The extension of software metrics in object-oriented development

M. Hamza, B. Lees, C. Irgens

Department of Computing and Information Systems, University of Paisley, Paisley, PA1 2BE, UK

Abstract

Whilst some traditional software metrics are applicable to object-oriented technology, these need to be supplemented by additional metrics directed at the particular features of object-oriented software. An investigation is described in which, having identified a set of software metrics, relating to object-oriented software development, the aim is to apply these metrics using Quality Function Deployment (QFD), supported by Case Based Reasoning (CBR), in different software cases, thus incrementally building up, over time in a case library, a collection of different cases, relating to various functional characteristics of developed software. It is believed that CBR may be effective in establishing the relationships between the criteria and the metrics in new software development cases. Metrics for assessing object-oriented systems are introduced. The combination of QFD and CBR methods in providing support during the formulation of customer requirements and in the assurance of software quality, is explored through a proposed model, establishing a relationship between software quality metrics and the software quality criteria of QFD.

1. Introduction

There are various problems associated with the production of quality software: software often does not meet size and quality expectations; it is often delivered late, incomplete, or over budget; it may not meet usability requirements. Quality software can be rapidly produced with existing reusable software components provided that the software components are of high quality. Object-oriented software engineering (OOSE) provides new solutions to existing problems in the software industry; in particular, the reuse of all forms of descriptions that are the result of system development. OOSE can be applied to
the development of technical and administrative systems.

With an object-oriented approach, the potential exists for the production of high quality software. For example, it has been shown [1] that an object-oriented design can lead to a better system architecture, and the object-oriented programming language enforces a disciplined coding style.

Quality measurement is usually expressed in terms of metrics. A software metric is a measurable property which is an indicator of one or more of the quality criteria that are to be measured. In this paper, the software metrics used in traditional development methods and how they might be extended to object-oriented development, are discussed.

A basic premise in this research is that the application of Quality Function Deployment (QFD) may be effective in mapping user requirements into the desired software quality characteristics. Problems in such application of QFD are the determination of the software criteria and the establishment of a relationship between these criteria and software metrics.

2 Software Metrics

The study of software metrics has attracted increasing amounts of attention recently. Various ESPRIT projects have been aimed at encouraging the European software industry to adopt a more quantitative approach to software development, in the belief that this will lead to better software.

Different authors have taken different approaches to metrics. The two pioneers of hierarchical models, McCall [2] and Boehm [3], adopted different approaches to metrication.

Boehm advocated methods based upon checklists requiring 'yes/no' answers. For example, structuredness is measured by questions such as: Have the rules of transfer of control between modules been followed? Are modules limited in size? Do all modules have only one exit point? Do all modules have only one entry point? A well structured program will produce positive answers to such questions.

McCall’s approach is more quantitative, using scores derived from equations such as:

\[ \text{McCall’s structuredness metric} = \frac{n_{o1}}{n_{tot}} \]

where: \( n_{o1} \) = no. of modules containing one or zero exit points only and \( n_{tot} \) = total number of modules.

Almost ten years later, Watts [4] suggested seven criteria of 'goodness' for a
software metric. These criteria are: objectivity, reusability, validity, standardisation, comparability, economy, usefulness.

Watts cited 40 metrics from the software engineering research literature. However, these metrics are not evenly distributed among the criteria. Although the objective measurement of quality through the use of metrics within a hierarchical framework is an attractive one, many problems arose with these metrics. These problems are:

- they cannot be validated,
- they are not generally objective,
- quality is a relative, not an absolute quantity,
- they depend upon a small set of measurable properties,
- they do not measure the complete set of quality criteria,
- the metrics measure more than one criterion with the result that conflicts may sometimes ensue.

Software metrics may be divided into process-related metrics and product-related metrics, the latter of which may be divided into goal, syntax and execution-based metrics.

*Goal-based metrics* compare the requirements of the software with its actual operation; they assess how well software achieves its functional requirements.

*Syntax-based metrics* examine source code and produce statistics describing how well it is implemented.

*Execution-based metrics* measure the runtime characteristics of the software; these metrics are facts about the software’s execution behaviour such as performance and fault tolerance.

Clearly, goal-based metrics are not independent of either syntax or execution-based metrics. Syntax and execution-based metrics are often used to measure how well goals have been achieved.

