DESIGN ANALYSIS TECHNIQUES FOR SOFTWARE QUALITY ENHANCEMENT

By
DANIEL DEE WILLIAMS

A dissertation submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

August 2007

To the Faculty of Washington State University:

The members of the Committee appointed to examine the dissertation of DANIEL DEE WILLIAMS find it satisfactory and recommend that it be accepted.

_____

Chair

_____

_____

ACKNOWLEDGMENTS

I wish to express my gratitude to my adviser Dr. Orest Pilskalns whose support and guidance has been exemplary and whose friendship has been invaluable. I could not have completed this accomplishment without the support of my family and particularly my wife, Lori. I owe much to the entire staff at WSU for their openness and availability. Everyone with whom I worked was willing to go the extra mile to help me achieve success.

DESIGN ANALYSIS TECHNIQUES FOR SOFTWARE QUALITY ENHANCEMENT

Abstract

by Daniel Dee Williams, MSCS
Washington State University
August 2007

Chair: Orest Pilskalns

In the software life cycle, early detection and correction of flaws and weaknesses in the design phase can reduce overall costs associated with development and maintenance. Current software development methodologies such as the Model Driven Architecture rely on quality *Unified Modeling Language* (UML) design models. Often these models are complex and consist of many structural and behavioral views. This can lead to inconsistencies between views. Existing approaches remedy many of these inconsistencies but do not address consistency across design views nor software quality metrics in the design phase. This thesis presents two approaches. (1) The first approach is aimed at detecting and resolving security faults in UML designs. The approach defines the notion of security consistency in designs, analyzes UML views for security inconsistencies, and generates a set of recommended design changes that include Object Constraint Language (OCL) expressions. The OCL can be used as a test oracle in both the design and implementation phases of the software life cycle. This work provides an empirical study that demonstrates that the generated OCL

reduces security faults. (2) The second approach evaluates design quality using metrics. During software development it is important for component developers to design components that show high cohesion within a component and low coupling between components. Empirical data shows that software artifacts possessing these properties are easier to develop and maintain. Current practice in design metric evaluation relies on extracting structural metrics from individual UML views. This thesis defines a dynamic approach that collects metrics during execution of a model that integrates both UML Class and Sequence Diagrams. These design metrics are used to evaluate component choices by examining cohesion and coupling properties. The design metrics are based on code metrics that have been positively correlated with maintainability and quality. This thesis provides an empirical study that demonstrates a positive correlation between design and code metrics.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The evolution of software development processes is still relatively young and is branching in many directions as research explores new approaches. One promising trend is the use of *Model Driven Architecture* (MDA) where the software designer encapsulates the design concepts with a set of UML views. This kind of development process bears a resemblance to the firmly established practice in circuit design of developing a model of the system prior to implementation in a manufacturing environment. Although the two realms of product development are quite different in many respects, there is enough similarity that each can benefit from some of the strengths of the other. To illustrate this point, let's explore the comparison further.

The primary goal of creating a model of a circuit design is to simulate the behavior of the circuit. If the circuit model is of high quality and simulates the circuit with sufficient accuracy, the designer can achieve a high level of confidence that his circuit design is correct and will meet the goals of his product specification. This high level of confidence is critical, because the expense of implementation of the design is

enormous. Any flaws in the design that are discovered after implementation in real hardware will be very expensive, in terms of resources and time, to fix with a second design cycle.

Many aspects of the circuit design process are also present in the software design process. After a system design has been conceived and represented in some kind of design specification, the project is ready to enter the implementation phase. As in the circuit design cycle, the implementation phase is very expensive. A major portion of allocated resources are consumed in this phase. Flaws in the design concept that are not discovered until after implementation will require much more effort to eliminate than those discovered prior to implementation. However, software development processes are much more agile then circuit design processes, and revisiting a design is usually part of the process known as an iterative approach. Yet, this should not be an excuse to accept or expect low quality software design documents.

The similarity of the two design processes diverges significantly in the area of creating a system model. The model assembled by the circuit designer is a functional model. Its intended use is that of simulating system behavior. In contrast, the model created by the software designer is purely conceptual and often contains structural and behavioral views of the software to be developed. Behavior may certainly be specified, but the behavior cannot be simulated. In addition to various views, a software designer needs to take into consideration "best practices", security concerns, quality metrics, etc. This leads to designs that are very complex and can be internally inconsistent. The research questions this thesis attempts to answer are:

1. Can an approach be developed that analyzes designs for consistency and generates constraints that can be used to increase the quality of the software by using the design and generated constraints?

2. Can a design phase approach be developed to define and extract metrics that are good indicators of quality after implementation?

This thesis proposes two approaches that answer the research questions. Both approaches have been validated using empirical studies. These approaches should not be considered comprehensive, but by employing these techniques the designer will certainly be able to raise his confidence level in the quality of his system design concept. These validation activities are divided into two categories that will be treated in two separate chapters. The first will focus on the notion of security consistency between the various *Unified Modeling Language* (UML) views of the system to be modeled. The second approach uses the concept of quality metrics derived from the UML views. Various quality judgments are formulated based upon the measurement of cohesion and coupling represented in the UML views.

Chapter 2

Background and Related Work

2.1 The UML

The Object Modeling Language is officially defined and maintained by the Object Management Group (OMG). The goal of the group is to create a language for specifying software systems that employ object-oriented languages in the implementation phase. Thus, the design views offered by the UML are tailored toward describing objects and their interactions.

The core of the UML consists of the following diagrams and notations: Use Cases, Class Diagrams, Sequence Diagrams, State Diagrams, Activity Diagrams, and Physical Diagrams. It has been found that in practice, only a small subset of the entire UML is actually used [24]. The most common diagrams used are Use-Cases, Class, Sequence, and State Diagrams. Use-cases are used primarily for requirements, which drive the design. There has been considerable work done on transforming State Diagrams into precise models by Harel et al. [25], so this work will not focus on State Diagrams. Therefore, this thesis focuses on the UML Diagrams and artifacts that are most often used by developers, Class and Sequence diagrams.

UML diagrams provide various views of how a software system operates. For example, the Class Diagram provides a static view of the relationships between classes in a software system. By itself this diagram cannot be used to test how the system operates because it does not include any dynamic information. The Class Diagrams represent the static relationships between objects in a system by showing associations and subtypes. An example of an association is that a poker-dealer deals with cards, where the dealer and cards form an association. An example of a subtype is that draw-poker is a type of poker. In addition the Class Diagram, contains attributes and operations for a class. Figure 3.1 is an example of a Class Diagram for an on-line vacation package purchase system. The elements that are key to Class Diagrams are the class name, visibility, super class information, attributes from the class, methods for the class, and constraints for both the attributes and method parameters and their visibility. These elements are defined in the UML 2.0 specification [30].

A Sequence Diagram provides dynamic information about calls between objects. The Sequence Diagram represents the dynamic relationships between objects in a system, by showing method calls and logical decisions. The diagram consists of objects represented by boxes at the top of the diagram. From each box extends a line representing the life-line of the object. The arrows between the lifelines present the method calls between the objects. Figure 3.3 shows a Sequence Diagram associated with the handshaking method from a simplified *Secure Socket Layer* (SSL) implementation. The key elements of a Sequence Diagram are the object names, class names, method calls (including calling object and called object), method parameters, return types, decision constructs, and looping constructs. These elements are defined in the UML 2.0 specification. Sequence Diagrams do not contain the necessary information to build an object, since it does not contain class hierarchy information. So individually, the UML diagrams only provide a piece of the overall system and its operation.

2.2 Security Approaches using Object Constraint Language (OCL)

Because this thesis relies on standard OCL for specifying constraints in a software design, a brief introduction is necessary. OCL is one of many different and specialized embodiments of the notions and axioms of the First Order Predicate Logic (FOL). Its constructs allow the expression of the foundation concepts contained in the FOL, such as predicate statements, quantification and inference rules. However, because OCL was defined for a subset of the domain of all possible logic applications, its syntax narrows the formation of statements to specific categories crafted for expressing constraints on a system design.

The constraints constructed from OCL syntax aim to impose boundaries on object-oriented design artifacts. These constraints can be divided into three categories: invariants, pre-conditions and post-conditions. An invariant is a condition that must always be satisfied by all instances of the constrained artifact (i.e. an interface, type, method etc.). An invariant is a Boolean expression that is true whenever the invariant is satisfied. Pre-conditions and post-conditions are similar constraints that must be satisfied before or after a method is executed.

The primary purpose of the OCL is to define constraints. Therefore, the OCL has no side effects on the system state. In other words, the language defines boundaries, but does not alter the attribute values of a system. Recent extensions of the standard language definition have considerably expanded the power and expressiveness of the language. These additions to the syntax now allow some constraints upon system behavior. This added utility will be utilized in this thesis.

Adding security mechanisms during the design phase often requires constraining the design. The obvious language for writing constraints for UML designs is the OCL.

Using OCL to increase the security robustness of a software design has been applied by others. The common theme of most of these approaches is to augment the OCL by adding security enhancements to the language.

Medina et. al. [3] propose enhancements to the standard OCL which requires modification of the OMG specification. They call their modified OCL the *Object Security Constraint Language* (OSCL). The OSCL is used to assign a security level to all classes, attributes, operations, and associations in a UML design. In contrast, the methods presented in this thesis require neither an extension of the OCL nor the use of security levels.

Ahn and Shin [1] propose role-based authorization constraints to specify access control systems. They use OCL to enforce a set of rules that define separation of duties. The rules constrain the system to prevent the assignment of conflicting roles to the same user. In addition their rules address role-based conflicts such as:

1. Conflicting permissions cannot be assigned to the same role.

2. Conflicting users cannot be assigned to the same role.

3. Conflicting roles cannot be activated in the same session.

A second example of the role-based approach is described by Alam et al. [2]. In their approach, a designer builds an interface model for accessing web services by including security requirements with OCL and imposing a role-based access model. The designer then generates from these specifications a complete configured security infrastructure in the form of *Extended Access Control Markup Language* (XACML). Again, the approach proposed in this thesis requires neither the imposition of a role-based conceptual model nor the use of *extended OCL* and other markup languages.

2.3 Object-Oriented Cohesion and Coupling Metrics

Coupling measures the degree of interdependence or interaction between software modules [23] [32]. In this context, software modules are classes or components. A low amount of coupling is desirable between software modules, because high coupling has been empirically linked with low quality and high maintenance. *Cohesion* is the extent to which an individual module relies on internal components to perform a task [23] [32]. Once again, in this context, modules are classes and components. A high amount of cohesion in software modules is desirable, because empirical studies have shown that software with high cohesion is of higher quality and easier to maintain.

In [15], Briand and Wuest summarize empirical results concerning code coupling and cohesion metrics that have been shown to correlate with software quality. Of the 37 different coupling metrics used in empirical studies only a few had a positive statistically significant relationship with quality ($p < 0.01$). The metrics that showed a positive relationship included *Response For Class* (RFC) [16], *Other Method-Method Import Coupling* (OMMIC)[19], and *Information-Flow-Based Coupling* (ICP) [18]. In addition, of the 12 cohesion metrics used in empirical studies only one had two empirical results showing a significant relationship with quality, the *Information-Flow-Based Cohesion* (ICH) [18]. Table 2.1 summarizes the selected metrics [15].

In [12], Basili et al. experimentally investigate some object-oriented metrics. In the experiment they created eight different software managements systems and collected several metrics: *depth of inheritance* (DIT), *number of children* (NOC), *weighted method per class* (WMC), *coupling between object classes* (CBO), *response for class* (RFC), and *lack of cohesion metric* (LCOM). In the experiment several of the metrics were helpful in predicting fault proneness. They state that these object-

Table 2.1: Coupling and Cohesion Metrics.

Metric Name	**Entity Measured**	**Type**	**Scale**
Response For Class(RFC)	class coupling	indirect	absolute
Other Method-Method Import Coupling (OMMIC)	class coupling	indirect	absolute
Information-Flow-Based Coupling (ICP)	class coupling	indirect	absolute
Information-Flow-Based Cohesion (ICH)	class cohesion	indirect	absolute

oriented metrics could be helpful in early stages of software development. Some of these metrics are more easily adapted for designs than others. For instance the DIT would be easy to calculate in the design phase. However, the RFC metric is more complex and needs method response information. Their work can be expanded by defining these metrics for a particular design notation such as the UML.

Dynamic metrics are important when evaluating object-oriented designs and code. Dynamic metrics allow for the measurement of dynamic features in object-oriented software such as dynamic binding. Inheritance and polymorphism in object-oriented software often hide or misrepresent class coupling. Dynamic metrics allow for a more accurate measurement of coupling and cohesion by examining a system while it is executing. Dynamic metrics can be applied at many different levels of granularity including the object level, class level, and component level.

In [11], Arisholm et al. define and investigate dynamic object-oriented code metrics for both code and design. They propose that static metrics do not properly reflect modern object-oriented code due to the increased use of inheritance and dynamic binding. They propose new dynamic metrics for coupling that precisely measure coupling in systems that use inheritance and dynamic binding. Their dynamic metrics using the following categories:

1. Direction, which indicates if the coupling is an import or an export.

2. Entity, which indicated the granularity (class or object) of the coupling measure.

3. Type, which can be categorized as follows:

   (a) Dynamic Messages (total count of messages from one object to another)

   (b) Distinct Method Invocations (distinct method calls between objects)

   (c) Distinct Classes (count classes used)

These categories are combined to form 12 different dynamic coupling measures. They also provide empirical data that indicates that these metrics may be linked to quality.

