A Measure of Testing Effort

John McGregor and S. Srinivas
Clemson University

A Measure of Testing Effort

John D McGregor
Satyaprasad Srinivas
Dept. of Computer Science
Clemson University
Clemson, SC 29634
johnmc@cs.clemson.edu
satya@cs.clemson.edu

Abstract Accurate estimates of the time and resources needed for a project are difficult to achieve. Numerous metrics have been proposed and a few have proved reliable in making these estimates. With the increased emphasis on quality and testing, estimates of the amount of effort required to test a product are a necessary part of any complete project estimate.

Estimates of the effort to test object-oriented components and systems are particularly important because these components are often added to repositories to be used many times. The amount of effort required to test the component is related to its complexity. We consider several measures of method and class complexity and relate them to testability.

The main focus of this research is to estimate the effort that is needed to test a class, as early as possible in the development process. We investigate the testability of a method in a class and indirectly estimate the effort that is needed to test a class. We define a concept termed the visibility component of a method. It is a measure of the accessibility of the information that must be inspected to evaluate the correctness of the execution of a method. We show that the testability of the method is a function of its visiblity component.

Keywords: Testability, Data State Collapse, Syntactic/Semantic complexity.

1 Introduction

The ease with which a software component can be tested is an important element in determining the amount of effort required to build that component. The aggregation of the testing efforts for all of the classes has a major impact on the overall effort required to build a software system. The goals of our metrics research have been expanded from quantifying the construction process by considering the design complexity of individual classes[20][12][1]to quantifying project attributes by using metrics[17]. In this paper we present early results from our work on estimating the amount of effort that will be required to test an object-oriented software system.

Software metrics are quantitative values that provide information for decision making. Metrics are used to quantify attributes of the product being developed and of the process used to construct the product. These measurements can be used to estimate the effort, size and cost requirements for the development of a software system.

Given an iterative, incremental process model, repeated applications of a metric can provide an opportunity to recognize trends that can be used to predict the progress, or lack thereof, of the project. This technique provides an opportunity to identify problems early in the developmental cycle. Project attributes are therefore observed/estimated as early as in the analysis phase, even though there is a risk of reduced accuracy[15].

The main focus of this research has been to estimate, as early as possible in the development process, the effort that is needed to test a class. We have investigated the testability of a method in a class and use that measure to indirectly estimate the effort that is needed to test a class. We introduce a concept termed the *visibility component* of a method. It is a measure of the accessibility of information that must be inspected to evaluate the correctness of the execution of a method. We show that the testability of the method is a function of its *visiblity component*. The estimation of a class's testability given the testability of its methods is also presented. The metric appropriately handles features in object-oriented languages such as the visibility of inherited attributes and other language features such as exception handling.

This paper makes four main contributions:

- This work extends existing research on the testability of procedural software to object-oriented software,
- It considers the levels of information hiding within an object-oriented program, such as protected and private areas, and how these contribute to hiding faults,
- It considers the levels of scoping such as attributes with object or class level scoping, and
- It addresses such modern programming features as exception handling.

The remainder of the paper is organized as follows. In the next section, we review the background and related work necessary to understand the metric. In section 3, we define a set of basic terms used in the description of the metric and define the metric itself. In section 5, we consider our metric in the context of criteria that have been widely used for evaluating complexity metrics and we present an experimental validation of the metric as well. In section 4, we present a set of examples to illustrate the computation of the metric. Finally in section 6, we present conclusions and directions for future work.

2 Related Work

The early work on metrics for object-oriented software includes an OOPSLA paper by Chidamber and Kemerer[5] and a book by Lorenz[13]. The topic of testing metrics for the object oriented paradigm has emerged only as a recent topic. Research in the area of testability metrics is meager and is limited to the procedural paradigm. Several code models/tools have been proposed to determine the testability of procedural code, but these cannot be directly applied to object-oriented programs. These techniques fail to recognize the associations among the methods in a class and they fail to account for the state of the objects created from the class.

Among the measures in the procedural paradigm that are of interest is the Domain Range Ratio metric(DRR) proposed by Voas[24]. This metric is a simple measure that calculates the testability of a procedure based on the information that is flowing into and out of the a procedure. Although DRR establishes an upper bound on the testability of the code, it does not help locate the particular piece of code where an internal data state collapse occurs. Voas has also developed the PIE[23] technique to help identify locations that have a probability of hiding faults. The technique uses a set of semantic and syntactic mutants to help identify locations having a high probability of internal data state collapse[23].

A major goal that has animated our research has been to estimate the effort required for testing a class at a very **early** stage of the development life cycle. Given the fact that the process of testing involves the actual code, early calculation of the metric limits the testing-related attributes that are available for the computation. So our metric calculates an upper bound on the testability of a class. We also feel that the PIE model, although very demonstrative, is computationally intensive and impractical to apply using only specifications. The use of syntactic and semantic mutation testing are also factors influencing our decision.