### 3 Extending Software Metrics in Object-Oriented Development

#### 3.1 Object-Oriented Methods

Existing methods for system development can basically be divided into function/data methods, and object-oriented methods. By function/data methods we mean those methods in which function and/or data are more or less separated, and where the system is typically broken down into its functional components, and data is transferred between these functional components. An emerging trend in software development is the increasing use of object-oriented methods, which offer the potential advantages of, *inter alia*, higher quality reusable code, easier maintenance, more extensible systems and improved
system architecture. Object-oriented methods view function and data as highly integrated. A problem facing system development using a function/data method is that the system becomes difficult to maintain causing instability of the development process, as any modifications will cause effects in other areas. Another problem is that the system is not structured from the model taken from the requirement specification. This leads to difficulty in modification and thus decreases reusability. It has been found that object-oriented methods could be a solution to these problems, by providing better support for the software designer’s natural way of thinking.

Object-oriented technology encompasses not only object-oriented programming systems (OOPS) but also other object-oriented aspects such as user interface, analysis, design and database management systems. In object-oriented development, an object model is built to represent a problem domain (analysis) and its solution domain (design). There, are four major elements of an object model: abstraction, encapsulation, modularity and hierarchy. Abstraction arises from a recognition of similarities between objects and processes in the real world. Encapsulation refers to the bundling of data and methods together within an object so that the data can be accessed only through the object’s own methods. Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules. Hierarchy can be either aggregation ("a part of") or a generalisation/specialization ("a kind of") relationship.

The advantages offered by object-oriented programming include: classification of data resources in class/subclass relationships, code reuse through inheritance & polymorphism, maintainability, open architecture, rapid prototyping, and generalisation.

3.2 Metrics for Object-Oriented Development
Traditional software metrics may to some extent be used also in object-oriented software. However, some traditional metrics may be of lesser importance in object-oriented software; therefore the introduction of some new software metrics is essential. The shortage of existing metrics and the need for new metrics especially designed for object oriented development have been identified by a number of authors. Tegarden et.al. [5] and Bilow [6], have called for theoretical rigour in the design of object-oriented metrics. The objective is to propose metrics which would be practically applied while strongly rooted in theory. Some initial proposals for such metrics are set out by Morris [7], although they are not tested. Lierberherr [8] presents a more formal attempt at defining the rules of correct object oriented programming style, built on concepts used in traditional programming, like coupling and cohesion. Moreau [9] suggests three metrics for object-oriented graphical information systems. Lake [10] prescribes metrics for measurement of inheritance in C++ environments, and has gathered data from an experimental system using an automated tool. Chidamber and Kemere developed six design metrics and then
analytically evaluated against Weyuker's proposed set of measurement principles [12]. Pfleeger [13] also suggests the need for new measures, and uses simple counts of objects and methods to develop and test a cost estimation model for object-oriented development. Lee [14] presents a set of complexity metrics for object-oriented systems based on information flow models. The proposed metrics are examined using Weyuker's method [15]. Chung describes a graph-theoretic complexity metric and illustrates how it can be used to manage and control object-oriented software complexity.

Process related metrics which may usefully be collected in object-oriented development include:

- **total development time;** which is dependent upon the programmers and their experiences, and so can not be an absolute measure,
- **development time for each process and subprocess;** which depends on the nature and function of the process and subprocess,
- **time spent in modifying existing components to meet new requirements;** which may be relatively low in the case of good quality object-oriented software, as one of the main advantages of object-oriented methods is reusability; this metric is particularly meaningful in object-oriented development,
- **number of different kinds of faults found during reviews;** the smaller the number of faults, the higher the quality of the system; however, this number is also dependent on the method of detection of faults; this is a metric for both traditional and object-oriented development.
- **cost of quality assurance;** this is the cost to achieve quality, usually measured in man-hours; the objective is always to decrease cost as much as possible.
- **cost of introducing new development processes and tools;** this is a very difficult measure to assess as it is dependent on the nature and type of the application and the development processes and tools available; it is also dependent on the experiences of the staff developing these processes and tools.

Traditional product related metrics may to some extent be used also in object-oriented software. However, the most common metric, lines of code, is not, in itself, a particularly revealing measure for object-oriented software. Another traditional metric which can be used in object-oriented software is McCabe's cyclomatic complexity [16], which measures the complexity through counting the connections and nodes of the graph representing the program. Although it still could be used in object-oriented software, McCabe number is of less interest due to polymorphism.

Product-related metrics may be divided into metrics of code and metrics to measure the quality of design and analysis. Metrics of code are more applicable; however it is interesting to collect metrics to measure the quality of design and analysis which could be of great use for project management and
control. Some metrics of code that are more appropriate for object-oriented software are:

**Number of classes.** Classes are the heart of object-oriented development. Classes and libraries can be considered as project guidelines for complexity, size and comprehension. A class may maintain several quality metric variables such as number of operations, lines of code and McCabe cyclomatic number. Classes are divided into reused classes and newly developed classes. With object-oriented software the objective is to have a greater number of reused classes to achieve reusability, one of the main advantages of object-oriented development. Reusing software is important, so it is probable that large class libraries will exist as common resources to the software engineering community. There is a great need to integrate specifications and tests within the software components themselves. Another potentially useful metric is the number of classes on which a specific class is dependent; this is a metric which indicates the degree of multiple inheritance. Inheritance is measured by the number of immediate subclasses subordinate to a class in the class hierarchy. The greater the number of these subclasses, the greater the potential for class reuse.