In [31], Ritzhaupt defines some simple metrics for class diagrams. The metrics include counts of the following design attributes: public methods, method arguments, directional references, and references. Ritzhaupt does not provide any analogous code metrics that are linked to quality.

Currently there is one UML tool called SDMetrics (http://www.sdmetrics.com) that extracts structural metrics for designs, such as size and complexity. The tool does not simulate execution, thus the metrics that it provides are based on counts of structural properties.

2.4 Component Based Approaches

One of the objectives of this thesis is to apply analysis methods to component based software. Qualification of components in the design phase has not been addressed in the software community, primarily due to lack of support for precise component

definitions in the UML. The following work in component qualification in later phases serves as a starting point.

In [28] [27], Kontio et al. apply a selection and evaluation method to multiple case studies. The method investigated is referred to as OTSO (Off-The-Shelf Option). OTSO describes a systematic approach to selecting packaged components. The method includes six phases: search, screening, evaluation, analysis, deployment, and assessment. Lester et al. [29], applied the idea of using stereotypes, class compartments, and association rules for qualifying the reuse of software artifacts. These UML constructs are used to define search criteria for reuse candidates. The stereotype is used to limit the search of objects to those objects that contain the stereotype or are derived from the object with the stereotype. Attribute-Value classification can be used to provide a structured way to integrate association roles into the search criteria of an object.

The COTE (COmponent TEsting) project [26] is concerned with developing an integrated environment (IE) for qualifying and testing components. The research is primarily interested in using the IE for testing and qualifying implemented components modeled in UML. COTE is borrowing from Offutt's and Labiche's work as outlined earlier. Their research interests lie in applying automated testing to component qualification and integration at the system testing phase of software development. Thus the COTE group is not developing new techniques for component testing, but is focused on developing an integrated environment of existing techniques. Their approach does not address design testing at the pre-implementation phases.

2.5 Literature Search Summary

Momentum is beginning to shift toward the concept of quality assessment and validation in the design phase of development, but the movement is still in the early stages. Standardized design model definition is becoming a reality through the UML constructs, but these constructs still allow too much ambiguity in their expression of system specification. This work attempts to add to the momentum by addressing some of these weaknesses.

Most of the prior work that addresses quality enhancement with a security focus requires the imposition of a complex role structure or other extensions of standard OCL constraint language. In contrast, this thesis offers an approach that does not require convincing the mainstream to adopt a highly specialized syntax to strengthen the security of software products.

Metric collection as a means of design analysis is also a relatively new area of research. Dozens of these metrics are being defined, but few of them have been proven to correlate well with quality in the final product. This thesis uses a small subset of the field of defined metrics and presents some data to validate the effectiveness of the kind of metrics involved.

Chapter 3

Security Consistency in UML Designs

Often the first steps for designing a software system (using the Model Driven Architecture (MDA) methodology and the Unified Modeling Language (UML)) include defining Class and Sequence Diagrams. The Class Diagram depicts the structural aspects of the system while the Sequence Diagrams illustrate the behavioral aspects of the system. The behavioral model focuses on describing the desired sequence of events in the system. Interaction between diagrams can cause non-obvious undesired behaviors. The Sequence Diagram is specifically designed to illustrate only desired behavior. Many potential security risks may be overlooked if only desired behavior is accounted for while undesired behavior is not addressed by the system design.

As an example, a Class Diagram may contain an association between two classes A and B. The association is marked with multiplicities indicating a one-to-many relationship between A and B. However, in the Sequence Diagram we see that only two objects of type B need ever be created. The diagrams are usually considered consistent; however, let us assume a worst case scenario. The implementer of the system

allows an unrestricted number of B objects to be created by a web client resulting in a system failure. In addition, designs should be consistent with established security principles. One of Graff's [5] design principles states "a program should run with the minimum privilege necessary to complete its task". If a design violates this principle we should note that the design is inconsistent with known secure design principles. Therefore, we need a way to describe consistency from a security perspective that takes into account consistency across design views and consistency with secure design principles.

Once we define consistency from a security perspective, we need a systematic approach to analyzing and detecting inconsistencies. Once the inconsistencies are detected, they need to be resolved. This leads to the research questions addressed in this chapter:

1. Can we define security inconsistencies for UML design views?

2. Assuming we have an adequate definition of security inconsistencies, can we define an approach for detecting and addressing these inconsistencies?

To address these questions, this chapter first defines the notion of security consistency and proposes an approach for detecting and addressing security inconsistencies. Next, the chapter presents an example system and how the techniques may be employed with it. Finally, the chapter details the design of an experiment that uses the same example system and then presents the results with some accompanying conclusions.

3.1 The Security Consistency Approach

Current UML design analysis tools check for static consistency between elements in UML Diagrams. Diagrams are considered consistent, as long as values from one diagram lie within the bounding values of another diagram. Yet one view may be less restrictive than another view. This can lead to misinterpretation of the design during implementation resulting in security or quality faults. Consistency is usually measured in terms of boundaries. For example, a Class Diagram may state that the association (between objects of type A and B) is a normal association. The Sequence Diagram may show that the relationship is more specifically a composition. Current design tools will not complain since the Sequence Diagram does not violate the association. However, as pointed out in the introduction such inconsistencies can lead to security and quality faults later in the design.

The approach proposed here defines "security consistency" as the consistency of bounding values on non-abstract *elements* of a UML diagram. In the UML meta-model, *elements* are the building blocks of UML Diagrams. The role of a metamodel is to define the semantics for how model elements in a model become instantiated [8]. The UML infrastructure specification [8] defines the term *element* as follows: "*Element* is an abstract metaclass with no superclass. It is used as the common superclass for all metaclasses in the infrastructure library." By constraining the super class of the UML meta-model we can impose a consistency requirement across all UML diagrams and their elements. Consider the following more formal definition for "securely consistent":

> Bounding values (actual or derived) must agree at all points between common elements in two or more diagrams if the diagrams are to be considered securely consistent.

If we consider "secure design principles" as a set of UML diagrams and accompanying constraints (Secure Patterns), this definition of security consistency answers the first question posed in this chapter. However, we are faced with the problem of comparing bounding values across diagrams. In the introduction, the example of consistency between multiplicities in a Class Diagram and the number of instances used in a Sequence Diagram was pointed out. The Sequence Diagram values need to be derived by examining the entire Sequence Diagram. This problem presents the need for "normalizing" bounding values between diagrams or creating a set of analysis rules.

Given the security definition provided earlier, this thesis outlines a small set of rules (that will eventually be lengthened) for some commonly used diagrams. Class Diagrams were selected because they are often the starting place when designing a new system. Class diagrams may not properly restrict how objects should interact. It is left to the behavioral views to show proper object interaction. The following four rules enforce security consistency between Class Diagrams and Sequence Diagrams:

Rule 1 Check if the **operation access** of each object in a Class Diagram is *securely consistent* with the object interactions supplied in the Sequence Diagrams.

Rule 2 Check if Class Diagram **compositions** are *securely consistent* with the object interactions in the Sequence Diagrams.

Rule 3 Check if Class Diagram **multiplicities** are *securely consistent* with the behavior of objects in the Sequence Diagrams.

Rule 4 Check if the **sequence of access** to the operations in the Class Diagram is *securely consistent* with the sequences depicted in the Sequence Diagrams.

The goal is to satisfy each rule by creating an algorithm that generates an OCL

constraint based upon the behavior of the system. The OCL constraint can be used to check if the system is consistent either using Pilskalns et al. [9] approach or using Gogolla's constraint checking tool [4]. In addition these constraints can be used to generate unit tests for testing code in the implementation phases of the life cycle.

3.1.1 Rule 1 : Operation Access

Class Diagrams use an association to show that one class uses operations in another class. The Class Diagram (without constraints) does not show which operations are allowed to be called by a specific class. Therefore, Class Diagrams contain an inherent access ambiguity by default. Often, the designer can overcome this problem by specifying access in a Sequence Diagram. However, the Sequence Diagram only depicts correct access to an object and does not restrict or constrain the access. By applying rule one, we can generate constraints that restrict access in a Class Diagram based upon usage in a Sequence Diagram. These constraints can serve as design phase test oracles [9], as reminders when implementing the system, and can also serve as tests cases for unit testing.

In order to carry out this consistency check we need to assess the dynamic behavior of the Sequence Diagram. Sequence Diagram "execution" [9] can be accomplished by traversing all feasible paths in the Sequence Diagram. To check if associations are consistent from a security perspective we need to observe the object interactions in the Sequence Diagram. The objective is to create constraints, by explicitly allowing only the interactions between classes that are associated via method calls in the Sequence Diagram. These constraints can be checked for consistency using either the Pilskalns et al. [9] approach or the Gogolla [4] approach.

In Algorithm 1, SD represents the current Sequence Diagram, i represents a

sequence event in SD, $ob_{i,j}$ and $ob_{i,j+1}$ represent two consecutive objects associated with the i-th event, C_j represents the type (class) of $ob_{i,j}$, C_{j+1} represents the type (class) of $ob_{i,j+1}$, m represents a method of $ob_{i,j+1}$, and $C_{j+1}.m.oclSet$ represents the set of classes that are allowed to access m. $I(c)$ is used to represent the set of all invariants within an OCL context c. The outputs of this process are OCL statements consisting of *context* and *inv* keywords.

In Algorithm 1 each method is assigned an $oclSet$, which is the set of classes that are allowed to call the method. The $oclSet$ for each method is initially empty, thus all classes are not allowed to interact with the method. In Algorithm 1 each object pair $ob_{i,j}$ and $ob_{i,j+1}$ associated with a method call is examined. The calling object's class is used to constrain the called class. The constraint allows access from only the calling class and by default restricts access by any other class. As the Sequence Diagram is traversed classes acquire constraints that allow access based upon the method calls used in the diagram.

3.1.2 Rule 2 : Composition

Another significant form of constraining class relationships pertains to observing an object's life span in comparison to other objects. UML defines different kinds of associations with respect to the relationship between the class of a container object and the class(es) of the object(s) it contains. Thus, a container object is a *composition* if the contained objects are:

1. created by the container
2. and their life span doesn't exceed that of the container.

If it is determined by observing the Sequence Diagram that a composition relationship exists and it is not depicted in the Class Diagram, then the relationship needs to

Algorithm 1 Operation Access Algorithm

```
/* Pass one - generate sets of all calling classes
     for each method. */
/* Each set should initially be empty. */
for each SD in the system design {
     for each event i in an SD{
          get method m for each (ob_{i,j}, ob_{i,j+1}) in i
          /* Add source object's class to set for m. */
          m.oclSet.append(C_j)
     }
}
/* Pass two - generate invariant statements for each m. */
for each SD in the system design {
     for each i in an SD{
          get m for each (ob_{i,j}, ob_{i,j+1}) in i
          /*actual OCL statement*/
          context ← "context C_{j+1}::m"
          invariant1 ← "inv: oclSet = m.oclSet"
          invariant2 ← "inv: oclSet− >includes(source.type)"
          if I(context) doesn't exist
               create I(context)
          add invariant1, invariant2 to I(context)
     }
}
```

be introduced by adding a constraint. Unlike the first rule, composition can be represented in the Class diagram, so it is possible for the Class Diagram to be inconsistent with the Sequence Diagram. Therefore, the Class Diagram composition should also be represented as a constraint. This allows testing techniques such as [9] to identify the inconsistency.

In Algorithm 2 SD represents the current Sequence Diagram, i represents an event in SD, $ob_{i,j}$ and $ob_{i,j+1}$ represent two consecutive objects in the i-th event, C_j represents the type (class) of $ob_{i,j}$, C_{j+1} represents the type (class) of $ob_{i,j+1}$, m represents a method of $ob_{i,j+1}$, x represents the life span of $ob_{i,j}$, and y represents

the life span of $ob_{i,j+1}$. $I(c)$ is used to represent the set of all invariants within an OCL context c. The *non_compositions* set represents all relationships that are not composite. Likewise the *candidate_compositions* represent all possible compositions.

Algorithm 2 traverses each object interaction pair. The method associated with each pair is tested to see if it is a constructor. The pair of objects are also analyzed to see if the life span of the $j+1$ object lies within the life span of the $j-th$ object. In addition, each object pair can only be admitted into the composition set, if it does not already appear in the non composition set. The actual compositions cannot be determined until the entire Sequence Diagram has been traversed. The outputs of the algorithm are OCL constraints.