We have designed the metric to use the design information in the class specifications. The attributes that are available from the specifications include the data attributes of the class and the method names along with their respective signatures. The objects that are returned from each of the methods are also considered.

The metric focuses on the information content in the class specification rather than on the actual interaction of the methods and their messages. Although one might argue that this limits the accuracy of the forecast, it is our opinion that the metric strongly represents areas in the class deserving additional attention. This can be used to estimate and schedule the resource allocation during the testing phase of the lifecycle.

We have *validated* the metric against the formal list of desiderata proposed by Weyuker[25]. Although the list has its drawbacks, the discussion provides one form of theoretical validation of the metric. Additional validation work has been carried out and will be discussed in setion 5.

We have also developed a testing architecture that improves the testability of the software by improving the visibility of the internal attributes of the objects[19]. This organizing principle reduces the amount of effort required to test by requiring a test class for each production component and by using language mechanisms to overcome the information hiding built into the design.

3 Metric Definition

The life of a program is a series of *invocations*[3]. During each invocation, the program interacts with the rest of the world, receiving inputs and, as a result, producing outputs. The specification of a program can be used to unambiguously decide whether the program's output is correct, given the input that the program received.

A program may behave non-deterministically, i.e. its results may differ between executions with identical inputs. A program is said to be **correct** on an input i, iff the output produced on that input is compliant with the specification of that program. The program is categorized as being **faulty** otherwise. The set of inputs that cause failures forms the failure Domain DF which is a subset of the input domain D[24].

Testability is the prediction of a method's ability to reveal faults in its implementation given a particular input distribution[3]. Testability is dependent on the structure/specification and the input profile of the method. At a very early stage of the developmental cycle, the possible set of inputs and outputs to the method, together with the input profile can be used to determine the testability of a method.

A method specification that maps many inputs to a single output indicates some *information loss* during execution. This indicates that more information flowed into the method than flowed out. A one-to-one specification indicates that no information loss has occured. Unlike procedural systems, a method in an object-oriented program may actually have more information flow out than flowed in because it may access an attribute and return it or some subset of it. Methods that implement many-to-one specifications may produce information that is stored in its internal states and that may affect the results of program execution much later in the computation. It is our hypothesis that this uncertainty affects the amount of testing that must be performed in order to achieve a specified level of test coverage.

3.1 Terminology

Generally, the terminology used to define and discuss the proposed metric is the same as used by the C++ programming community to discuss object-oriented concepts. One exception to this is the use of "method" as opposed to "member function". The decision to employ C++ terminology does not indicate that the metrics are applicable only to C++ designs. Rather it is indicative of the fact that C++ provides a sufficiently broad range of constructs to support discussion of other models[1].

3.2 Basic Terms

- **Explicit parameter** - An object that is explicitly declared in the method signature, see Figure 1.
- **Implict parameter** - An object that is visible to the method by virtue of the fact that it is declared as a class or instance variable in the class declaration. Not all of the data attributes in the class are visible to all the methods in the class. Syntactically all the protected and the public attributes are visible to a derived class during inheritance. Therefore the implicit parameters for a derived class includes the attributes of the base class that are visible to the derived class. A detailed set of counting rules for parameters are presented in section 3.7.
- **Constant Object** - A constant object is one that does **not** change its state during the execution of the method. These consist of objects that are made state invariant syntactically by definition, such as

