ABSTRACT

In this paper, a brief survey of object-oriented design considerations is discussed. A software design metric approach, known as Factor-Criteria-Measurement, is enhanced to adopt the use of Unified Modeling Language (UML) class diagrams to derive useful object-oriented design metrics. A structural method to present the metrics is also discussed.

The model-driven approach is thoroughly examined and the advantages and limitations are presented. An example based on software reusability is used to show the seamless nature of using UML class diagrams to derive design metrics. This process is both simplistic and useful. This research advocates the use of design metrics derived from UML class diagrams to improve object-oriented software design quality, and to guide the efforts in the software development lifecycle phases.

General Terms
Management, Measurement, Documentation, Design, Reliability, Theory

Keywords
Object-orientation, Object-oriented design, software engineering metrics, design metrics, unified modeling language, factor-metric-criteria

1. INTRODUCTION

Measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules [1]. Measurements are used to objectively and quantitatively evaluate a particular object. Accordingly, organizations are in need of measurement to increase understanding, and to control and optimize measurable practices. In more recent years, the software industry has adopted measurement programs to continuously improve defined software processes, more commonly referred to as software engineering metrics.

Software engineering metrics are the result of the software industry evolving and maturing over time.

As with any engineering discipline, the software industry requires measurements for the evaluation, and improvement of software development products, processes, and resources. Being that the computing field is still a relatively new industry; much effort is being put in developing practical software metrics.

The primary objective for most software metrics is to achieve high product quality, and to simplify the planning process. Software product metrics are applied to incomplete software products in an effort to measure their complexity and to predict properties of the final product [2]. For example, some product metrics can be used to estimate the amount of testing necessary or total development costs for a specific software development project. In contrast to product metrics, process metrics are concerned with the progress of a systematic design process and to predict future effects, problems, and costs of a development [2]. For example, maintenance cost can generally be estimated using process metrics coupled with other techniques outside the scope of this paper.

2. SOFTWARE DESIGN METRICS

Another approach is to use measurement as a necessary step of the design process to assist software developers in improving the overall quality of a product [2]. This beckons the question as to which elements positively and negatively affect a system’s design and how to use this information to improve the system design process. Consequently, the most difficult part of using design metrics is determining how to correlate the metrics to acceptable design principles. Design metrics, theoretically, take the next step by relating the software design principles to structural software characteristics.

Some design metrics help overcome two problems often encountered with traditional product and process metrics: complexity, and lack of thresholds and criteria. Complexity becomes an issue when many metrics are combined in a single formula. It becomes difficult to trace backwards and relate the results to specific characteristics of the system. Many metrics have no criteria or thresholds to judge whether a value indicates a situation that needs improvement. Without clear guidelines it can be difficult to act on the metrics.

Another important question is whether design metrics can be considered process or product metrics. Unfortunately, the answer isn’t clear. Design metrics can be characterized as both product and process metrics. For instance, examining design is clearly a process that can improve the overall quality of a software product; however, the design metrics can also guide future development efforts, or estimate the cost of maintenance due to complexity.
Thus, design metrics could be process metrics, product metrics, or both.

2.1 Object-oriented design metrics

With the emerging popularity of object-oriented technology, many advantages and benefits have been brought to software engineering as well as some new challenges. One of these challenges is how to measure and improve object-oriented software design. Unlike more traditional structural paradigm design metrics, the object-oriented paradigm requires a shift to a different way of assessment due to the differing characteristics.

Object-oriented design metrics can be viewed from a number of differing perspectives. At the highest-level, there are static and dynamic design metrics [2]. Dynamic metrics are extracted at run-time, whereas static metrics can be extracted earlier in the development cycle. Another way to characterize design metrics is at a particular development phase. One would think design metrics are implied at the design phase. This is not the case. The focus of the approach presented here is at the design phase; therefore, only static metrics are presented.

A recent study has shown that the use of object-oriented design metrics reduced the error rate to a meager 9% when estimating post release defects for a commercial Java application [2]. Undoubtedly, the application of object-oriented design metrics can be a potentially useful device early in the software development cycle.