Algorithm 2 Composition Algorithm

```
for each i in SD{
    for each (ob_{i,j}, ob_{i,j+1}, m) in i {
        if (! non_compositions.elementOf(C_j, C_{j+1}) && m == constructor && x > y {
            candidate_compositions.add(⟨C_j, C_{j+1}⟩))
        }
        else {
            non_compositions.add(⟨C_j, C_{j+1}⟩)
        }
    }
}

for each element, E_i, in candidate_compositions {
    if (! non_compositions.elementOf(E_i)) {
        /* extract C_j and C_{j+1} from element */
        /*actual OCL statement*/
        context ← "context C_{j+1}"
        invariant ← "inv: C_j.AllInstances().notEmpty()"
        if I(context) doesn't exist {
            create I(context)
            add invariant to I(context)
        }
    }
}
```

3.1.3 Rule 3 : Multiplicities

Often multiplicity assignments in Class Diagrams contain many to many relationships that do not properly constrain the system. Sequence Diagrams provide a behavioral view of object interactions. By observing and recording these object interactions we can record the number of instances actually needed to complete a task. If operational profile information is available via use-cases or domain knowledge, the accuracy of the object interaction information can be increased. However, the Sequence Diagram by itself provides enough information for initial constraints on the multiplicity of objects.

In Algorithm 3, SD represents the current Sequence Diagram, i represents a sequence event in SD, ob_i is the object associated with the i-th event, C_j represents the type (class) of ob_i, and $C_j.counter$ represents the number of instances created of class C_j. $I(c)$ is used to represent the set of all invariants within an OCL context c. The output of this process are OCL statements consisting of *context* and *invariant* keywords.

Algorithm 3 tracks each object and its associated class. Each time a class instance is used, a counter associated with the class is incremented. This simplistic approach does not account for the destruction of classes, thus the instance count may be over-estimated. This can be remedied by contriving a more complex algorithm that uses a table to track the instantiation and removal of each instance of a class.

3.1.4 Rule 4 : Sequence of Operation

The behavior of a component is only partially specified in a Class Diagram. Associations indicate that a class may use services offered by another class within the conceptual constraints of a role labeled by the association. The actual behavior is more specifically depicted in a Sequence Diagram where events are shown occurring

Algorithm 3 Multiplicity Algorithm

$C_j.counter = 0$;
for each ob_i in SD {
    $C_j.counter$++
}

for each C_j in SD {
    /*$actual\ OCL\ statement$*/
    $context \leftarrow$ "**context** C_j"

    /*$actual\ OCL\ statement$*/
    $invariant \leftarrow$ "**inv**: self.AllInstances()− >size() <= $C_j.counter$"

    if $P(context)$ doesn't exist:
        create $P(context)$
    add $invariant$ to $P(context)$
}

in a specified order. Both of these diagrams specify only desired behavior and make no attempt to discourage or prevent undesirable behavior. Therefore, the rigorous designer who wants to build a very secure system is faced with the prospect of trying to anticipate all the ways a careless implementer or even a malicious user might be able to abuse or attack the system. The approach to addressing this form of ambiguity in UML design views is to use constraints to set very tight behavior boundaries. Not only is desired behavior specified, but all other behaviors are constrained to be illegal. This frees the designer from having to anticipate specific undesirable behavior because any and all behavior not specified as acceptable is therefore deemed to be unacceptable. This section adds one of the most powerful sets of constraint statements to a system that will aid the designer in accomplishing such an extreme security goal.

The constraint statements proposed here will enforce the order of events specified in the Sequence Diagram. The enforcement implies that any class or set of classes

that offer operation services used in a Sequence Diagram must be responsible for maintaining a memory of the system state as the sequence is performed. The addition of state variables will require structural modifications to a design view after the generation of the constraint statements with the algorithm shown in Figure 4. These modifications are essentially implementation details that should be left to the implementer.

In Algorithm 4, SD represents the current Sequence Diagram, i represents a sequence event or method invocation in SD, OI_k is the k-th object instance in the current SD, m is the method invoked in the ith event, SC_m represents the type (class) of the source object who is invoking m, SS is a sequence set , and CC_n represents the nth client candidate in a client candidate list. The output of this process is a set of OCL statements consisting of *context* and *invariant* keywords.

Conceptually, the algorithm is quite simple. It scans each Sequence Diagram and reads each method invocation event for each object instance (target) in the Sequence Diagram. As the method invocations are scanned, the algorithm compiles a client candidate list that includes the class of each unique calling (or source) object. At the end of pass one each target object in the Sequence Diagram has an associated list of client candidates that have used its methods. The event scan also compiles an ordered set for each candidate of the methods the candidate has invoked. Thus, if a candidate only uses one method from a particular target object, its associated ordered set will consist of a single element. Because an ordered set of one is a trivial case, there is no need to create a constraint for this case. This is why the algorithm checks the size of each ordered set and only generates a constraint statement for those sets whose size is greater than one. The complete algorithm is shown in figure 4 below.

Algorithm 4 Sequence Constraints Algorithm

```
/* Pass one - generate client candidate lists and sequence sets. */
for each SD in the system design {
    for each object instance OI_k in the SD {
        for each event i terminating at OI_k (read in order) {
            get method m for the event i
            if not SC_m in OI_k.client-candidate-list {
                OI_k.client-candidate-list.append(SC_m)
            }
            OI_k.client-candidate-list.SC_m.SS.append(m)
        }
    }
}

/* Pass two - generate invariant statements for each SD */
for each SD in the system design {
    for each object instance OI_k in the SD {
        for each client candidate CC_n in OI_k.client-candidate-list {
            if CC_n.SS->size() > 1 {
                /*actual OCL statement*/
                context ← "context C_k"
                auxStatement ← "let SS_n : oclSet = CC_n.SS"
                invariant ← "inv: SS_n->forAll(SS_n[p].hasReturned implies
                    (SS_n[p − 1].hasReturned and not SS_n[p + 1].hasReturned))"
            }
        }
    }
}
```

3.2 Experimental Validation with an Example System

This section attempts to answer the first research question by experimentally validating the approach outlined in this chapter. This experiment uses a non-trivial system that is modeled using the UML and will be implemented using the Java programming language. The section starts with a description of the system and follows with a description of the experiment.

3.2.1 The Client-Server System Design

The design concept of this example system is quite simple. It consists of two components, a client component and a server component. These two components communicate over a network to perform a simple commerce transaction. This kind of example was chosen because it is one of the most ubiquitous cases where security and reliability of the components are critical. The Enterprise JavaBeans (EJB) Application Programming Interface (API) package is used to implement the functionality of communication between the two components. The Secure Sockets Layer (SSL) protocol is employed as the means to establish a secure connection. The specific context of the system is an on-line business which sells vacation packages to customers who wish to conduct the transaction over a network.

In a real-world setting, even the simple system described above would be quite complex and beyond the practicality of a simple illustrative example. To make it a plausible system for the demonstration of the principles of this thesis, it has been simplified as much as possible without loosing its essential character. The following points of simplification are listed below to clarify the distinction between a real-world

system and this 'toy' system:

1. Instead of using SSL records, as required by the full SSL specification to communicate between client and server, the system simply uses method calls to accomplish this task. For example, the client, in order to start the handshake protocol, calls a method with the following signature: HelloMessage helloMessages(HelloMessage serverHello); This method supplies a HellowMessage object to the server and returns a HellowMessage object to the client.

2. The simplified handshake protocol does not verify authenticity of both parties by using certificates. The system assumes that the two parties are already trustworthy.

3. There is no negotiation for choosing encryption and hashing methods. The chosen method of encryption is DES because this method is automatically provided by the security API in Java. Similarly, the method for hashing is assumed to be MD5. During handshaking, when keys are exchanged, this is accomplished by the straightforward RSA methods. All of these functions are provided by existing API's in Java.

4. The full complement of derived keys is not required. After the exchanging all the required information in the handshaking protocol, the client and server are able to generate a single master key that they both can then use for symmetric encryption.

Figures 3.1, 3.2, 3.3, 3.4 and 3.5 are the UML views that were given to the student subjects in the experiment as part of their system design specification. There is a separate Class Diagram for the server and client components. These diagrams show only the associations relevant to the students for the parts of the system they needed

to develop. This allows the diagrams to be less cluttered and easier to assimilate than fully developed ones. The Sequence Diagrams again only specify object interactions that were needed for the assigned task.

3.2.2 Applying the Rules

Applying Rule 1 : Operation Access

Careful examination of the Sequence Diagrams reveals the operation access ambiguities. A comparison of the Sequence Diagrams shown in Figures 3.4 and 3.5 shows that the *AdminPanel* class and the *PaymentPanel* class both use the method $getSecureTransaction()$ from the $clientGUI$ instance. This is also true of the $getInstance()$ method from the *Cypher* class, the $init()$ method from the *Cypher* class and the $SealedObject()$ constructor from the *SealedObject* class. However, only the *AdminPanel* class uses the method $sendPriceChange()$ from the $secTransaction$ instance and only the *PaymentPanel* class uses the method $sendPaymentInfo()$ from the same instance. An examination of the Class Diagrams from Figures 3.1 and 3.2 gives no hint of these operation access patterns and thus allows a measure of ambiguity in the system design specification.

The operation access algorithm eliminates this inconsistency between the diagrams by producing OCL statements that constrain access to allowed classes only. Algorithm 1 creates a set of classes that are allowed to call each method. Each set is initially empty. The algorithm systematically adds classes that are allowed to access a method. After applying Algorithm 1, the following constraint is created for the $getSecureTransaction()$ method:

$context\ ClientGui :: getSecureTransaction()$

$inv : oclSet = \{AdminPanel, PaymentPanel\}$

$inv : oclSet- > includes(source.type)$

Similarly, the other methods mentioned above that are used by both the AdminPanel class and the PaymentPanel class have the following associated constraint statements:

$contextCypher :: getInstance()$

$inv : oclSet = \{AdminPanel, PaymentPanel\}$

$inv : oclSet- > includes(source.type)$

$contextCypher :: init()$

$inv : oclSet = \{AdminPanel, PaymentPanel\}$

$inv : oclSet- > includes(source.type)$

$contextSealedObject :: SealedObject()$

$inv : oclSet = \{AdminPanel, PaymentPanel\}$

$inv : oclSet- > includes(source.type)$

Constraint statements for those methods that are only used by one other class are shown below:

$contextSecureTransaction :: sendPaymentInfo()$

$inv : oclSet = \{PaymentPanel\}$

$inv : oclSet- > includes(source.type)$

$contextsecureTransaction :: sendPriceChange()$

$inv : oclSet = \{AdminPanel\}$

$inv : oclSet- > includes(source.type)$

When both passes of the algorithm have been completed, a constraint statement will exist for each method that is present in any Sequence Diagram.

Applying Rule 2: Composition

According to rule 2, Class Diagram compositions should be consistent with Sequence Diagrams. Figure 3.1 does not contain a composition. At first glance, we cannot tell if such a relationship exists in the Sequence Diagrams. However, if we apply Algorithm 2 we find that a composition relationship exists between the $TravelSession$ class and the $SecureTransaction$ class. All $SecureTransaction$ instances are created by $TravelSession$ instances and the life span of a $SecureTransaction$ instance never exceeds that of a $TravelSession$ instance. However, the Class Diagram gives no indication of this relationship.

Algorithm 2 systematically observes the life line relationship between every object that creates (using a constructor) another object. If we traverse Sequence Diagrams 3.4 and 3.5 we find that $TravelSession$ objects always create $SecureTransaction$ objects and outlive them as well. When applying Algorithm 2, we create a candidate composition when the first "TravelSession, SecureTransaction" pair interaction takes place. When we finish traversing the Sequence Diagrams we find that there were no relationships between the $TravelSession$ and $SecureTransaction$ objects that violated the composition relationship. Thus, we add to the $SecureTransaction$ class an OCL statement that describes the relationship. Algorithm 2 recognizes the composition inconsistency and addresses it by producing the following constraint statements:

$context SecureTransaction$

$inv : TravelSession.AllInstances- > notEmpty()$

The constraint statement asserts that in the context of a message instance, at least one instance of the container class (Account) must exist whenever a message instance exists. With the addition of this constraint, the system is now assured that stray transaction message instances will never be created outside of the proper composition relationship, thus adding to the security consistency of the system.

Applying Rule 3: Multiplicity

The typical scenario for conducting transactions with the example system involves only one transaction per customer at a given time. The Sequence Diagrams in Figures 3.4 and 3.5 illustrate this behavior. In each case, the *paymentPanel* or the *adminPanel* object initiates a transaction by obtaining the handle to the instance of $SecureTransaction$ by calling the method $getSecureTransaction$. When the customer has finished making his purchases, he logs out and the instance of $SecureTransaction$ that was used is then deleted by invoking $closeSecureTransaction$. Thus, there is never more than one $SecureTransaction$ instance associated with the active instance of $TravelSession$ at a given time. However, the Class Diagram allows a multiplicity of many, whereas the Sequence Diagram clearly calls for behavior that is considerably more constrained. Here again, an inconsistency between the Class Diagram and the Sequence Diagrams exists, allowing behavior that might be exploited by the malicious user.

As mentioned in the definition, the multiplicity constraint algorithm is straightforward. When applied to the example, the algorithm tracks each $SecureTransaction$

object in the class diagram. In the diagram we see that only one $SecureTransaction$ instance is created for any instance of $TravelSession$. After applying it to a single Sequence Diagram, it will produce the following constraint statement for the $SecureTransaction$ class:

$context SecureTransaction$

$inv : self.AllInstances()$-¿$size() <= 1$

If applied to the entire system, the algorithm will overestimate the constraint limit because it won't account for the destruction of $SecureTransaction$ instances after completion of a session. Again, in a more rigorous form, the algorithm could account for instance destruction and obtain a more accurate constraint limit.