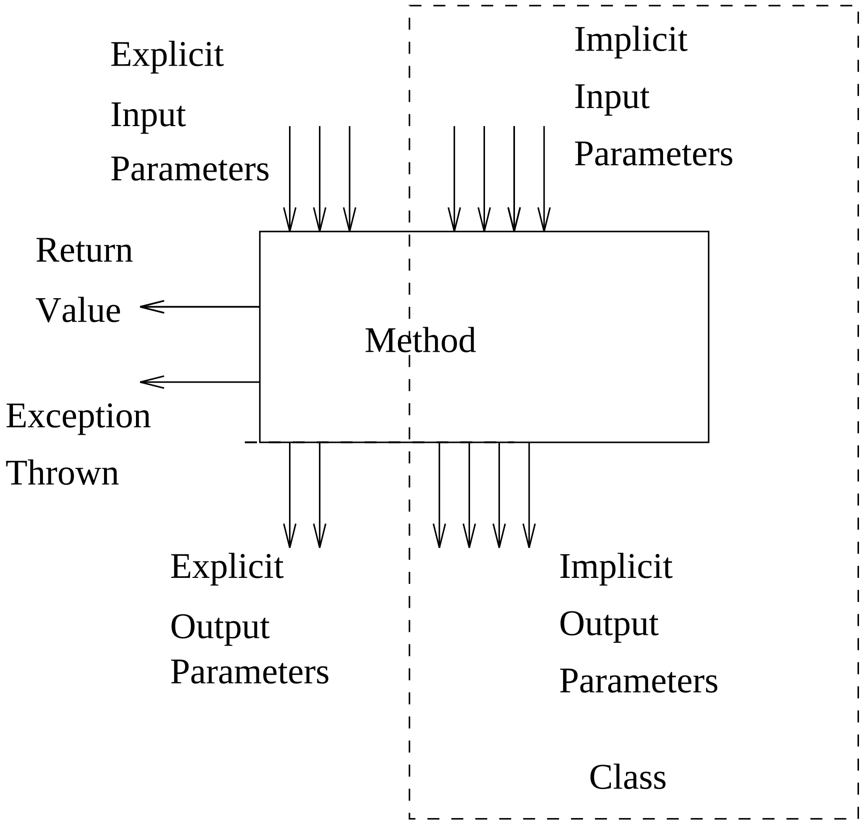

Figure 1: Method Inputs and Outputs

const objects in C++ or *final* objects in Java.

- **Constant method** - A constant method is one that does **not** modify any objects. This includes implicit and explicit parameters.

An explicit parameter can be either a **modifiable** or an **inspectable** object with respect to a specific method. The classification is determined by whether the object changes its state during the execution of the method. For the implementation of the metric being presented in this paper, all the implicit parameters are assumed to be **modifiable** objects. This metric is applied to the specification of a class which will not provide sufficient information to determine which of the implicit parameters might be modified. This assumption will make our metric value an upper bound on the effort estimation.

The attributes that are available for calculating the testability metric are the attributes defined in the class, the parameters declared in the method signatures and the attributes that are visible to the methods due to inheritance and aggregation.

3.3 Visibility Component of a Method

The Domain/Range Ratio metric proposed by Voas [24] is defined as the ratio of the cardinality of the possible inputs to the range of the possible outputs.

This metric is not directly applicable to object-oriented programs because the number of explicit parameters that are being passed to a method is not a real indication of the loss content in the method. This is because object-oriented languages discourage procedural autonomy and instead organize methods into a collection of operations that share the object state through non-local variables.

We therefore include the attributes defined in a class to be the **implicit parameters** to all the methods defined in the class. We take this conservative approch at the design phase due to lack of information about the actual set of objects that change their states at runtime. A reformulation of the metric for the coding phase would include only those objects that have been utilized. This redefinition would provide a greater accuracy of the metric.

Adapting the above metric, we introduce a concept termed the **visibility component (VC)** of a method. It defines the accessibility of objects that may have changed state during the execution of a method. We assume that the implicit and explicit parameters have equal potential to affect the object's current state.

The definition of the **visiblity component** ζ is defined as

$$\zeta = \frac{cardinality\_of\_possible\_outputs}{cardinality\_of\_possible\_inputs} \quad (1)$$

The following are considered to be *"Inputs"* in the calculation of the above metric.

- The number of explicit parameters in the method signature, if any.
- The number of implict parameters declared in the class.

The following are considered to be *"outputs"* in the calculation of the above metric.

- The explicit reference parameters in the method signature, if any.
- The implicit parameters, object attributes, defined in the class.
- The return value of the method, if any.
- Any exceptions thrown by the method.

Syntactically, C++ allows the definition in the class header of the exception objects that are thrown by a method. We therefore propose that the visibility component of that method includes the visiblity component of its exception objects too. The inherent assumption is that the exception objects that are thrown by the method include information about the state of the object and the correctness of execution just as do the implicit and explicit parameters to the method.

Future extensions of this work may include the calculation of information loss content based on the individual complexities of the thrown objects.

3.4 Testability of a Method

The **testability** of a method, η is dependent on the visiblity component of the method. We define the testability of a method in a class to be:

$$\eta = constant * (\zeta) \quad (2)$$

To make the testability estimate of a method a true indicator of the effort needed to test the method, we take the constant to be the number of unique objects that participate in the calculation of the metric. That is, we count the objects used as explicit input parameters, implicit input parameters, return objects and exception objects. We do not include the implicit output objects nor the pass by reference, explicit parameters in the count. This constant is a reasonable choice because its value indicates the maximum number of objects whose state may have been changed incorrectly.

3.5 Testability of a Class

We base our definition of the testability of a class on the testability values, η, of all the methods in the class. We define the testability of a class, θ, to be the **minimum** of the testabilities of the methods in the class.

$$\theta = min(\eta) \quad (3)$$

Since the testing effort is *maximum* in the method that hides the most information, we have selected the lower bound of the testabilities of all the methods, to be the testability of the class.