3. OBJECT-ORIENTED PARADIGM AND DESIGN

The object-oriented paradigm is characterized by many different factors; however, the most important among these is that the data and the operations that manipulate the data are bound to the same object. Unlike the more common procedural paradigm, abstraction, encapsulation, generalization, association, and specialization are natural mechanisms. Consequently, there is a seamless transition between the analysis-design-implementation phases of the software development lifecycle. This section discusses some object-oriented characteristics that might determine design quality: [3]

- Depth and width of class hierarchies – A deep hierarchy is considered more extensive to the reuse of classes because inheritance is reflected in a deep and narrow rather than shallow class hierarchy. A deeper hierarchy also attests to delegation of responsibility.
- Variety of classes – Certainly, the complexity of a system can be measured by the number of classes used in a system design. This is not a definitive measurement because a complex problem requires a larger number of classes than a simple one.
- Newly developed classes – The number of classes that are newly created should be minimal. The more new classes that are created, the more maintenance and complexity of the overall system.
- Reused classes – The larger the number of reused classes in a project, the better the quality. The reasoning behind this is that the classes in the library are supposed to be correct and reliable because they have been used and tested. Most importantly, reuse of existing classes can reduce the workload of the programmer.
- Attributes in a class – Attributes in a class should be encapsulated. The only interclass communication should be message passing. The lower level of data encapsulation, the higher the level of complexity.
- Operations per class – The number of operations per class should be minimal. A large number of methods and data in one class mean a lack of delegation of responsibility and a large concentration of knowledge and responsibility. This is generally discouraged.
- Reusability of operations – Building generic operations that are tightly coupled to a class and loosely coupled with other classes leads to better reusability. This is also realized though inheritance.
- Length of operations – The length of operations might affect the number of operations in a class. Similar to limiting the number of operation per class, we should also limit the functionality of an operation to a specific task, also known as functional decomposition.
- Number of arguments and types – The number of arguments and the type of arguments passed to an operation can be a measure of complexity. Passing primitive data types might be less complex than passing abstract data types. Generally an operation that accepts five arguments has more responsibility than a method that is passed one argument.

With these general object-oriented design principles, we can build metric models that support a variety of quality considerations, such as reusability, reliability, complexity, et al. It is important to note that these general considerations are not mutually exclusive. Some reusability criteria could also be criteria to measure complexity, and so forth.

4. FACTOR-CRITERIA-MEASUREMENT

So the next logical step is determining how to organize metrics in a way that they are meaningful and useful to software designers. Many different models and approaches have been suggested. One method presented by Erni and Lewwerentz is the Factor-Criteria-Measurement approach. Here, the Factor-Criteria-Measurement (FCM) approach is selected and enhanced. The enhancement focuses on the approach of collecting and presenting the metrics. The FCM suggests the first necessary step is to recognize the factors that influence software quality. After clearly defining these factors, we select criteria that influence each factor, and define measurements for the given criteria [3]. Figure 1 depicts this notion.

![Image](image_url)

**Figure 1 – FCM Diagram**

The FCM can be integrated and extended. In [2], a slightly extended software quality model is built around the premises of the FCM. This model traces a quality goal to a number of factors, and then the factors are mapped to design principles. From the design principles, design rules are defined, and the accompanying metrics corresponding to the design rules. This makes the task of tracing the metrics to meaningful design principles a simple task.
A reusability software quality model presented by [2] is used to explore the model-driven approach.

Figure 2 shows the extended FCM model integrated with a software quality model for reusability. Reading the quality model from right to left, reusability is mapped to factors that determine reusable software. These factors are mapped to design principles: modularity, simplicity, and abstraction. For every design principle, a set of design rules are defined, and the accompanying object-oriented design metrics are derived. This extended model allows developers to focus on design rules and principles, not just meaningless numbers.

The model shows that reusability of object-oriented software is largely based on three design principles: modularity, abstraction, and simplicity. Modularity is characterized by modular decomposability, composability, understandability, continuity, and protection [2]. Low coupling, good cohesion, simplicity, sufficiency, and completeness determine abstraction. Finally, simplicity is based on the contention that “the simpler the design, the easier to understand, extend, and reuse [2].” This software quality model is used in the ensuing sections to provide a framework for the metrics we examine.