Applying Rule 4 : Sequence of Operation

The algorithm for generating sequence of operation constraints is fairly straightforward. This section will demonstrate the results of applying the algorithm on the Sequence Diagram shown in figure 3.3. In this diagram an instance of the ClientGUI class invokes four methods from the TravelSession instance. The algorithm finds this sequence and creates the following OCL statements:

context $TravelSession$:

Let $SS_0 : oclSet$ = Sequence{hello_messages(), server_key_exchange(),
client_key_exchange(), finished() }

$inv : SS_0 -> forAll(SS_0[p]$.hasReturned implies
($SS_0[p-1]$.hasReturned and not $SS_0[p+1]$.hasReturned))

The algorithm finds no other sequences of method calls from a client candidate object instance. Thus, only the above constraints are produced for the doHandshake Sequence Diagram.

3.2.3 Experiment Overview and Context

The general context of the experiment may be characterized as one where multiple tests or trials are conducted on a single object. The object is a product specification accompanied by a partially complete software implementation. The subjects are computer science students at Washington State University who have enrolled in a computer security course. The subjects were motivated to complete the task and perform a reasonably good effort to produce a quality product because the assignment was graded and weighted enough to significantly affect the final course grade.

The student subjects were divided into two equal groups in order to introduce an independent variable into the experiment. The control group was given a product specification that did not include any OCL constraint statements. The remainder of the subjects received a specification that included four sets of OCL constraints corresponding to the four areas of security consistency focus described in this thesis. The task assigned to each subject was to complete the unimplemented parts of the system, guided by the accompanying design documents included in the product specification.

Experiment Design

Here the design of the experiment is exposed. The section begins by defining the critical terms used in the experiment analysis.

Goal definition template - The experiment analyzes the effect of including constraint statements in a design specification for the purpose of assessing the resulting

application code with respect to security consistency and robustness from the perspective of a researcher.

Null Hypothesis statement - The inclusion of constraint statements in a software specification does not affect the security robustness of the resulting application.

Alternate Hypothesis Statement - If a software design specification includes specific constraints about behavior boundaries, the finished product will contain a higher level of security robustness.

The independent variable - A software design specification

Treatment 1 - the specification does not include constraint statements

Treatment 2 - the specification does include constraint statements

The dependent variables - The scores derived from unit tests performed on the resulting code from each trial. Four different unit tests are used to generate four unique dependent variables for each trial.

Subjects - The subjects are students enrolled in a combined undergraduate/ graduate level computer security course.

Objects - There is only a single object used by the subjects of the experiment. This is the product specification or design document set. This object then becomes the independent variable, as explained by the treatments definitions above.

Context Analysis

The overview of the context was described above, but this section further characterizes the context of the experiment with a brief description of the following four dimensions:

1. The experiment was *off-line* because it was conducted in a controlled classroom environment. However, the subjects were not directly observed as they accomplished their given task. Thus, the level of control of all possible independent variables was not iron-clad. The experiment depended to some extent upon voluntary conformance to rules of conduct verbally delivered at the time of introducing the task. Consultation with other subjects within the same group was allowed with the stipulation that independent code was generated for final submission. Collaboration between subjects not in the same group was forbidden.

2. The subjects were *students* whose selection could be characterized as random. Their selection is random in that all students enrolled in the course were included in the experiment and the students on the enrollment list should represent a random sample of the general population of all students in the computer science major. None of the subjects would have been affected by any predispositions about the experiment because it was not part of any published curriculum for the course.

3. The system to be developed has characteristics of both *real* and *toy problems*, but the stronger resemblance is that of a *toy problem*. The functionality is common in real world systems, but the system is simplified to the point that it becomes more of a toy problem than a real system. This will become obvious in the system description that follows later.

4. Subject selection was specific because the subjects were not representative of the entire population of WSU students.

Validity Analysis

Figure 3.6 below represents the various principles in this experiment. By examining the constructs, it is possible to analyze the different threats to validity. [10]

On the top, we have the theory area, and on the bottom, the observation area. To properly analyze the validity of our experiment, we want to draw conclusions about the theory encompassed by the hypothesis, based on our observations. In drawing these conclusions we look at four steps, in each of which there is one type of threat to the validity of the results.

1. Conclusion validity.

   A considerable threat to the conclusion validity is that not all subjects completed the task assigned to them. Of the 6 subjects in the constraint group, only three implemented the constraints. Of the 5 subject in the control group, only three were complete and functional enough to test. The effect on the experiment is that the number of usable trials from which to draw conclusions shrinks considerably. However, the trend illustrated by the unit tests is so strong that even a low number of usable trials should not invalidate the conclusions drawn.

   Conclusion validity can be compromised if the application of the treatments are unreliable or inconsistent. In the case of this experiment, the subjects could have very different experience and understanding of the OCL statements. To address this, a short lecture on general principles of OCL was given to all subjects and a specific lecture focusing on the OCL statements included in the design documents was given to the OCL group only.

2. Internal Validity.

   Internal validity is concerned with whether the conclusions or outcomes of an

experiment are truly caused by the treatments. Several threats of this kind are discussed here. A few specific threats to causality were anticipated. These were addressed proactively to avoid the harmful effects. Other threats were perceived after the results were analyzed. The only possible compensation for these cases is to make a note of these issues and take care to weigh them when making conclusions.

It became evident that different experience and skill levels could skew the results within groups. The list of subjects included both undergraduate and graduate level students. To compensate for this, the mix in each group after a random selection produced the two groups was examined. Minor adjustments were made to assure that the ratios of undergraduate to graduate students was approximately equal in both groups.

Another threat to causality is the possible communication of subjects across group boundaries. Because the subjects were not observed during their implementation activities, they were theoretically unhindered from causing strong outcome influences by collaborations with friends on other groups. To address this threat, the presentation of the task included specific mention of the threat to validity inherent in these collaboration. The authority of a student-teacher relationship and the motivation of a good grade for the assignment was expected to discourage these kinds of specifically illegal interactions.

After the data was analyzed, some weaknesses in the instrumentation were discovered. The constraint statements did not always specify a context that directed the subjects to implement the constraint code in the server side. Consequently, the constraint code arbitrarily resided in either the client or server code. Because the unit test code used to test for constraint compliance was in-

serted in the client code, it could often be designed to circumvent the constraint code. This caused the pass or fail outcome of some of the unit tests to become potentially somewhat arbitrary. This potential threat to internal validity was minimized by using unit tests that were as uniform as possible for each subject, as opposed to custom created for each one. This assured a consistent test for each subject and reduced the likelihood of arbitrary outcomes.

3. Construct Validity.

   The experiment contains only one independent variable. It is therefore subject to mono-operation bias or an under-represented cause construct. However, the difference in the intent of a design document which includes constraint statements versus one which does not is quite recognizable and dramatic. Thus, the representation of the cause construct with a single variable is not as vulnerable to the mono-operation bias.

   In order to reduce the likelihood of hypothesis guessing, which might skew coding behavior, careful attention was paid to the presentation of the task assignment. Because the context of the course was software security, the hypothesis for the experiment was related to the course subject, but the underlying principles were not directly studied. No course lecture material was devoted to pointing out the inherent ambiguities in UML design views. This means that the subjects were less likely to guess the intent of the experiment.

4. External Validity.

   External validity is concerned with the ability to generalize the experiment results to industrial practice. This experiment contains some concerns that merit discussion. First, the effective number of samples is considerably reduced

by the fact that some of the subjects did not complete the assignment sufficiently to yield test results. The experiment would be stronger with a larger class of students to enlist for the test.

Second, there is some concern that the nature of the *toy problem* might be too far removed from the complexity of a real industrial scale problem. In the case of this experiment, however, the coding behavior is not expected to change very much with scaling of the exercise. The testing of this assumption might be a good basis for further investigation.

3.2.4 Experiment Results

The student subjects submitted the results of their implementation activities as Java project folders. The processing of these projects required embedding of the unit tests within the Graphical User Interface (GUI) of each system. The instrumented GUI's were then executed and the individual unit tests were invoked manually one by one by clicking on the appropriate GUI button.

Each unit test was designed to test the implementation of OCL constraints for a single dependent variable. For example, "the Access Control Test" button is used to invoke the unit test that attempts to violate the *access control* constraints. The unit test reports the result of the test with an output statement, which is then recorded manually. The compiled results of all the unit tests are presented in the following table.

Results Analysis and Conclusions

Processing of the project submissions revealed some minor flaws in the administration of the experiment. The design specifications given to the OCL group included the

	Access Implem. ?	Result P/F	Multipl. Implem. ?	Result P/F	Compos. Implem. ?	Result P/F	Sequence Implem. ?	Result P/F
OCL Group								
subject1	yes	P	yes	P	yes	P	yes	P
subject2	yes	P	yes	P	yes	P	yes	P
subject3	yes	P	yes	P	yes	P	yes	P
subject4	no	F	no	F	no	F	no	F
subject5	incom	NA	incom	NA	incom	NA	incom	NA
subject6	incom	NA	incom	NA	incom	NA	incom	NA
Totals		3/3		3/3		3/3		3/3
Control Group								
subject7	no	F	no	F	no	F	no	F
subject8	no	F	no	F	no	F	no	F
subject9	incom	NA	incom	NA	incom	NA	incom	NA
subject10	incom	NA	incom	NA	incom	NA	incom	NA
subject11	no	F	no	F	no	F	no	F
Totals		0/3		0/3		0/3		0/3

Table 3.1: Unit Test Results

OCL statements, but did not specify the proper system response to attempts to violate an OCL constraint. This resulted in some variety in the manner that constraints were implemented. Some systems only reported on events that violated constraints, but did not prevent the illegal behavior. Others actually prevented the attempted abuses and threw Java exceptions that reported the attempt. This response variety did not harm the outcome of the experiment. It only caused the measurement of the results to be more difficult. Because of the response variety, the classification of the outcome of a test as Pass or Fail had to be verified by the test researcher and could not be automated across all tests.

Because of the lack of variance in the data, it becomes quite evident that a statistical analysis of the numbers will not yield any insights that aren't already discernible from simple observance. However, for the sake of completeness, the formal methods of hypothesis testing will be presented here. As discussed above, the experiment contains one factor, or independent variable, and two treatments. This design category

logically leads to the *t-test* for parametric hypothesis testing. As prescribed in [10], the calculations for the test appear below:

The four dependent variables embodied in the unit tests yield four results for each of two independent samples. Because the results are always the same for each dependent variable, the calculations use the data from a single variable. The data results are converted to numerical values by the arbitrary assignment of 2 for a passing unit test and a value of 1 for a failing unit test. This yields the following two independent sample sets:

$x = \{2, 2, 2\}$ for the OCL group

$y = \{1, 1, 1\}$ for the control group

The null hypothesis, H_0, states that the inclusion of constraints in the design specification does not affect the security robustness of the end product, as judged by unit test results. In terms of the sample data above, the null hypothesis becomes: $\mu_x = \mu_y$, i.e. the expected mean values of the sample sets are the same.

The lack of variance in the data makes the calculations trivial.

$S_x^2 = 0$

$S_y^2 = 0$

$S_p = 0$

$t_0 = \infty$

The number of degrees of freedom is:

$f = 3 + 3 - 2 = 4$ yielding:

$t_{0.025,4} = 2.776$

Since $|t_0| > t_{0.025,4}$, it is possible to reject the null hypothesis with a two tailed

test at the 0.05 level.

Conclusions from the results are readily apparent. Even in the face of the fact that the number of experiment trials, represented by submitted systems, was reduced significantly by incomplete projects, the remaining good trials indicate strong trends. The data demonstrates decisively the following statements of conclusion:

1. All systems that implemented constraints either reported or prevented behavior outside the boundaries defined by the constraints. Thus, the security robustness of the system is always positively affected. This statement strongly disproves the null hypothesis for the experiment because the inclusion of constraints in the design always affected the system security robustness.

2. No systems that did not implement constraints prevented such behavior. Therefore, strong evidence is demonstrated that without the inclusion of specific constraints in a design document, developers are unlikely to be motivated to add the higher level of quality and security robustness to their systems that are achieved by using design constraints.

3. By including such constraints in the design document, the designer can guarantee that his finished system will achieve a higher level of security robustness. This statement confirms the alternate hypothesis statement for which the experiment was designed.