We do not include **constant objects and methods** in our complexity calculations. The reason is that the metric considers the set of objects that have the potential to change their state, possibly erroneously, during the execution of the method. Since constant objects and methods are **syntactically state invariant**, at least during the execution of the method, this assumption makes the metric more sensitive to those objects that may have taken on erroneous values that will be propagated on to other methods, and possibly other objects.

3.6 Inheritance

McGregor and Dyer[18] considered the inheritance relationship between two classes as a mapping that carries out certain transforms on attributes as they are inherited from one class to another. Each object-oriented language has its own set of transforms that may be applied.

This means that the attributes that are visible to the methods of the derived class are a subset of the attributes from the base class. The visibility component calculation therefore has to be modified to handle this.

In the case of C++, all of the **protected** and **public** attributes of the base class are visible to the derived class. This increases the set of data objects that are visible to the methods in the derived class. Other languages have different visibility and scoping rules, like the implicit private area in Smalltalk objects, and encapsulation mechanisms such as packages in Java. We therefore consider the set of implicit parameters in the derived class to include all the objects from the base class that are visible in the derived class.

$$implicit\_parameters = base\_class\_objects \bigcup derived\_class\_objects$$

These modifications also make the metric sensitive to the level of inheritance in the class hierarchy. Since the implicit parameter calculation is influenced only by the visibility level (i.e private, public or protected) of the objects in the base class rather than the class of the object, the metric is monotonically increasing. To address this issue, we have assumed that the count of the implicit parameters of a class at any level in the inheritance graph **includes** the protected/public attributes of all the classes that lie above it in the inheritance tree. If a class multiply inherits from many classes, the count of the implicit parameters that inherit from this class, increases substantially. This demonstrates that misuse of multiple inheritance could lead to potential problems in maintainance and testing.

The above counting rules also make the metric compatible with the DIT metric proposed by Chidamber[5]. The deeper a class in the hierarchy, the greater is the number of data objects it is likely to inherit, making it more complex to test.

3.7 Summary of Counting Rules

In this section we will summarize the set of counting rules that have been used for the calculation of the visibility component of a class.

- For a simple class without any inheritance, the set of implicit parameters is the set of all the private, public and protected data attributes in the class. The set of possible objects that can change state is limited to the set that is visible to the methods.

- For a class that inherits from another class, the set of implicit parameters is the set of all the data attributes that are defined in the class plus the set of implicit parameters from the base class except for any attributes that are private to the base class. The same counting rules are used while calculating the set of implicit parameters for a class that multiply inherits from different classes.

- Constant objects and methods are not considered in the complexity calculations. The metric is designed to estimate the set of objects that have the potential to change states.

- The Visibility Component of a method does not change as it is inherited into subclasses. Even though new attributes may be introduced, the method does not contain code to access these attributes and cannot modify them. Hence, the method treats these new attributes as constant objects.

- The Visibility Component will change monotonically as the method is redefined in subclasses. The signature of the method cannot be changed. Thus the only change to the computation of the metric is in the number of attributes that are implicit to the method. Since the metric calculationis solely based on the signature, changes in the implementation of the method do not influence the value. The output/input ratio will not change, but the testability will increase as the constant multiplier changes.

4 Example Calculations

The testability metric proposed in this paper can be used to estimate the effort that is required in the testing phase of the developmental cycle. Typically the parameters that are considered during effort estimation, include data due to metrication , maturity of the software developmental process within the company and the actual resources that are available for the project.

We propose that the effort that is needed to test a class hierarchy is the sum of all the resources expended in the testing phase of the project life cycle. One of the important factors that determine the effectiveness of testing cycle is the **coverage** of the testcases i.e the reach of the testcases in determining/verifying the set of objects that have changed states during execution. It is of statistical significance that the testing coverage is proportional to the number of test cases defined in the test plan. It is often the case that an increased activity of state changes within a method often correlates to an increase in the number of testcases to validate that method.

The proposed metric estimates, although conservatively, the number of objects that have the ability to change their states in a method. It is therefore logical to investigate the correlation between static and dynamic measures by measuring the actual objects that have changed states in a method.

5 Validation

5.1 Theoretical Validation

In this section we will continue to validate the measure theoretically, first by considering the response of the measure to the features specific to object-oriented systems. Then, in the rest of the section, we will consider a set of axiomatic properties which should be satisfied by any complexity measure. The complexity metric is then validated against these properties.

Object-Oriented Features Object-oriented designs have certain features that have been found to reduce the complexity of the software. Below we present these features and explain the response of the measure in those situations.

1. *Inheritance*: Whenever a class has to be developed we may have a choice of inheriting from an existing class or creating one from scratch. The measure increase the complexity of the class indicating that testing is always involved proposition when there is a lot of reuse.

2. *Information hiding*: By encapsulating its attributes or properties, a class restricts access to its information. This is often leads to loss of information which results in a greater complexity measure for the class. As the measure is inversly proportional to the loss component, it is very sensitive to feature of object-orientation. It is also necessary to note that the measure of the metric is dependent on the design of the class. If the methods in the class have a high visibility component then the measure decreases.