**Number of stimuli (messages).** In the object-oriented approach, objects communicate with each other primarily through message passing. A message can cause an object to behave in a particular manner by invoking a particular operation. An operation can be viewed as a definition of a response to possible messages. A response set for a class of objects is defined by the set of all operations that can be invoked in response to a message to an object of that class. It is clear that if a large number of operations can be invoked in response to a message, the testing and debugging of the class becomes more complicated and that the larger the number of operations that can be invoked from a class, the greater the complexity of the class.

**Number, width and height of inheritance hierarchies.** The width of the inheritance hierarchy means the number of immediate descendants of the class; the height of the inheritance is the depth of the class in the inheritance tree. However, in cases involving multiple inheritance, the height is the maximum length from the node to the root of the inheritance tree. It indicates the extent to which the class is influenced by the properties of its ancestors. The width of the inheritance indicates the potential impact on descendants. Both the width and height of inheritance indicate the genealogy of a class. Increased height of inheritance causes a greater number of operations to inherit, making it more complex to predict the behaviour of the class and thus causing greater complexity.

**Number of operations.** The number of operations per class is considered an important metric since it is one of the indicators of complexity. The complexity of the operation in a class is directly proportional to the number of operations
in that class. The larger the number of operations in a class, the greater the effect on the subclasses, as they will inherit all the operations defined in the class. Also, classes with a larger number of operations are more likely to be application specific, decreasing reusability. Operations in a class may be divided into similar operations and dissimilar operations (the degree of similarity for two operations in a class being defined as the intersection of the sets of instance variables used by the two operations). The larger the number of similar operations, the more cohesive the class, which is consistent with the traditional notions of cohesion that measure the inter-relationships between portions of the program. Lack of cohesion increases complexity and also implies that classes may be split into two or more sub-classes.

Examples of metrics to measure the quality of design and analysis are:

**Number of objects.** An object-oriented model consists of a number of objects, which are delimited parts of the modelled system. All information in an object-oriented system is stored within its objects and can only be manipulated when the objects are ordered to perform operations. Objects having the same characteristics are grouped in a class. Increasing the number of objects for a class leads to a higher complexity.

**Number of blocks.** The blocks, into which the source code is grouped, communicate by sending stimuli to each other. As the number of blocks increases, the complexity of the system increases. The number of classes per block and consequently the number of operations per block are important metrics that affect the complexity.

To manage the complexity of large systems, it is essential to divide the system into subsystems. Each subsystem contains a number of blocks so the number of subsystems may play a role as a software quality metric.

The metrics mentioned within this section are divided into metrics of code and metrics to measure the quality of analysis and design; however, with object-oriented methods, there is often much less distinction between the traditional separate areas of 'analysis', 'design' and 'code'. New more iterative development processes (e.g. 'spiral' and 'cyclic') are considered to replace the traditional waterfall model.

### 4 Quality Function Deployment

Customer satisfaction is ultimately the objective of any software developer. Quality may be defined as 'fitness for purpose' which expresses a degree of close correlation with the customer's requirements.

The software reusability argument may be viewed along the same lines as in the case of the manufacture of other, physical objects, with the resulting benefits
of standard components and assemblies. The implications of this are that (i) there is a need to be able to define the likely behaviour of such standard, reused components of software, (ii) it is necessary to classify fault sets and types which can be used to characterise the functional behaviour of software modules. Existing work makes attempts at such software fault classification.

Assuming that fault classification is possible, it remains to be able to show that certain types of software modules have a tendency to exhibit certain faults and to identify the conditions under which this may happen. Any practically useful metrics must rely upon the accumulated historic knowledge about reusable software modules which are available to the software developer. In the case of mechanical or electronic reusable standard parts and assemblies, a certain degree of change and alteration is very often needed for each instance of reuse. This is where the inheritance hierarchy is viewed as a potential method in determining the degree of similarity between a module of which we have functional knowledge and a new instance of a similar module. The inheritance model should include both number and type of operations. The reason for the inclusion of type is that fitness for purpose is to a large extent expressed by the overall software system’s emerging properties which in turn is dependent upon the permutations of primitive functions which are brought into the system by its constituent modules. The implication of this is that the new metrics must include the ability to classify and measure these emerging properties as well as the functional properties supplied directly by the software components.