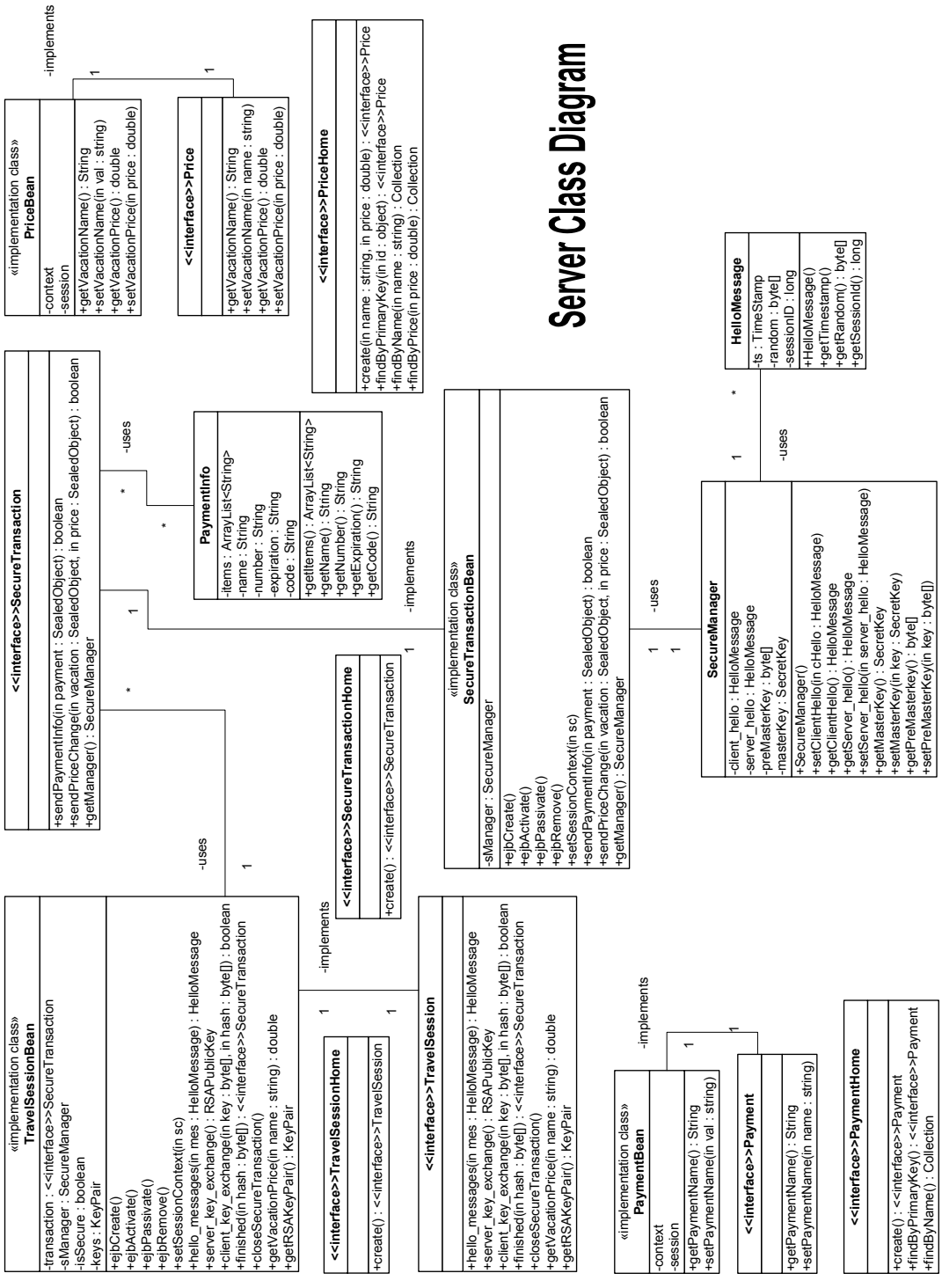

Figure 3.1: Class Diagram for the Server Side

Figure 3.2: Class Diagram for the Client Side

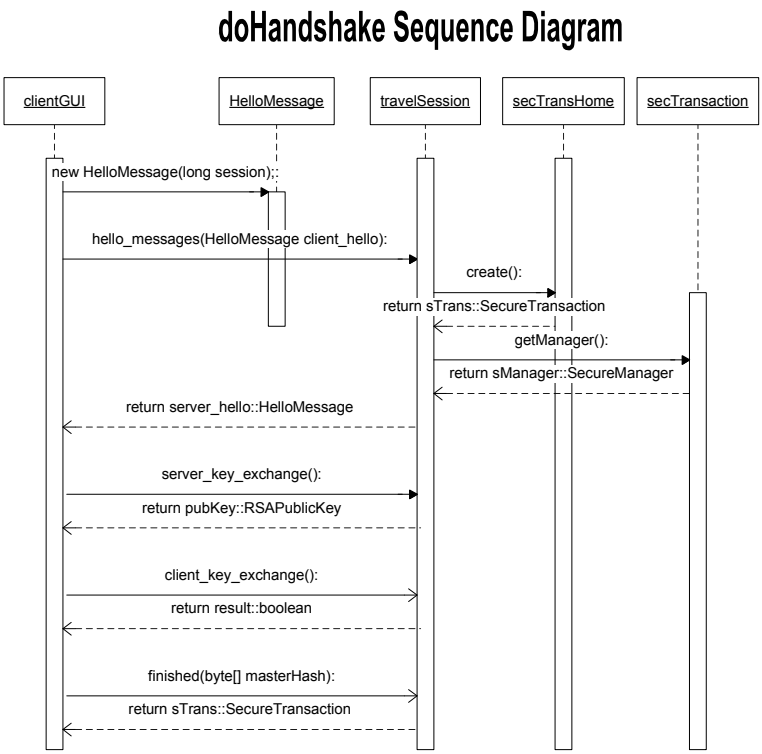

Figure 3.3: Sequence Diagram for the doHandshake method

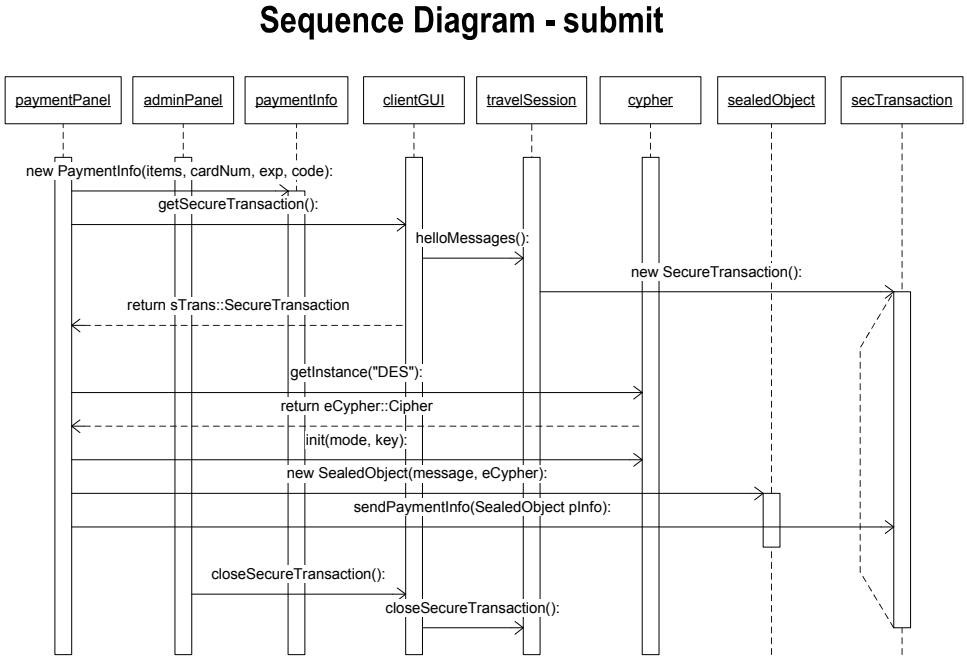

Figure 3.4: Sequence Diagram for the submit method

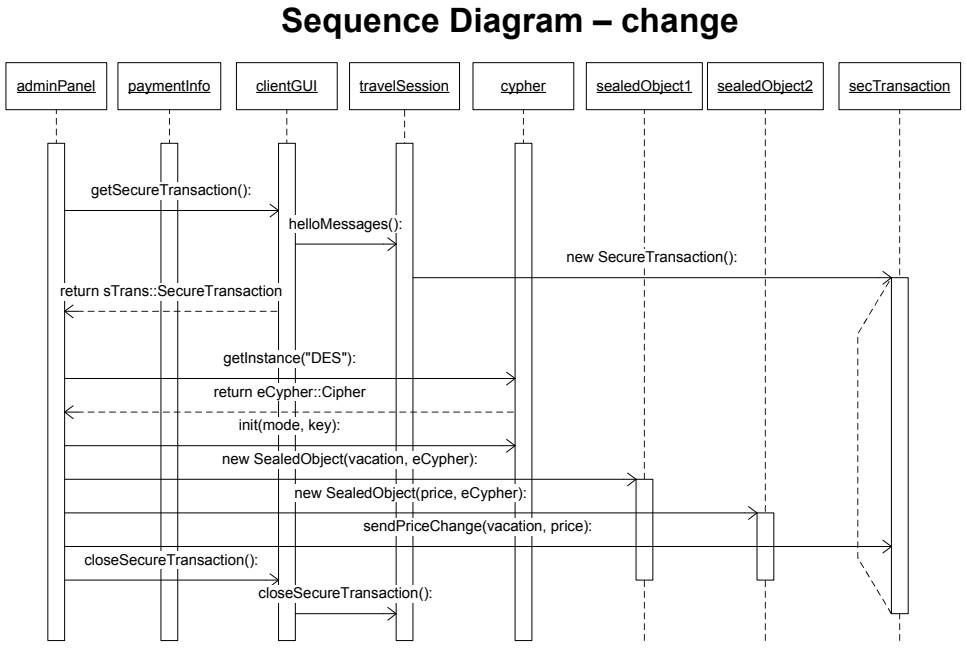

Figure 3.5: Sequence Diagram for the change method

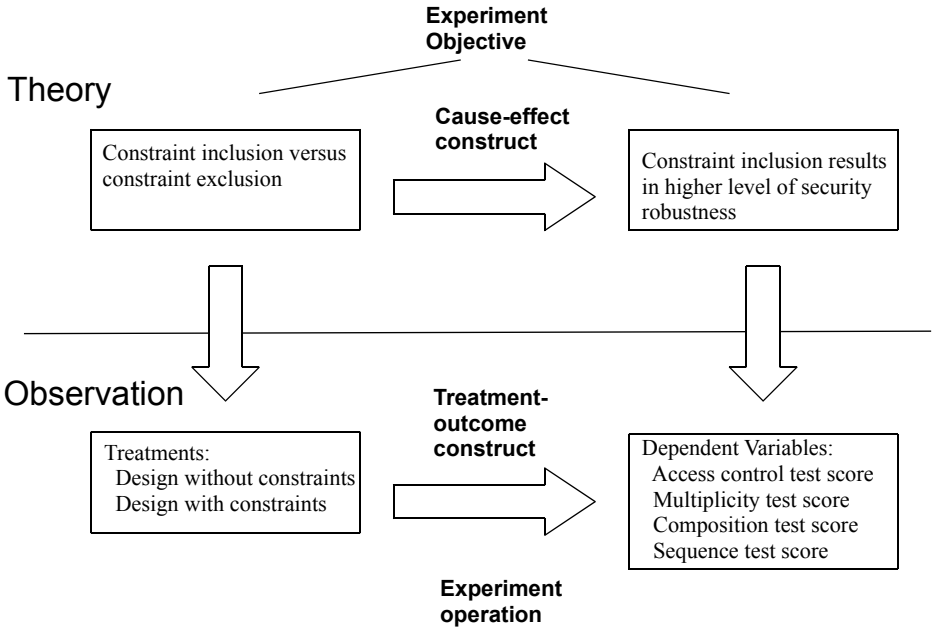

Figure 3.6: Experiment Principles for Client-Server Example

Chapter 4

UML Design Metrics for Quality Enhancement

4.1 Introduction

Software design increasingly includes component based software development [17]. This happens from two perspectives: (1) a software designer wants to use components and needs to define how components fit with the remainder of the design. (2) a software designer wants to design components for reuse. This could be part of a product line architecture or part of a set of components developed for reuse. In either case, one needs to determine all component boundaries and interfaces before implementation. Often quality, and more specifically maintainability, are the important drivers when making design decisions concerning component boundaries. Arisholm et al. [11] showed that dynamic coupling and cohesion metrics are linked to software quality. The goal of this chapter is to define dynamic coupling and cohesions measures for classes and components that can be used in the design phase to evaluate quality of the components based on these metrics.

The approach considered here considers both structural and behavioral aspects of coupling and cohesion. This requires (1) combining class diagrams and sequence diagrams into a model that reflects both structure and behavior, and (2) executing this model based on an expected operational profile to collect measures.

Section 4.2 describes the use of an integrated UML model to evaluate components. This section defines cohesion and coupling metrics based on an integrated model and an operational profile. Section 4.3 illustrates the method on an example. Section 4.4 describes the results of an empirical investigation correlating design metrics to code quality and maintainability metrics. Conclusions and suggestions for further study are deferred until the final summary for the thesis.

4.2 Component Evaluation Approach

During design in Component Based Software Engineering (CBSE), the designer has to define components and component interfaces. Often there are choices and ideally all components should have high cohesion and low coupling for quality and maintainability purposes. Candidate components are evaluated based on their cohesion and coupling. When designing a component based system using the UML, cohesion and coupling measures are computed for each candidate component and they form the basis for evaluating candidate components and for choosing between them. Because cohesion and coupling metrics use information from multiple UML views, it is necessary to combine them into an integrated design model.