3. *Weak Coupling*: Classes should be as independent of each other as the domain they represent permits. Unnecessary interaction with other classes is considered bad design. Coupling is indicated by objects used as attributes within a class and as parameters to methods. The metric rewards weak coupling since it produces a lower value for classes that have fewer input and output parameters.

4. *Composition*: When the internal mechanism of a class uses the services of an already existing class through instantiation, the complexity of the containing class is reduced. This is the second form of reuse typical in object-oriented programs (the

first being reuse through inheritance.) Composition is an example of coupling which has already been addressed above.

Axiomatic Properties Weyuker[25] listed a set of properties (necessary but not sufficient) that are desired of a complexity measure for procedural programs. Although these properties have been criticized[1] , Cherniavsky [4], they do provide a basis for some validation of complexity metrics short of using experimental data. As shown by Weyuker, with such properties it is possible to filter out measurements with undesirable properties.

We have not used all of the properties defined by Weyuker. These properties were defined for metrics intended to measure procedural programs and, as such, define some constraints that are not applicable to object-oriented software. In particular, assuming a single entry, single exit module does not address the object-oriented aggregation of methods into a class. Where there is an obvious mismatch between our goals and the properties we have omitted the property.

In the list of properties below, E, F and G represent classes and the complexity of E computed by an ideal measure is represented by $\mid E \mid$. Classes can be concatenated either by *inheritance* or *composition* (collaboration). When a class F inherits from another class E, we depict the resulting subclass as $F \uparrow E$ Also, when a class E collaborates with another class F, the collaborating class is shown by $E \rightarrow F$. Two classes are said to be *equivalent* if they have the same behaviour for the classes.

- Property 1: $(\exists E)$ $(\exists F)$ $(\mid E \mid \neq \mid F \mid)$.
There exists two classes E and F such that the complexity of E is not equal to the complexity of F. The property requires that a measure should not produce the same value for every class. With our measure we can always come up with two classes with two different complexity values. We can always construct a class F which has a different number of data attributes or methods with different signatures from those in class E, such that the Visibility Component is different.

- Property 2: *Let c be a nonnegative number. Then there are only finitely many classes of complexity c.*
A measure has to be sufficiently sensitive, and this property requires that a measure have a bounded range of values that it can assign. By the principle of number theory we can show that there is a finite combination of series whose sum is equal to $(\mid c \mid)$. Each series in the combination represents a particular set of attribute Visibilities. Since there is a finite number of distinct series, the number of classes with the Visibility Component, c is finite.

- Property 3: *There are distinct classes E and F such that,* $\mid E \mid = \mid F \mid$.
A measure that assigns a unique value to every class is not much of a measure. It would go against the principle of measurements which requires that the number of objects that can be measured be greater than range of the values of the measure. Our measure clearly satisfies this property, since the Visibility Component does not depend on the types of the attributes. Given a class E, we can create a new class with a different number of attributes and methods having high visiblities so that the new class F has the same complexity value.

- Property 4: $(\exists E)$ $(\exists F)$ $(E \equiv F$ *and* $\mid E \mid \neq \mid F \mid)$.
There exist classes E and F such that E is equivalent to F but the complexity of E is not equal to the complexity of F. There are different implementations for the same functionality. Since our metric does not depend on data types, it independent on class implementations. Therefore this property is satisfied by our metric.

- Property 5: $(\forall E)$ $(\forall F)$ $(\mid F \mid \leq \mid F \uparrow E \mid)$.
For all classes E and F, if F inherits from E then the complexity of $F \uparrow E$ is greater than or equal to the original complexity of F. Due to the inheritance, class F has access to all the public and protected data

[1] The major drawback of Weyuker's properties is that definition of complexity is given.

attributes of E. This increase in the implicit parameters of class F making the measure sensitive to inheritance.

Since our measure happens to satisfy all the above properties it can be considered to have passed a significant part of the theoretically validation process. The measure has been tested againt those axioms proposed by Weyuker that are applicable to both procedural and object-oriented software systems.

5.2 Experimental Validation

The most conclusive validation of the metric presented in this paper would be to collect data from actual development projects. Since the VC metric presented in this paper is intended to give an estimate of the effort required to test a class, a possible experiment would be to apply the metric to a set of classes, predict the amount of effort required to test each one and then to have each class tested by a developer. The actual effort required to test each could be obtained and the accuracy of the prediction could be measured.

Since such experiments are difficult to control and even more difficult to have approved, we propose an alternative approach in which the number of test cases required to achieve a specified level of test coverage is used to estimate the total testing effort. For our experiment we used **branch coverage**[2] as our coverage criteria. The criteria chosen is irrelevant provided that it is recognized as a valid, and likely, technique that might be used by developers on an actual project.

