear two or more times in the interface should *not* be counted twice; for example, such methods may be defined as subclassResponsibility in a superclass and implemented in a subclass.

This discounting of duplicated methods in the public interface can be justified by observing that, from the point of view of understanding a class interface (so that the class can be reused, for example), all classes sharing some particular method share that aspect of the interface, and we only need to understand that aspect once. This can result in a very significant reduction in the computed complexity. For example, in a class hierarchy of 3D graphics objects (spheres, cubes, toroids and so on), all classes inherit much interface from an abstract class ThreeDObject. Many methods which have sensible behaviour for all kinds of object (such as transformBy:, moveTo: and so on) are defined by class ThreeDObject and inherited by all 75 subclasses. In this case, the complexity of the transformBy: method is only counted once.

The protected interface has an inherited component as well. Instance variables inherited by a class will usually be available in that class's protected interface, in a fashion identical to instance variables actually defined by that class. Protected methods inherited by a class will also appear in that class's protected interface, unless an implementation is provided which not expected to by again overridden in further subclasses (i.e. effectively made private). In this case, the method does not appear in the class's protected complexity computation. Also, inherited protected methods which are re-implemented, but continue to appear in the protected interface should only be counted once. Thus, under some circumstances, methods can be discounted from the protected interface.

One view of the purpose of inheritance is to make it easy to provide many classes of objects which have very similar (*compatible*) behaviour, without the difficultly of implementing every method separately. Such compatible classes can be used interchangeably (*polymorphically*), so that one can be removed and another inserted to change the behaviour of the application in a controlled fashion. This polymorphic approach means that object-oriented systems can be readily changed (for maintenance or extension), or the components reused in other applications.

In some circumstances, there may be classes which can be used in this compatible fashion, even where the classes which are *not* related by inheritance. In this case, it is sensible to reduce the overall computed interface complexity by discounting methods in the public interface, as considered above. Note that some object-oriented programming languages will not permit this particular flexibility; this discounting is therefore only appropriate where the design is specifically targeted at a programming language where this can be achieved.

Another aspect which can arise is when two or more classes have methods which are not quite compatible, but are very close. These classes may, or may not, be related by inheritance. For example, consider the classes Point (representing locations in a two-dimensional space) and ThreeDPoint (for three dimensions). These classes both define a method scaleBy:; for class Point, the argument can be a Point or number (as discussed before), while for class ThreeDPoint the argument can also be a ThreeDPoint. The two methods are not strictly compatible, and have a different complexity measure; nevertheless, they are rather similar. One solution to this problem is *not* to discount one of these methods in the overall complexity; this encourages the redesign of the scaleBy: method in class Point to accept the same kinds of argument as the version in ThreeDPoint.

A larger number of well-designed classes which are compatible (and perhaps share a common superclass) are not necessarily more complex, and often may be less complex, than a single class with the same functionality. For example, a viewer for 3D objects was originally implemented as a single class, which provided a large amount of functionality: a viewpoint transformation, projection and a variety of rendering techniques. The complexity of the public interface for this class was high, as might be expected from the considerable amount of function supported. Also, metrics for internal class cohesion (see later) suggested that this class had a poor structure.

After a re-design, the viewer class was broken up, with different abstract classes provided for aspects such as view transformation, projection and rendering. For each of these classes, different subclasses were defined; for example, different rendering classes where provided for 'wire-frame' display and ray-tracing. Despite that fact that some tens of classes are now provided where previously only one was implemented, the aggregate interface complexity of the classes as a whole was significantly reduced. The cohesion of each class was also much improved.

Complexity Estimation for Testing

So far, we have considered how to combine our complexity measures to estimate how hard it is to understand a group of classes, and therefore be able to use (i.e. reuse) them effectively. However, the perspective when considering the testing of classes is somewhat different. For example, the abstract class ThreeDObject defines the transformBy: method; this is re-implemented in around 45 of the subclasses. To estimate ease of understanding, these re-implementations can be discounted; however, each method will have a different implementation (often, very significantly different), and will have to be tested separately. So, to use interface complexity to estimate the testing requirements for an object-oriented design, we should *not* discount inherited methods.