Pilskalns et al. [21] provide a testing model that transforms Sequence Diagrams into an *Object Method Directed Acyclic Graph* (OMDAG). This approach uses test-cases to traverse different paths in the graph. Here the same approach is adapted to component evaluation. The adaptation involves replacing test-cases with an op-

erational profile, adding Class Diagram information (structural information) to the OMDAG, and collecting metrics for cohesion and coupling. The approach to component analysis consists of the following steps:

1. Build an integrated model (OMDAG) of Class Diagrams and Sequence Diagrams.

2. Identify candidate components in the integrated model.

3. Define an operational profile (work load).

4. Execute the operational profile (traverse paths in graph) and collect metrics.

5. Compare candidate components and make decisions.

Sections 4.2.1 through 4.2.5 describe each step. Figure 4.1 outlines the approach.

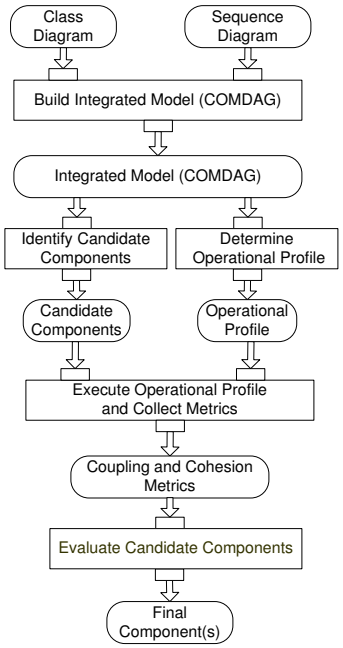

Figure 4.1: Component Evaluation

4.2.1 Building the integrated model (COMDAG)

Building the integrated model consists of three steps. The first step maps Class Diagrams into tuples, called Class Tuples (CT). A class tuple is a mathematical representation of a class and is similar to the idea of representing a class using the XMI specification. The second step consists of mapping Sequence Diagrams into an Object Method Directed Acyclic Graph (OMDAG). The third step consists of combining each OMDAG with the CT information. This results in the CT Object Method Directed Acyclic Graph (COMDAG). The COMDAG represents the integrated model that combines Class Diagrams and Sequence Diagrams.

The Class Tuple (CT)

The CT consists of a class name, attributes (represented as tuples if non-primitive) from the class and super classes (if applicable), and methods (represented as tuples) for the class and super classes (if applicable). Classes may contain non-primitive attributes that are defined by other classes. Thus, CTs may contain other CTs by definition. A Class Tuple of a class (c), the Attribute Tuple, the Method Tuple, and the method's Parameter Tuple have the following forms:

(4.1) $CT(c) = \langle \{\langle\, Parent\ CT \,\rangle\},\ \{\langle\, Attribute \,\rangle\},\ \{\langle\, Method \,\rangle\}\,\rangle$

(4.2) $Method = \langle\, method\ name,\ return\ type,\ \{Parameter\}\rangle$

(4.3) $Attribute = \langle\, attribute\ name,\ attribute\ type\,\rangle$

(4.4) $Parameter = \langle parameter\ name,\ parameter\ type\rangle$

Any or all of the tuple elements can have null values denoted by a null place holder for that element. The *Parent CT* has the same structure as the CT, thus part of the CT is recursively defined.

The Object Method Directed Acyclic Graph (OMDAG)

The OMDAG maps the dynamic information in a Sequence Diagram to a directed acyclic graph. The OMDAG is created by mapping object and sequence method calls from a Sequence Diagram to vertices and arcs in a directed acyclic graph. The mapping between Sequence Diagram and OMDAG preserves the relationships in the Sequence Diagram. The mapping consists of (1) associating methods in the Sequence Diagram with their originating objects, (2) traversing the Sequence Diagram for the purpose of mapping successive method executions to edges of the OMDAG. These edges are also annotated with any conditions the Sequence Diagram may impose on their execution.

The OMDAG is a tuple $\langle V, E, s \rangle$ where V is a set of vertices, E is the set of edges, and s is the starting vertex. Each vertex, v, is defined by the tuple $v = \langle o, m, \{ARGS\}, c \rangle$, where o is an object, m is the method or return call, $ARGS$ is a set of arguments, and c is a class name. Arguments are the actual method parameters that are used in a Sequence Diagram. The actual parameters may have values associated with them. The $ARGS$ tuple is defined with the following triple: $\langle$ *type, name, value* $\rangle$. Note that only the class name is known in the Sequence Diagram; details about the class are not available. An edge, E, is represented by the tuple $\langle v_i, v_{i+1} \rangle$. For details on mapping the Sequence Diagrams to an OMDAG see [21].

The CT Object Method Directed Acyclic Graph (COMDAG)

The final step in building an integrated model is to combine OMDAG and CT information. The COMDAG is built by replacing each class name, c, in each OMDAG

vertex v with the corresponding CT(c). This results in an expanded vertex definition:

(4.5) $v = \langle\, o,\, m,\, \langle ARGS \rangle,\, CT(c) \,\rangle$

4.2.2 Defining Components in the Model

A COMDAG consists of a set of vertices, $v_1 ... v_z$. Each vertex v, as defined in Equation 4.5, contains a class c. A candidate component consists of a proper subset of the vertices in a COMDAG and the classes associated with those vertices. The subset of vertices is called the CV set. Vertices of the CV set may or may not be connected to other vertices in the set. To define a component, select vertices from the COMDAG and place them into a CV set. The class set that is associated with the CV set is called the component class (CC) set. The CC can be defined as follows:

(4.6) $CC = \bigcup_{\forall v \in CV} \{c \mid c \, in \, v\}$

The technique does not automate the process of defining components. Rather it provides the designer with metric feedback once candidate components have been selected. The process of deciding which vertices to include in a component CV set should use domain knowledge. The edges in the COMDAG gives the designer some indication of how classes are coupled, hence placing vertices in a CV that create a large number of *boundary edges* may not be a wise choice. Typically a designer looks for natural boundaries where there is a low number of connections between groups of vertices. A designer typically chooses a set of vertices that has a minimal amount of connections with the rest of the graph. Thus a min-cut algorithm could be employed, if one wanted to automate the generation of a CV. The result of this step is a set of candidate components with their associated CV and CC sets.

4.2.3 Defining an Operational Profile

An operational profile exercises a system under the conditions in which it is expected to operate [20]. In other words, the system is exercised with a suite of test cases and their frequency that represents how the system will be used in practice. In [21], executing a test case results in executing a path through the graph. Thus we can describe an operational profile in terms of a set of paths through the COMDAG and their frequency.

An operational profile is recorded in a table consisting of a path definition (as a sequence of nodes) and the number of times (frequency) the path should be traversed. The designer determines the number of traversals and which paths represent how the system will be used based on domain knowledge and/or requirements (Use-Cases).

4.2.4 Executing and Collecting Metrics

Operational profile execution consists of traversing the COMDAG using the path and frequency specified in the operational profile. While traversing the COMDAG, the following metrics are collected:

The RFC Metric in Designs

The RFC of a class is the cardinality of the set of all method invocations that may be executed in response to a message received by an object of that class. The RFC for designs measures the cardinality of the response set for a class by traversing paths in the COMDAG. Let $P = \{v_1...v_n\}$, where P is a path in the COMDAG and v_i $(i = 1...n)$ is a vertex defined by Equation 4.5. A vertex, v_i, in path P, contains a method call m_i. Then there exists a vertex v_j $(j > i)$, which corresponds to a return call for m_i. $MA(m_i)$ is the set of methods activated by m_i before its return. The

methods activated set (MA) is defined as:

(4.7) $MA(m_i) = \{m_k \mid i < k < j, v_k \; in \; P\}$

Then the class response set $CR(c_n)$ for all methods m in class c_n is defined as follows:

(4.8) $CR(c_n) = \bigcup_{\forall m \; in \; CT(c_n)} MA(m)$

The c_n represents the class in which the method, m, resides (m is used to index the MA sets). The $MA(m)$ may change each time m is invoked due to conditional statements within the method.

The set of all methods for a class, c_n, can be defined as follows:

(4.9) $M_{CT}(c_n) = \{m \mid m \text{ in method tuple of CT}(c_n)\}$

The following equation yields the RFC for a class:

(4.10) $RFC_{COMDAG}(c_n) = \mid CR(c_n) \bigcup M_{CT}(c_n) \mid$

The RFC can also be calculated for a component. This can be done by treating all methods in the component as if they belonged to the same class. Thus when adding methods to a component's response set $CR(CC)$, all methods in the CR sets belonging to the classes of the component are added. The $CR(CC)$ is defined as follows:

(4.11) $CR(CC) = \{ \bigcup_{\forall c_i \in CC} CR(c_i)\}$

The same applies to the M_{CT} resulting in a set that includes methods from all of a component's classes. The $M_{CT}(CC)$ is defined as follows:

(4.12) $M_{CT}(CC) = \{ \bigcup_{\forall c_i \in CC} M_{CT}(c_i)\}$

The component RFC is the cardinality of the union of the $CR(CC)$ and $M_{CT}(CC)$ sets. The RFC is defined as follows:

(4.13) $RFC_{COMDAG}(CC) = \mid CR(CC) \bigcup M_{CT}(CC) \mid$

The RFC for designs is an indirect, absolute measure and can be used to compare both classes (Equation 4.10) and components (Equation 4.13) in a system.

The OMMIC Metric in Designs

The OMMIC measures the coupling for a class by counting method calls to other classes that are not in its inheritance hierarchy. The OMMIC for designs is measured by traversing paths in the COMDAG. The paths are traversed with a frequency indicated in the operational profile to simulate a workload. The set of all paths can be defined as $P = P_1...P_y$. Let $P_t = \{v_1...v_n\}$, where v_i is a vertex defined by Equation 4.5. Vertices $\langle v_i, v_{i+1}\rangle, i = 1...n-1$ define the edges in the COMDAG. The vertex, $v_i = \langle o_i, m_i, \langle ARGS\rangle, CT(c)\rangle$ in path P_t contains an object o_i, a method m_i, the method arguments, $ARGS$, and a class tuple $CT(c)$. The vertex, $v_{i+1} = \langle o_{i+1}, m_{i+1}, \langle ARGS\rangle, CT(d)\rangle$ in path P_t is defined similarly. The CT contains all class information including parent class information. We can define the set of all classes in vertex v_i as follows:

$$classes(v_i) = \{e \mid e \text{ occurs in inheritance structure of } c, \text{ where } v_i \text{ contains } c\} \tag{4.14}$$

The OMMIC metric relies on distinguishing if a method call between vertices v_i and v_{i+1} is invoked by a class within the same inheritance hierarchy. To make this determination, the OMMIC metric needs to define the Boolean function *other* :

$$other(v_i, v_{i+1}) = \begin{cases} 1, d \notin classes(v_i) \\ 0, otherwise \end{cases} \tag{4.15}$$

The OMMIC for a class c is summed over all paths in an operational profile. Different paths may have differing method calls, thus interaction between classes may differ with each path. Summing over all paths includes all interactions in a class. The frequency, f_t, is associated with a path P_t. The frequency indicates how often that path is traversed. The frequency, f_t, is used as a weight to give classes with higher

usage a higher coupling value. The OMMIC for designs is defined as follows:

$$(4.16)\ OMMIC_{COMDAG}(c) = \sum_{t=1}^{y}(f_t * \sum_{i=1}^{n} other(v_i, v_{i+1})), \quad \forall v_i \textit{ containing } c$$

This means that the OMMIC of c is increased every time a method call is made to a class outside of the inheritance hierarchy of c.

The OMMIC can be calculated for a component CC with classes $c_1 ... c_n$ by modifying Equation 4.15 to include all classes in the same component. The $classes_c$ set for components can be defined as follows:

$$(4.17)\ classes_c(v_i) = \bigcup_{\forall c_i \in CC} \{e \mid e \text{ occurs in inheritance structure of } c_i, \text{ where } v_i \text{ contains } c_i\}$$

The $classes_c(v_i)$ creates a set of all classes defined in a component, where the class in v_i is a member of the component.

The ICP Metric in Designs

The ICP measures the coupling for a class by counting the number of calls to other classes in a design. The definition of the ICP requires an operational profile and the same definitions for paths, vertices, and CTs used in defining the OMMIC. The ICP metric relies on distinguishing if classes associated with vertices v_i and v_{i+1} are the same or different. If the classes are different, then the method call from v_i to v_{i+1} is external. External calls from a class indicate interaction with other classes, which is a commonly accepted definition of coupling. Thus, the definition of the Boolean function *external* becomes:

$$external(v_i, v_{i+1}) = \begin{cases} 1, d \neq c \\ 0, otherwise \end{cases}$$

The ICP for a class c can be calculated by summing over all paths in an operational profile. The ICP uses the number of method parameters, $ARGS$, in v_i as a weight. The frequency, f, is used as a multiplier to give classes with higher usage a higher coupling value. The ICP is defined as follows:

$$ICP_{COMDAG}(c) = \sum_{t=1}^{y}(f_t * \sum_{i=1}^{n} external(v_i, v_{i+1}) * \mid ARGS \mid), \quad \forall v_i \; containing \; c \tag{4.18}$$

This means that the ICP is increased for class c every time a method call is made to a class outside of c.

The ICP can be calculated for a component CC with classes $c_1 ... c_n$ by modifying Equation 4.18 to exclude all classes in the same component. The $external_c$ function for components is defined as follows:

$$external_c(v_i, v_{i+1}) = \begin{cases} 1, d \notin CC, \text{ where } c \in CC \\ 0, otherwise \end{cases} \tag{4.19}$$

The ICP design metric is indirect and uses attributes that are measured on an absolute scale.