One criteria for the level of test coverage that is important is that the testing technique should utilize the same level of information as VC. The value of VC presented here is the *internal* visibility because it considers attributes that are not visible external to the object. This estimate would approximate the amount of effort that would be devoted to testing the individual methods. This internal view corresponds more closely to a structural testing approach, such as branch testing, than to a specification-based approach. In the conclusion section we will consider the *external* visibility of an object that corresponds to specification-based testing.

For our experiment, we chose classes from several sources including locally written code and widely used libraries. The VC was calculated for each method and each class. Each modifier method was examined and the number of branch test cases needed to achieve 100% branch coverage of each method was calculated.

A non-parametric correlation statistic, Spearman's rho, was used to determine the amount of agreement between the number of test cases required to give 100% branch coverage and the internal visibility. This comparison resulted in a correlation of .35. The significance of the correlation was investigated using the student's t distribution due to the sample size. The correlation was found to be significant at the .1 confidence level.

Initial comparisons were less successful than the present one because they included both accessor and modifier methods. Accessor methods confound the VC metric in that there is more information flowing out than flowing in. Accessor methods are also very simple structurally and usually have only one path; therefore, the testing effort is negligible. For this experiment and those reported below, accessor methods were excluded from the calculation.

Several other experiments have been conducted to experimentally validate the metric.

The testability of a method is directly related to the complexity of its structure[14]. Using the original set of classes, we computed the cyclomatic complexity for each method. Once again there was statistically significant agreement between VC and the cyclomatic complexity.

In a previous study[1], a set of design problems was presented to a group of developers experienced in object-oriented design. Figure 2 presents one set of design solutions for a problem in tracking the idle time for a set of processors. The developers were asked to rank three solutions to each design problem. These rankings were then compared to the rankings computed using a complexity metric, Permitted Interactions. In a subsequent study another metric, intended for use even earlier in the development process, was compared to these expert rankings. In each case, statistically significant agreement was found.

For the testability metric, a similar compar-

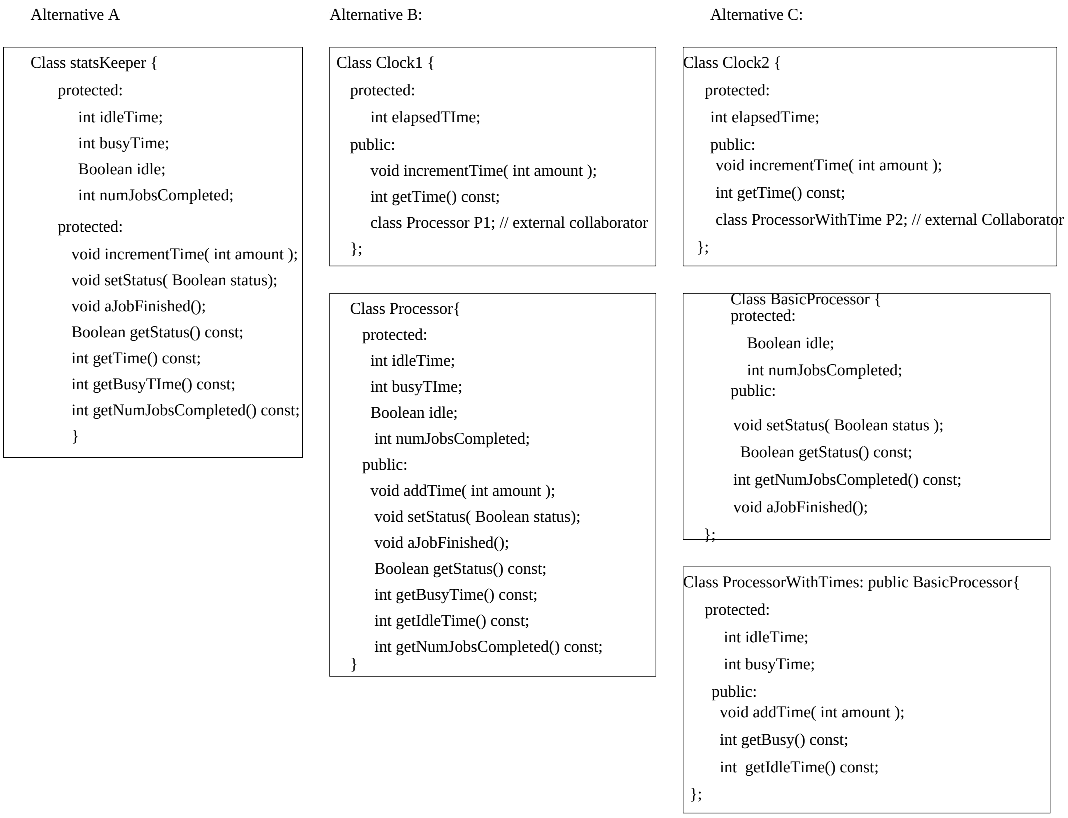

Figure 2: Example Solutions to a Design Problem

ison was made between the rankings by experts and the VC metric. Figure 3 shows the visibility calculations for the design alternatives. The results of the comparison are shown in table **??**. The VC metric agreed with the expert rankings in 14 of the 17 cases. In two of the three cases for which there was disagreement, previous studies[1, 12] had also shown some disagreement indicating that these examples may have alternatives that are too closely related.