Another aspect to be considered when we attempt to use interface complexity to estimate testing requirements is the provision of public 'convenience' methods. When classes are designed for very general use, and especially where extensive reuse is anticipated, then it is conventional to include large numbers of methods whose functionality is (strictly speaking) unnecessary and which add nothing to the basic operation of the class, but which might prove to be convenient for some future use. Such methods clearly add to the complexity of the public interface, and these methods will of course have to be tested. However, we might not wish to use interface complexity arguments to completely discourage designers from including convenience methods, as they might well enhance future reusability; one possible approach is to distinguish between 'essential' and 'convenience' methods in the interface, and weight the 'convenience' methods less highly during complexity metric computations. Note also that 'convenience' methods are often rather simple, and merely repackage existing functionality.

As an example, the ThreeDObject class introduced previously defines two related methods: transformBy:, which transforms the object itself, and transformedBy:, which *copies* the object and transformed the copy. Clearly, the second can also be achieved by client code first copying the object and using transformBy:; nevertheless, having both methods enhances the usefulness of all subclasses.

Note that this issue is a reflection of a general point about designing highly reusable software: if much effort is expended to make general-purpose software components, which can be used in many circumstances, then they will necessarily be more complex (both at the interface and internally) than components designed for a particular tightly-specified purpose. Reusable components will probably be harder to understand, implement and test, but need only be constructed once.

CONCLUSIONS

By using interface complexity metrics, we can reasonably compare alternative designs for the same application or system, meeting the same requirements specification. Of course, comparing designs for different systems does not really make any kind of sense. Our metrics may well identify parts of a design which are poor, or which have been poorly coded. However, metrics do not (directly) suggest alternatives; we need to invent design alternatives for comparison using suitable metrics. Design guidelines can be useful here. Another application are is the incorporation of complexity metrics into design reorganisation tools (e.g. [12]), so that designs or programs can be semi-automatically restructured to improve the design quality.

Some other work [13] on object-oriented design metrics is based on the familiar notions of 'coupling' and 'cohesion' between software components or modules. Classes should have a high level of internal cohesion, and as low a coupling with other classes as possible. These approaches have been applied in an object-oriented fashion [14, 2], in various ways. Some work has linked the complexity of a system to the coupling of the components of that system; however, from an object-oriented perspective, it is difficult to assess coupling, since the same component (class) may be used in many different circumstances (e.g. through extensive reuse), or different classes may be used interchangeably. Cohesion within classes has been linked to the *internal* complexity of those classes; the identification of 'sub-interfaces' within a class (such as a single instance variable and the methods using it) give rise to evaluations strikingly similar to cohesion measures.

Evaluating Metric Correctness

Metrics are an ad-hoc mapping of various computable measures about software structure onto some abstract notion of quality. The absence of an underlying physical or casual model for these kinds of metrics is a a fundamental problem for evaluation, since without a model it is hard to devise a scientific validation of the measures. In practice, however, we need to be sure that the metrics used genuinely measure what we anticipate: that 'better' numbers really indicate 'better' designs or implementations. The relationship had better be at least monotonic, and ideally linear (or some other well-understood mathematical function). Furthermore, when weighting factors are used, we need to determine these in some way.

Two possible approaches are, firstly, to compare the measures with 'expert opinion'. This is fraught with problems; first, find an expert you believe in! However, experts are notorious for not always agreeing with each other, so an exact approach is tricky. Alternatively, apply the metrics to some successfully implemented systems. Of course, the trouble with this approach is that not all successfully implemented systems are well-designed, and not all well-designed systems get to be implemented. Clearly, there are other significant factors here. While initial results suggest that the complexity metrics suggested in this paper might be useful, they cannot yet be regarded as anything other than plausible candidates for design quality assessment.

Metrics for Reuse

There are really two kinds of reuse measure: the first is a measurement of the amount (and effectiveness) of reuse actually achieved in a particular case. This *achieved reuse* is a relatively straightforward idea, and there are fairly obvious ways of measuring this. For example, the ratio of new to reused classes in a design or implementation gives some idea of the effectiveness of the reuse, and perhaps an indication of the future reusability of the parts reused. However, this approach is based on the (naive) assumption that all classes are comparable: as a counter-example, consider the case where we reuse nine very simple classes and build just one new and very complex class – are we really achieving 90% reuse?