The ICH Metric in Designs

The ICH measures the cohesion in a class by counting the number of method calls internal to a class. The definition of the ICH requires an operational profile and the same definitions for paths, vertices, and CTs used in defining the OMMIC. The ICH metric relies on distinguishing if classes associated with vertices v_i and v_{i+1} are the same or different. If the classes are the same then the method call from v_i to v_{i+1} is internal. Internal calls in a class indicate interaction within the class, which is a commonly accepted definition of cohesion. Thus, the definition of the Boolean

function *internal* becomes:

$$internal(v_i, v_{i+1}) = \begin{cases} 1, d = c \\ 0, otherwise \end{cases} \tag{4.20}$$

The ICH for a class c can be calculated by summing over all paths in an operational profile. The ICH uses the number of method parameters, $ARGS$, in v_i as a weight. The frequency, f_t, is used as a multiplier to give classes with higher usage a higher cohesion value. The ICH can be declared as follows:

$$ICH_{COMDAG}(c) = \sum_{\forall P}(f_t * \sum_{i=1}^{n} internal(v_i, v_{i+1}) * \mid ARGS \mid), \quad \forall v_i \; containing \; c \tag{4.21}$$

This means that the ICH is increased for class c every time an internal method is called. The ICH can be calculated for a component CC with classes $c_1 ... c_n$ by modifying the Equation 4.20 to include all classes in the same component. The $internal_c$ function for components is defined as follows:

$$internal_c(v_i, v_{i+1}) = \begin{cases} 1, d \in CC, \text{ where } c \in CC \\ 0, otherwise \end{cases} \tag{4.22}$$

The ICH design calculation is indirect and uses attributes that are measured using an absolute scale.

4.2.5 Component Evaluation

Metric collection produces values of the RFC, OMMIC, and ICP coupling metrics, as well as for the ICH cohesion metrics for every candidate component. There are various

ways in which these values can be used by a designer to either analyze candidate components for strengths and weaknesses or to select among component choices.

1. The designer considers the actual values for a design component and determines whether individual values are satisfactory or not. If not, the designer can contemplate redesign to either increase, or decrease specific values. This is the most subjective way of using the metrics.

2. The designer evaluates the metrics against previously agreed upon thresholds for each cohesion and coupling metric. When component choices are to be made, only components that fall within the thresholds are further considered. This states objective requirements for cohesion and coupling. On the other hand, it may take a certain amount of historical data to determine such thresholds. If only one candidate component is evaluated and does not meet all thresholds, the designer must redesign to improve inadequate values of these metrics.

3. Given that there are three coupling and one cohesion metric, there are four variables with which to make a decision. A candidate component may have high coupling and high cohesion, while another may have low coupling and cohesion. Each option is thus good in one area and less desirable in another. How should one choose between the two? In this case, preferences between cohesion and coupling have to be defined. It is possible to use Multicriteria Decision Making Techniques [14] on the two groups of measures. Either lexicographic ranking or the more sophisticated Analytic Hierarchy Process (AHP) [22] can be used.

Whether a designer uses simple thresholds or more involved decision making methods, is often up to the specific situation, how many choices exist and how similar or different the collected measures are. It is not possible to recommend one of these methods over another.

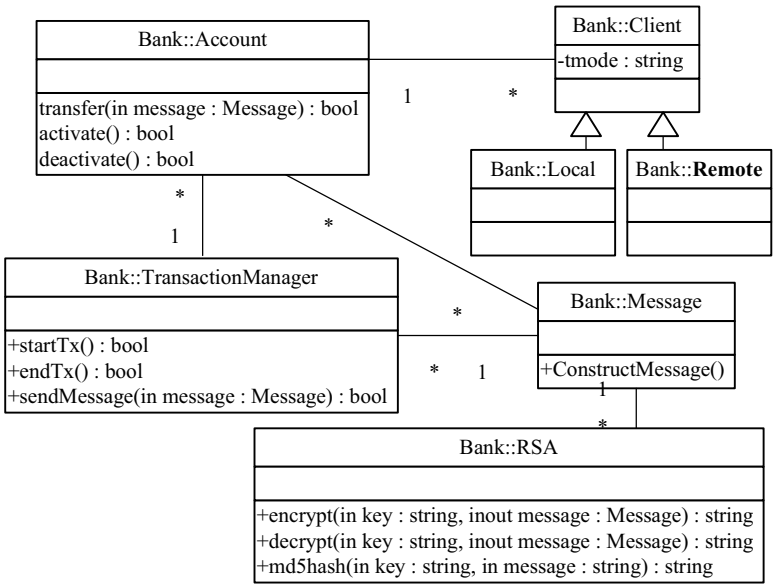

Figure 4.2: Banking System Class Diagram.

4.3 An Example

This example, adapted from [13], represents a design for a banking system. Figure 4.2 shows the Class Diagram and Figure 4.3 shows the Sequence Diagram. The diagrams outline the structural and behavioral aspects of a simplified banking system. The system contains two types of clients, a local client and a remote client. The clients interact with the system by sending messages. If the client is remote then the message needs to be encrypted and signed using an RSA encryption algorithm. Once the message is encrypted (if remote), it is transferred to the bank account using a transaction manager. The transaction manager can begin and end a transaction. A transaction such as a transfer is not finalized until the transaction manager indicates an end to the transaction.

The designer of the system wants to componentize the system for reuse purposes. The Class Diagram contains one generalization where the *Remote* and *Local* classes

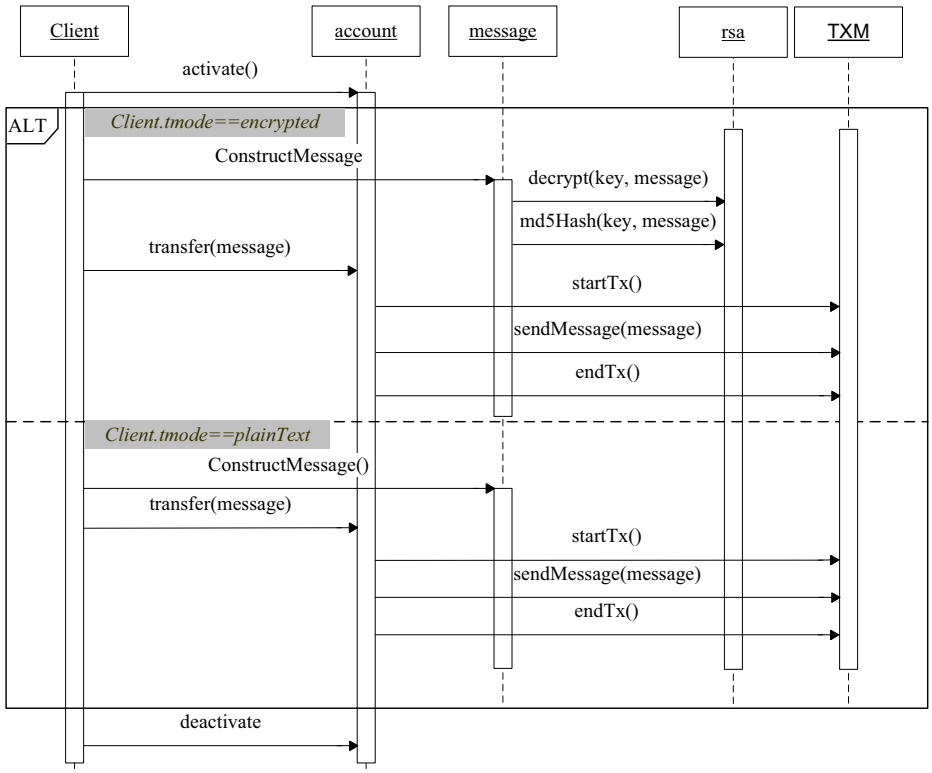

Figure 4.3: Banking System Sequence Diagram.

inherit from the *Client* class. The *RSA* class provides encryption (decrypt) and signing (md5hash). The *Account* class contains the information about a banking client's account. The *TransactionManager* class starts and stops transactions to provide protection against system failure during a transaction. If the transaction is not complete and the system fails, the transaction manager backs off the transaction. In this example, remote transactions occur 4 times as often as local transactions. Two scenarios are represented as two paths in the Sequence Diagram. One scenario shows a client interacting with a bank account locally and the other shows a client interacting remotely.

4.3.1 Build An Integrated Model

The first step of the approach is to build an integrated model by combining the Class and Sequence Diagrams into a COMDAG. Each class from Figure 4.2 is mapped into a CT according to the mappings outlined in section 4.2. The lower right side of Figure 4.4 contains the CT information. The Sequence Diagram in Figure 4.3 is traversed and mapped into the OMDAG according to the procedure in section 3. The information is combined into the COMDAG shown in the left side of Figure 4.4.

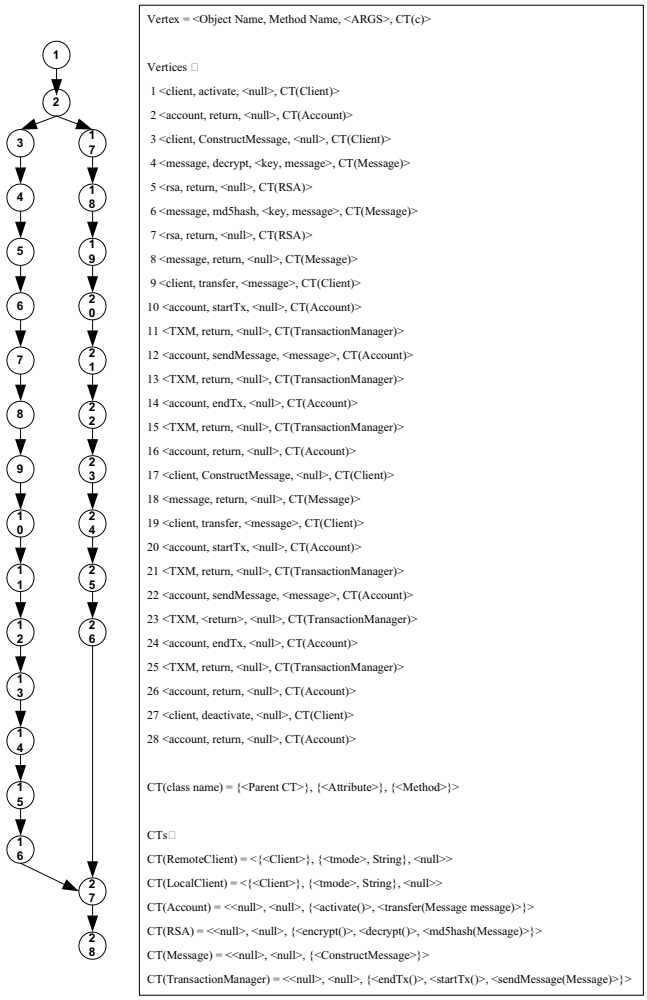

Figure 4.4: CT + OMDAG = COMDAG.

4.3.2 Define Components

The next step in the approach is to decide on candidate components. In this example the designer wants to see the difference between combining: (1) the messages with the encryption utility into component $COMP_1$ or (2) combine the messages with transaction management into component $COMP_2$. The candidate component set for $COMP_1$ is the set $CC_1 = \{Message, RSA\}$. The candidate component set for $COMP_2$ is the set $CC_2 = \{Message, TransactionManager\}$. By consulting the COMDAG we find that the first component vertex set consists of the following vertices: $CV_1 = \{v_4, v_5, v_6, v_7, v_8\}$. The second component vertex set consists of the following vertices: $CV_2 = \{v_4, v_6, v_8, v_{11}, v_{13}, v_{15}, v_{21}, v_{23}, v_{25}\}$.

4.3.3 The Operational Profile

The next step in the approach is to define an operational profile, based on the expected system operation. The COMDAG in Figure 4.4 contains two paths. According to the example description, remote transactions are 4 times as common as local transactions. This means the first path (vertices 3-16) of the COMDAG is executed 4 times more often. This results in the following operational profile for the banking application:

Table 4.1: Operational Profile for Banking Operation

Path Definition	**Frequency**
$P_1 = v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}, v_{12}, v_{13}, v_{14}, v_{15}, v_{16}, v_{27}, v_{28}$	4
$P_2 = v_1, v_2, v_{17}, v_{18}, v_{19}, v_{20}, v_{21}, v_{22}, v_{23}, v_{24}, v_{25}, v_{26}, v_{27}, v_{28}$	1

4.3.4 Coupling and Cohesion Metrics

The approach now evaluates the two potential components with respect to coupling and cohesion by applying the coupling and cohesion metrics to each of the candidate components. Due to the length of the computations, detailed steps for only the RFC metric are given.