5.2.1 Analysis

The relationship between VC and the number of branch test cases is weak; however, the combination of validations that we used, taken together, provide compelling evidence for this approach.

One reason for the weak relationship may be due to the lack of sensitivity of VC. A large number of the methods that were examined had ratios of 1. We partially addressed the lack of sensitivity issue by multiplying the ratio by the number of unique objects associated with a method. We will further discuss the sensitivity issue in the future work section of the paper. The lack of sensitivity is not surprising given the stage in the development process at which we wish to apply the metric. Early models, of necessity, contain less exact information than later models, such as code; however, estimates of project size and effort must be made as early as possible with the best information available.

A post hoc analysis revealed an interesting relationship. The agreement between measures for the base class and the internal visibility is very high (for one set of classes the correlation was 1.0). This agreement declines as classes

Alternative A

Class StatsKeeper

Name of the method	Visiblity Component
incre,entTime()	5/4 = 1.25
setStatus()	5/4=1.25
aJobFinished()	4/4=1

Testabality of Alternative A is: 1.25

Alternative B:

Class Processor:

Name of the method	Visiblity Component
addTime()	5/4 = 1.25
setStatus()	5/4 = 1.25
aJobFinished()	4/4 = 1.0

Class Clock1

Name of the method	Visibility Component
incrementTime	2/1 = 1

Testability of Alternative B is: 2.00

Alternative C

Class BasicProcessor

Name of the method	Visibility Component
setStatus()	3/2 = 1.5
aJobFinished()	2/2 = 1.0

Class ProcessorWithTimes

Name of the method	Visiblity Component
addTime()	5/4 = 1.25

Class Clock2

Name of the method	Visibility Component
incrementTime()	2/1 = 2

Testabality of Alternative C is: 2.00

Figure 3: Example Calculations for VC

lower in the hierarchy are added to the computation.

6 Conclusion

The proposed testability metric estimates the likelihood that an object will reveal its faults. The metric is sensitive to changes in the number of data attributes and the depth of the inheritance. By making the metric dependent on the information content of the class, as represented by the signatures of the methods, rather than the structure of the class, the metric may be used in the early phases of the development cycle.

Although the sensitivity of the metric is less than it would be if applied later in development when the code is available, it is our opinion that the metric provides a sufficient basis for estimating the resources and effort necessary for adequate testing of the code.

There are several possible extensions of this work. These include:

- **Sensitivity studies** - The metric as currently defined is not very sensitive. A large number of methods cluster around metric values of 0, 1 and infinity[2]. Possible redefinitions could include weights for the individual attributes of a class, such as its methods and attributes to reflect their participation in the state of the object.
- **External visibility** - The external visibility is analogous to the internal visibility for a method/class. The external visibility does not include the implicit attributes of the class. The relationship between this measure and a specification-based testing technique is being investigated.

[2]Infinity is the result when a method has no inputs but some outputs.

Comparison	Expert Pref.	PC Pref.	PI Pref.	VC Pref.	
RV2 vs RV3	RV3	RV3	RV3	RV3	$\surd$
RV1 vs RV3	RV3	RV3	RV3	RV3	$\surd$
ST1 vs ST2	ST2	ST2	ST2	ST2	$\surd$
ST2 vs ST3	ST3	ST3	ST3	ST3	$\surd$
ST1 vs ST3	ST3	ST3	ST3	ST3	$\surd$
R1 vs R2	R2	R1 or R2	R2	R1 or R2	$\surd$
S1 vs S2	S2	S2	S2	S1	$\times$
Q1 vs Q2	Q2	Q1	Q2	Q1	$\times$
Q2 vs Q3	Q2	Q2	Q2	Q2	$\surd$
Q1 vs Q3	Q1	Q1	Q1	Q1	$\surd$
Q4 vs Q5	Q4	Q5	Q4	Q4	$\surd$
Q5 vs Q2	Q2	Q5	Q5	Q2	$\surd$
Q4 vs Q2	Q4	Q2 or Q4	Q4	Q2	$\times$
IO1 vs IO2	IO2	IO1	IO2	IO2	$\surd$
IO2 vs IO3	IO2	IO2	IO2	IO2	$\surd$
IO1 vs IO3	IO3	IO1	IO1	IO3	$\surd$
TT1 vs TT2	TT2	TT1	TT1	TT1 or TT2	$\surd$

Table 1: Preferences by experts, PC,PI and VC

- **Variability of the measure** - The difference between accessor and modifier methods was obvious and treating them differently was easily justified. Other factors that affect the testability may be worthy of study. For example, in our data analysis, we noted informally that the visibility measure decreased as the depth of the class in the inheritance hierarchy increased. These observations may lead to better understanding of other less obvious relationships.