It is necessary to establish a mechanism which can advice the software designer about quality conflicts. It is envisaged that the Quality Function Deployment (QFD) methodology can provide a suitable basis for this. Quality Function Deployment (QFD) is a highly structured method which aims at ensuring that the customer’s voice is taken into account from the earliest point of the product development life-cycle [17]. In terms of software development this means the early design stages when the software functional concepts are determined. QFD supports the customer’s priorities during this phase so that conflicting design priorities do not overshadow the customer requirements. QFD provides a set of techniques which rely upon a small number of simple information structures, two of which are of particular importance:

- A planning matrix which represents relationships between customer requirements in customer terms and product control characteristics.
- A deployment matrix which represents the greater details of the planning matrix so that critical component characteristics can be identified.

The QFD methodology can thus assist the designer in identifying critical customer requirements and associated critical product components. Thus, a need has been established for:

- a library of reusable software modules with their fault/success
characteristics in terms of fault classifications and instances, (fault-cases),
• design specifications in terms of well established hierarchical methods (OO),
• software metrics which may be called upon during the design phase to predict fault profile; the user must thereafter interpret this in the light of the customer requirements specification in order to establish the fitness for purpose,
• QFD based methods which can applied to the design in order to establish the what-how conflicts.

5 Artificial Intelligence Support

The application of artificial intelligence offers much potential for enhancing the support for software development. One strategy is to apply knowledge-based methods; however, the variant nature of software development makes it difficult to identify appropriate 'deep' knowledge expressible in the form of production rules. In other words there does not exist an appropriate model of software development which may be represented by a collection of production rules.

5.1 Case Based Reasoning

In solving problems for which there does not exist a model of problem solving one is forced to adopt an alternative approach. Although expertise involves knowledge of the principles to be applied in formulating problem solutions, it also involves past experience; it is important to remember what plans have worked in the past, and which have been unsuccessful. It may not always be necessary to proceed from first principles. Consider, as an example, the task of baking a nut cake. If one already knew how to bake a fruit cake, it would be a simple matter simply to follow the same procedure, but with the ingredients including nuts rather than fruit. This is essentially the approach of case-based reasoning (CBR) [18], which enables new problems to be solved based on the reuse and adaptation of past captured knowledge, which is represented in the form of structured cases. A case, in general, is a problem solving situation and its identifiable features, which has been previously encountered, together with its solution. A collection of such cases constitutes a case library; the cases are stored in the case library, suitably indexed to facilitate future matching and retrieval.

A CBR system then operates as follows. When a new problem solving situation is met, an attempt is made to match the characteristic features of the new situation with those of cases, held in the case library, which have been met on earlier occasions. If a suitable match is found, the matching case is retrieved and may be reused to solve the problem at hand, after perhaps adapting it to fit the new situation. If an existing case involves a significant amount of adaptation, it becomes an additional case, which is then included in the case library. Thus the system can be said to learn through the identification and modification of past problem solving experiences to fit the needs of new situations.
5.2 Application to Software Metrics

The case based reasoning approach, described above, may be applied to support a QFD methodology. Essentially what is required is the construction and maintenance of a case library that holds past software development histories, in the form of cases, and which comprises reusable objects and components, together with additional features which reflect the specifications and required functional characteristics of the software modules and their fault sets. The main purpose of the CBR system is to ensure the 'fitness for purpose' of a software module. A major aspect of future research on this will be the identification of an appropriate set of features which may be used to describe the essence, performance, metrics and quality characteristics relating to each case, and which will facilitate an indexing mechanism that will allow new software functional and quality requirements to be matched against existing cases.

6 Conclusion

Work which has been carried out in the engineering domain confirms the feasibility of CBR as a support tool for the designer of new products in a variant type design environment. The designer of engineering products translates the customer requirements specifications (CRS) into an object hierarchy of standard assemblies, parts and form-features which are instanced specifically to contribute primitive function(s) or part of function(s) to the overall emerging functional properties of the 'new' product. Accumulated knowledge of the functional behaviour of these components is a vital support to the designer. CBR has been used successfully to identify similar past products and to provide advice upon the instancing parameters of the standard parts, assemblies and form-features to achieve the desired functional characteristics.

The analogy with manufacturing of engineering products is quite clear. Ultimately the designer and the subsequent manufacturing processes must ensure that the final product conforms to the customer requirements specifications (CRS). Given the above framework of a QFD based mapping between the customer’s priorities and the product control characteristics, and the historic fault characteristics associated with each software module, cases can be constructed which are based on the OO hierarchy and fault profile of the standard software modules. The user would typically look for case similarity which is based on:

- similarity with respect to inheritance in the OO hierarchy,
- similarity with respect to CRS.

The QFD based information can then be used to effect design advice which is based on the most similar historic case and the known fault profile of its standard software modules.
References


15. Weyuker, E. *op.cit.*