5. EXTRACTING DESIGN METRICS

A strong tenet of object-oriented design metrics is that they are identifiable early in the software development process. Identifying the design metrics early in the software development lifecycle can guide future development efforts, or be used to estimate future costs associated with a system. All things equal, the earlier metrics can be collected in the development lifecycle, the better.

We know from experience that software metrics programs without automation or simplistic measurement procedures can be extremely costly to an organization. This necessitates the need for quick and simple procedures to extract useful software design metrics early in the development lifecycle. Thus, the notion of using the Unified Modeling Language (UML) Class Diagrams to derive useful object-oriented design metrics prior to the implementation phase can save time and money, while providing quality information for software designers. In the following sections, the extended FCM model is used to integrate with a model-driven approach.

5.1 A PRIMER ON CLASS DIAGRAMS

Software designer seasoned in both object-oriented techniques and UML class diagrams should extract the metrics from the diagrams. This helps ensure the metrics are recorded properly. In this section, UML class diagram notation is briefly touched upon to lead into a case study using the model driven approach.

UML is the result of many years of software modeling experience. The models are rich with content, which attests to the easy transition from design to implementation. Figure 3 displays the notation used in class diagrams [3]. It is clear that any metrics regarding countable characteristics can easily be accounted. It is assumed the reader has a basic understanding of UML class diagrams.

Class diagrams clearly present composition and association between classes. Class diagrams also depict cardinality, multiplicity, abstraction, encapsulation, generalization, and specialization. In the following section, we show how simply the diagrams can be used to extract the metrics.

6. MODEL-DRIVEN APPROACH

To exemplify the practicality of using class diagrams to extract object-oriented design metrics we revisit the extended FCM reusability model shown in Figure 2. At this point, software designers should have selected a software design goal, and traced through the design principles, design rules, and clearly defined metrics to analyze. In Table 1 the metrics selected in the reusability model are shown.
Table 1 – Metrics derivable from UML class diagrams

<table>
<thead>
<tr>
<th>Metric</th>
<th>Design Rule</th>
<th>Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public Attributes</td>
<td>Encapsulation</td>
<td>Yes</td>
</tr>
<tr>
<td>Public Methods</td>
<td>Narrow Interfaces</td>
<td>Yes</td>
</tr>
<tr>
<td>Method Arguments</td>
<td>Narrow Interfaces/Method Arguments</td>
<td>Yes</td>
</tr>
<tr>
<td>Directional References</td>
<td>Decoupling</td>
<td>Yes</td>
</tr>
<tr>
<td>References</td>
<td>Decoupling</td>
<td>Yes</td>
</tr>
<tr>
<td>N LOC Methods</td>
<td>Method Complexity</td>
<td>No</td>
</tr>
<tr>
<td>Weighted Method</td>
<td>Method and Class Complexity</td>
<td>No</td>
</tr>
<tr>
<td>Method Count</td>
<td>Class Complexity/1:1</td>
<td>No</td>
</tr>
<tr>
<td>Interface Size</td>
<td>Class:Abstraction/Maximize Cohesion</td>
<td>Yes</td>
</tr>
<tr>
<td>Abstract References</td>
<td>Abstract Interfaces/Decoupling</td>
<td>Yes</td>
</tr>
<tr>
<td>Un-Cohesive Methods</td>
<td>Maximize Cohesion</td>
<td>No</td>
</tr>
</tbody>
</table>

The first column is the design metric, the second are design rules corresponding to each metric, and the third column states whether the metric can be extracted from a UML class diagram. Only three of these eleven metrics cannot be derived from a class diagram.

Since seasoned software designers are expected to extract the metrics, when it comes to analyze the metrics, the designers will have stronger understanding of the system and whether the system design is in compliance with the set design rules. Next we will examine an example UML class diagram and extract the metrics into an easy to view format.

6.1 Example of the approach

Figure 4 is a simple UML class diagram showing a potential design relationship between an employee and their pay type. To analyze the system we must look at the diagram in three perspectives: system-level, class-level, and method-level. A system level perspective considers the relationship between all the classes in the system. A class-level perspective looks at each individual class, including the number and type of methods and data types. A method level analyzes individual methods found in each class with respect to parameters. Here we analyze the class diagram to derive metrics, however, the focus of this paper is not to interpret the metrics, only to present an approach to collect and present them.