Although this metric has not been tested on a full scale software project, our preliminary results from applying the metric on a wide range of C++ classes have been successful both theoretically and experimentally. It has been defined in such a way that it will fit into our framework for iterative metric management[17]. In this framework an early measure, such as VC, is paired with measures that can only be applied later in the life cycle, such as Voas's information loss metric[24] or the simpler McCabe's Cyclomatic Complexity[14] metric. With this framework and automated metric collection tools, a manager can estimate early and measure exactly later in the life cycle, to provide a continuous picture of the project.

References

[1] D. H. Abbott, T. D. Korson, and J. D. McGregor. A proposed design complexity metric for object-oriented development. Technical Report TR 94-105, Clemson University, 1994.

[2] Boris Beizer. *Software Testing Techniques.* International Thompson Computer Press, second edition, 1990.

[3] Antonio Bertolino and Lorenzo Strigini. On the use of testability measures for dependability assessment. *IEEE Transactions on Software Engineering*, 22:97–108, 1996.

[4] J. C. Cherniavsky and C. H. Smith. On weyuker's axioms for software complexity measures. *IEEE Trans. on Software Eng.*, 17(6):636–638, June 1991.

[5] S. R. Chidamber and C. F. Kemerer. A metrics suite for object-oriented design.

IEEE Trans. on Software Eng., 20(6):476–493, June 1994.

[6] N. Fenton. Software measurement: A necessary scientific basis. *IEEE Trans. on Software Eng.*, 20(3):199–206, March 1994.

[7] N. E. Fenton. When a software measure is not a measure. *IEE Software Eng. J.*, 7(5):357–362, May 1992.

[8] Renato R Gonzalez. A unified metric of software complexity: Measuring productivity and value. *Journal of Systems Software.*, 29:17–37, 1995.

[9] Warren Harrison. An entropy-based measure of software complexity. *IEEE Trans. on Software Eng.*, 18(11):1025–1029, November 1992.

[10] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Trans. on Software Eng.*, 7(5):510–518, September 1981.

[11] Vernon. V. Chatman III. Change points: A proposal for software productivity measurement. *Journal of Systems Software.*, 31:71–91, September 1995.

[12] Sanjay Kamath and John D. McGregor. A measure of psychological complexity. Technical report, Dept. of Computer Science, Clemson University, 1995.

[13] Mark Lorenz and Jeff Kidd. *Object-Oriented Software Metrics.* Prentice-Hall, 1994.

[14] T. J. McCabe. A complexity measure. *IEEE Trans. on Software Engineering*, 2(4):308–320, December 1976.

[15] J. D. McGregor. Managing metrics in an iterative incremental development environment. Technical Report TR 94-110, Clemson University, 1994.

[16] J. D. McGregor and D. A. Sykes. *Object-oriented Software Development: Engineering Software for Reuse.* Van Nostrand-Reinhold, New York, NY, 1992.

[17] John D. McGregor. Managing metrics in an iterative environment. *Object Magazine*, 5(6):65 – 71, 1995.

[18] John. D. Mcgregor and Douglas M Dyer. A note on inheritance and state machine. *ACM SIGSOFT SOFTWARE ENGINEERING NOTES*, 18(4):61–69, October 1993.

[19] John D. McGregor and Anuradha Kare. A parallel architecture for component testing. In *Proceedings of the Ninth International Quality Week*, 1996.

[20] Neeraj Ojha and John D. McGregor. Object-oriented metrics for early system characterization: A crc card-based approach. Technical Report TR 94-107, Dept. of Computer Science, Clemson University, 1994.

[21] M. Shepperd. An evaluation of software product metrics. *Info. and Software Technology*, 30(3):177–188, April 1988.

[22] M. Shepperd and D. Ince. *Derivation and Validation of Software Metrics.* Clarendon Press, Oxford, 1993.

[23] Jeffery Voas, Larry Morell, and Keith Miller. Predicting where faults can hide from testing. *IEEE Software.*, pages 41–48, March 1991.

[24] Jeffery M. Voas and Keith W. Miller. Semantic metrics for software testabality. *Journal of Systems Software*, 20:207–216, 1993.

[25] E. J. Weyuker. Evaluating software complexity measures. *IEEE Trans. on Software Eng.*, 14(9):1357–1365, September 1988.