For better measures of reuse achieved, better 'quantity' measures for classes are required. We could simply use some measure of code size, if this is available, but this still raises questions of comparability. Possibly, interface complexity measures could be used here, with the amount of reuse achieved depending on the ratio of the complexity of the new and pre-existing classes. This approach has not yet been explored in any systematic fashion, however. For another more sophisticated approach, Bieman [15] suggests separately assessing software re-use of components taking into account the amount of internal modification to those components.

The second reuse measure is much harder to assess: we want an estimation of the (likely) ease of reusing a software component (such as a class) *before* it has ever been reused. Also, we require an indication of how much reuse a class, or library of classes, will support. Some measures are required to assist in the design of genuinely reusable classes.

To reuse software, the interfaces have to be understood. A low client interface complexity of the entire library is therefore one indication of usefulness. Similarly, controlled polymorphism is desirable, since common compatible behaviour for many classes reduces the intellectual effort in understanding. Also, measures of the subclass interface complexity give some guides, as much reuse is enabled by inheritance. A low (non-inheritance) coupling measure also suggests ease of reuse of a class, as it does not need large numbers of other classes to be useful. However, it is better to use the other classes, rather than to re-implement everything inside a single class, as this will tend to increase the potential for reuse overall.

Nevertheless, all this is not enough to ensure effective reuse. There is an aphorism which states: "Software is not reusable, until it has already been reused – by someone else." In other words, there is nothing quite like a serious attempt at reusing code to expose hidden assumptions, incomplete interface specifications and a lack of understanding of the precise functionality of a class.

We believe that it is reasonable to suggest that object-oriented designs which are 'good' (i.e. low complexity interfaces) as assessed by the metrics discussed in this paper are more likely to be reusable. Such classes will be easier to understand, and good understanding is a pre-requisite to effective reuse. Low protected interface complexity makes it easier to add new subclasses, while a uniform public interface for many classes (with a low overall complexity) will make arbitrary composition of object easier. We believe that interface complexity metrics can be useful is assessing object-oriented developments, but they should always be treated with suspicion; they are no substitute for skill, experience and thought. In summary, design quality metrics are not a panacea for all ills, and interface complexity metrics are not perfect; we can expect continuing work in this area for some time.

References

[1] Barry W. Boehm, 'A Spiral Model of Software Development and Enhancement', *IEEE Computer*, May 1988, pp. 61-72.

[2] Shyam Chidamber and Chris Kemerer, 'Towards a Metrics Suite for Object Oriented Design', pp. 197–211, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications* (OOPSLA'91), October 1991.

[3] Ralph Johnson and Brian Foote, 'Designing Reusable Classes', pp. 22-35, *Journal of Object-Oriented Programming*, June 1988.

[4] Roxanna Rochat, *In Search of Good Smalltalk Programming Style*, Technical Report CR-86-19, Tektronix Laboratories, September 1986.

[5] K. Lieberherr and I. Holland, 'Assuring Good Style for Object-Oriented Programs', *IEEE Software*, September 1989.

[6] Ralph E. Johnson, 'Documenting Frameworks using Patterns', pp. 63–76, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications* (OOPSLA'92), October 1992.

[7] Brad J. Cox and Andrew J. Novobilski, *Object-Oriented Programming: An Evolutionary Approach* (second edition), Addison-Wesley, 1991.

[8] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.

[9] Grady Booch, *Object-oriented Design with Applications*, Addison-Wesley, 1991.

[10] T. J. McCabe, 'A Complexity Measure, pp. 308–320, *IEEE Transactions on Software Engineering*, vol. 2, no. 4, 1976.

[11] Adele Goldberg and David Robson, *Smalltalk-80: the Language and its Implementation*, Addison-Wesley, 1984.

[12] Bernd Hoeck, *A Framework for Semi-automatic Reorganisation of Object-oriented Design and Code*, M.Sc. Thesis, Computer Science Department, University of Manchester, September 1993.

[13] L. L. Constantine and E. Yourdon, *Structured Design*, Prentice-Hall, 1979.

[14] D. Embley and S. Woodfield, 'Cohesion and Coupling for Abstract Data Types', pp. 229-234, *Proceeding of the Sixth Annual Conference on Computers and Communication*, Phoenix, February 1987.

[15] James Bieman, *Deriving Measures of Software Reuse in Object Oriented Systems*, Technical Report CS-91-112, Department of Computer Science, Colorado State University, July 1991.