We can see at a system-level, there are four classes; three concrete classes, and one abstract class. There is an association between an Employee and a Paytype. The Paytype class has two children represented by generalization.

In a class-level perspective, we can see that Employee class is composed of an integer primitive, two String, and a PayType. All the data are private; and therefore, encapsulated within the Employee class. There are six public methods found in the Employee class showing a narrow interface to this class. Three methods accept no arguments, and the other three all accept one argument. To help understand all this information, we might present the data in a more meaningful format as shown in Figure 5.

This hierarchical model for the presentation of the metrics traces the through different levels of the system to help the users of the metrics understanding their meaning. The presentation of the information is very important. If the information is not organized...
in a meaningful way, it is difficult to understand how the metrics relates to specific design rules and components within the system.

6.2 Limitations to the approach

From our previous example we can easily identify some limitations to this model-driven approach. Most obvious of these limitations is that only static metrics can be derived from UML class diagrams. Since the metrics in this approach are collected during the design phase, we cannot collect any dynamic object-oriented design metrics. A counter to this limitation is that most object-oriented design metrics are static. Moreover, this does not mean these metrics cannot be collected later in the development process.

A not as easily noticeable drawback to this approach is that there is a lack of metrics that consider aggregation, and cardinality and multiplicity. These concepts are very important in software design and could be transformed into useful design rules and metrics. Although the model-driven approach does have some limitations, the practicality and simplicity of approach makes this a desirable methodology.

6.3 Other design metrics and UML

Here different object-oriented design metrics are shown to show that these metrics can be extracted from UML class diagrams. All the metrics shown in this section are static. The following summarizes the metrics, accompanied with a brief description: [4]

- Depth of Inheritance Tree – The maximum length from the node to the root of the tree (Generalizations)
- Number of Children – Number of immediate sub-classes subordinated to a class in a hierarchy (Generalizations)
- Coupling Between Classes – The number of classes to which a class is coupled (Associations)
- Class Complexity – A hybrid of many metrics to determine the complexity of a given class (Associations and Generalizations)

In combination with the metrics shown in the reusability model, we have surveyed many object-oriented design metrics. The metrics described here can also be found in UML class diagrams.

7. CLOSING REMARKS

In this paper we have shown that the use of UML class diagrams in collecting design metrics can be a simple and useful approach to quality software design. Simply gathering object-oriented design metrics does not determine quality, however, they can be mapped to design rules in the pursuit of design quality. Design metrics need to be coupled to specific design rules and mapped to design principles, factors, and goals.

The FCM approach and its extensions can be useful in providing structure and applicability to use of object-oriented design
metrics. No metric should be collected unless it is clearly defined and mapped to a specific purpose. The FCM helps us build upon this principle by providing a general framework for metric development and usage.

This paper shows that UML class diagrams can be effectively used to gather most object-oriented design metrics early in the development cycle and integrated with existing metric approaches. Collecting the metrics early in the development process can be beneficial to guiding the development effort in the subsequent development cycles. The approach is simple and can be implemented by novice to seasoned software designers. We've also seen how the metrics could be presented in a hierarchical model to make the metrics more meaningful to users. A poor presentation of metrics reduces the understanding and thus the use of the metrics. Although this approach does have limitations, the practicality and simplicity of approach makes it a desirable methodology.

8. FUTURE WORK
Research in object-oriented design metrics and the use of UML diagrams is open-ended. There is limited research in effectively applying object-oriented design metrics in industry. This section summarizes some future research ideas that could prove to be very useful to industry.

There is a UML consists of nine diagrams used for varying tasks. Only one these nine models are explored in this research. Other models could also be evaluated to determine whether valuable software metrics can be extracted.

Computer-Aided Software Engineering tools are often used to automate system design. However, most major UML tools do not generate comprehensive object-oriented design metric reports. Adding a utility for design metrics reporting to a CASE tool would not require too much effort, however, the benefits of such a utility would reduce the effort necessary to collect metrics and provide a consistent framework for metrics reporting.

All the metrics shown here do not consider aggregation, cardinality, or multiplicity. Work needs to be done to determine whether these concepts can be traced to design principles, and metrics.

9. REFERENCES