The RFC Coupling Metric:

To calculate the RFC for $COMP_1$, equation 4.7 is applied. The $COMP_1$ class set is $CC_1 = \{Message, RSA\}$. The $M_{CT}(CC_1)$ set is calculated by placing all methods from the CC_1 into a set. This results in the following set:

$M_{CT}(CC_1) = \{$ConstructMessage, encrypt, decrypt, md5hash $\}$

Next, the approach executes the operational profile for each path P_1 and P_2 by visiting each vertex in the path. Each vertex contains a method, m_i. Method m_i belongs to the class, c_n in vertex v_{i+1}. Notice that in the COMDAG, a method is always a member of the class contained in the next node. Thus if v_1 and v_2 are two vertices in the COMDAG, then a method m_1 used in v_1 belongs to the class in v_2. The $MA(m_i)$ is the set of methods activated by m_i before its return. Table 4.2 illustrates the stepwise execution for calculating the RFC, where column 1 contains P_t, the current path, column 2 contains v_i, the currently visited vertex in a path, column 3 contains c_n, the class associated with m_i, column 4 contains m_i, the method call associated with the vertex v_i, and column 5 contains $MA(m_i)$, the set of activated methods. The table contains only non-empty $MA(m_i)$ sets in Path P_1. Path P_2 is not shown since it does not contain any non-empty $MA(m_i)$ sets that belong to $COMP_1$. Note that the only message activations not immediately followed by a return are in v_3 (ConstructMessage), with a return in v_8, and in v_9 (transfer) with a return in the v_{16}. This results in the following non-empty MA sets:

$MA(ConstructMessage) = \{$decrypt, md5hash $\}$

$MA(Transfer) = \{$startTx, sendMessage, endTx$\}$

Table 4.2: RFC calculations

P_t	v_i	c_n	m_i	$MA(m_i)$
P_1	v_1	Account	activate	
P_1	v_2		return	
P_1	v_3	Message	ConstructMessage	
P_1	v_4	RSA	decrypt	
P_1	v_5		return	MA(ConstructMessage) = decrypt
P_1	v_6	RSA	md5Hash	MA(ConstructMessage) = decrypt
P_1	v_7		return	MA(ConstructMessage) = decrypt md5Hash
P_1	v_8		return	MA(ConstructMessage) = decrypt md5Hash
P_1	v_9	account	transfer	MA(ConstructMessage) = decrypt md5Hash,
P_1	v_{10}	TXM	startTX	MA(ConstructMessage) = decrypt md5Hash
P_1	v_{11}		return	MA(ConstructMessage) = decrypt md5Hash, MA(transfer) = startTx
P_1	v_{12}	TXM	sendMessage	MA(ConstructMessage) = decrypt md5Hash, MA(transfer) = startTx
P_1	v_{13}		return	MA(ConstructMessage) = decrypt md5Hash, MA(transfer) = startTx, sendMessage
P_1	v_{14}	TXM	endTx	MA(ConstructMessage) = decrypt md5Hash, MA(transfer) = startTx, sendMessage
P_1	v_{15}		return	MA(ConstructMessage) = decrypt md5Hash, MA(transfer) = startTx, sendMessage, endTx
P_1	v_{16}		return	MA(ConstructMessage) = decrypt md5Hash, MA(transfer) = startTx, sendMessage, endTx

The next step involves the calculation of the $CR(c_n)$ for each c_n. This is the union of the MA sets associated with methods of class c_n. For $COMP_1$ the *CR(Message)* is calculated by taking the union of all MA sets for each method in the *Message* class. Since there is only one method, $ConstructMessage$, the calculation is trivial. The CR for each class in $COMP_1$ is as follows:

$CR(Message)$ = {decrypt, md5hash }

$CR(RSA)$ = empty

The response set $CR(COMP_1)$ is calculated by taking the union of the $CR(c_n)$ sets, where c_n belongs to CC_1. The following set results:

$CR(COMP_1)$ = {decrypt, md5hash }

The union of the $M_{CT}(COMP_1)$ set and the $CR(COMP_1)$ set results in the following set:

$M_{CT}(COMP_1) \bigcup CR(COMP_1)$ = {ConstructMessage, encrypt, decrypt, md5hash}

The cardinality of this set is the RFC for component $COMP_1$, which is equal to 4.

Component $COMP_2$ is calculated in the same manner. The $COMP_2$ class set is $CC_2 = \{Message, TransactionManager\}$. The $M_{CT}(CC_2)$ set is calculated by placing all methods from CC_2 into a set. This results in the following set:

$M_{CT}(CC_2)$ = { startTx, endTx, SendMessage, ConstructMessage }

The response set CR for component $COMP_2$ is the union of the *CR(Message)* set and *CR(TransactionManager)* set. The *CR(Message)* set has already been calculated in $COMP_1$. The *CR(TransactionManger)* is empty since there are no nested method-calls made from within the class in either P_1 or P_2. The CR for each class in $COMP_2$ is as follows:

$CR(Message)$ = {decrypt, md5hash }

$CR(TransactionManager)$ = empty

The union of the M_{CT} and CR sets is

$M_{CT}(COMP_2) \bigcup CR(COMP_2)$ = { startTx, endTx, SendMessage, ConstructMessage, decrypt, md5hash }

The cardinality of this set (the RFC) for component $COMP_2$ is 6. We can see that for $COMP_2$ the CR set was not a subset of the M_{CT} set, thus it added to the number of members in the set, resulting in a higher RFC value.

The remaining metrics are calculated and can be found in Table 4.3.

4.3.5 Evaluation

The last step is to evaluate which component has better coupling and cohesion measures. In this case simple inspection of the values for cohesion and coupling metrics was all that was needed, since one component choice outperformed the other for all metrics, making the use of AHP unnecessary. According to Table 4.3 $COMP_1$ is the clear winner. Component 1 has a RFC value that is less than that of component 2. The primary difference in the OMMIC, ICP, ICH values for each component can be attributed to calls being external in component 2 and internal in component 1.

Metric	Component 1	Component 2
RFC	4	6
OMMIC	0	8
ICP	0	16
ICH	16	0

Table 4.3: Component Metric Summary

4.4 From Design Metrics to Maintainability

The approach presented in this chapter rests on the assumption that there is a correlation between cohesion and coupling metrics during design, and maintainability and quality of the resulting implementation. It has been established elsewhere that certain code coupling and cohesion metrics are correlated with quality and maintainability. Thus, all that remains is to show that the design level coupling and cohesion metrics employed by the approach of this chapter are correlated with their code counterparts. Then it becomes justifiable to use them to determine future maintainability and quality at design time. To this end, the chapter presents an empirical study and asks the question: is there a correlation between design metrics and code metrics? The study is designed with one factor (the target of metric collection) and two treatments (de-

sign or code). It analyzes a design and its corresponding implementation for the purpose of assessing their metrics with respect to their values from the perspective of a researcher.

The design consists of an annotated UML Class Diagram and Sequence Diagrams that describes a software package that can Gouraud Shade Polygons. The Class Diagram contains 15 classes and the Sequence Diagram describes their interaction. Annotations included potential inputs, which describe the format of a polygon. The output of the design is a 300x300 ppm image containing a shaded polygon.

The experiment was conducted in a classroom environment. The subjects were 10 students in a Senior Software Engineering course at Washington State University. The UML design was created based on a real world problem found in computer graphics. The project was part of a graded test in their Senior Software Engineering Class. They were asked to implement the design using Java. They were not given a time constraint. The dependent variables were the RFC, OMMIC, ICP design and code metrics. All students created working implementations of the design. Data was collected from each design and coded class. The data demonstrated that the *Pearson Product Moment Correlation* (PPMC) was greater than .95 in every case. All students created a central driver class based on the design. Figure 4.4 shows the results of the findings for the driver class.

This study clearly shows that there is a positive correlation between code and design metrics. It is also evident that there is a constant offset between the data sets. This might be explained by the fact that designs generally have less detail, specifically regarding method calls, compared to the actual code. One of the threats to validity is the completeness of the design. The design of this experiment was very detailed and complete. In practice, designs may be incomplete or very simplistic. In addition, the size of the implemented programs were only around 1000 lines of code. Industrial size

Design Metrics				
	RFC	OMMIC	ICP	
	10	14	14	
Code Metrics				
	RFC	OMMIC	ICP	PPMC
student1	24	33	36	0.970725
student2	24	35	38	0.979076
student3	24	35	39	0.966282
student4	25	35	38	0.975417
student5	23	33	36	0.975417
student6	23	33	36	0.975417
student7	22	33	36	0.979076
student8	23	33	36	0.975417
student9	25	35	38	0.975417
student10	24	35	39	0.998461

Table 4.4: Empirical Study Data

software is obviously much larger. These concerns and issues are certainly grounds for future studies.

Chapter 5

Conclusions and Future Work

5.1 Summary and Significance

This body of work has attempted to address two closely related research questions. Each question deals with a very specific technique for enhancement of quality in software products. The techniques attempt to accomplish their goal by analyzing aspects of the design concept as captured in UML design views. The overall aim of this approach is to enhance software quality more efficiently by investing extra effort in the design phase instead of in the product testing phase.

One technique addressed research question (1) by proposing an algorithmic technique for generating OCL constraints to address inconsistencies and ambiguities in UML diagrams. The validation experiment indicated that a developer is unlikely to address these deficiencies without the presence of the constraint language in the design specification. However, adding the constraint language achieves a very high likelihood that the design deficiencies will be addressed and eliminated.

A second technique addressed research question (2) by using a metric gathering technique. The metrics measured coupling and cohesion in the design views in order

to predict these same qualities in the implemented product. The metrics are used as an early indicator of maintainability of the end product, giving the designer an up-front opportunity to identify weaknesses prior to initiating implementation.

Both of the techniques described in this thesis have the same goal. They are tools to aid the software system designer improve the quality of his design before risking the resources reserved for the implementation phase of the development cycle. The central assumption of this kind of approach is that quality improvement efforts are many times more efficient in the design phase than they are in the later phases of the cycle.

5.2 Future Work

All of the techniques presented in this document have been discussed as conceptual approaches for accomplishing the goal of efficient quality enhancement of software products. All of the validation efforts used to test the hypotheses of these ideas were conducted by manually employing the proposed techniques. The ultimate aid to efficient quality enhancement in the design phase would be comprised of tools that automate the techniques and remove the drudgery of using them. Future work could certainly focus on opportunities to create some of these tools.

Bibliography

[1] G. Ahn and M. Shin. "Role-based Authorization Constraints Specification Using Object Constraint Language", *Proceedings of the 10th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pp 157-162, 2001.

[2] M. Alam, R. Breu, M. Breu. "Model-Driven Security for Web Services", *Proceedings of the 8th International Multi-topic Conference (INMIC 2004)*, pp 498-505, 2004.

[3] E. Fernandez-Medina, M. Piattini, M.A. Serrano, "Specification of security constraint in UML", *IEEE International Carnahan Conference on Security Technology*, Oct. 16-19, pp. 163-171, 2001.

[4] M. Gogolla, J. Bohling, and M. Richters, "Validation of UML and OCL Models by Automatic Snapshot Generation", *Proceedings from the 6th International Conference on the UML*, pp. 265-279, Oct 20–24, 2003.

[5] M. Graff, *Secure Coding: Practice and Principles*, O'Reilly, 2003.

[6] S. Mellor, K. Scott, A. Uhl, D. Weise, *MDA Distlled: Prinicples of Model-Driven Architecture*, Addition Wesley, 2004.

[7] J. Warmer, A. Kleppe, *The Object Constraint Language, 2nd Edition*, Addison-Wesley, 2003.

[8] Object Management Group, "UML 2.0 Specification", http://www.omg.org/uml, 2006.

[9] O. Pilskalns, A. Andrews, R. France, S. Ghosh, "Rigorous Testing by Merging Structural and Behavioral UML Representations", *Proceedings from the 6th International Conference on the UML*, pp 234-248, Oct 20–24, 2003.

[10] Wohlin C., Runeson P., Host M.k Ohlsson m., Regnell B., Wesslen A., *Experimentation in Software Engineering*, Kluwer Academic Publishers, 1999

[11] E. Arisholm, L. Briand, A. Foyen, "Dynamic Coupling Measurement for Object-Oriented Software, IEEE Transactions on Software Engineering, pp. 491-506, August 2004.

[12] V. Basili, L. Briand, W. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators", IEEE Transactions on Software Engineering, pp. 751-761, October, 1996.

[13] R. Binder, *Testing Object-Oriented Systems Models, Patterns, and Tools*, Object Technology Series, Addison Wesley, Reading, Massachusetts, 1999.

[14] P. Bogetoft, Peter Pruzan; *Plannning with Multiple Criteria*, North-Holland, 1991.

[15] L. Briand, J. Wuest, Empirical Studies of Quality Models in Object-Oriented Systems, Advances in Computers, Academic Press, vol. 56, 2002.

[16] S. Chidamber and C. Kemerer, "A Metrics Suite for Object-Oriented Design", Information and Technology, Vol 35, No5, pp 232-240, 1991.

[17] G. Heineman, W. Councill, *Component–Based Software Engineering: Putting the Pieces Together*, Addison–Wesley, Boston, MA, 2001.

[18] Lee, Y. -S., Liang, B. -S., Wu, S. -F., and Wang, F. -J., "Measuring the Coupling and Cohesion of an Object-Oriented Program Based on Information Flow", *International Conference on Software Quality* pp. 81-90, 1995.

[19] W. Li, and S. Henry, "Object-Oriented Metrics that Predict Maintainability", *Journal of Systems and Software*, 23(2), pp 111-122, 1993.

[20] J. Musa. *S*oftware Reliability Engineering McGraw-Hill, New York. 1999.

[21] O. Pilskalns, A. Andrews, R. France, S. Ghosh. "Rigorous Testing by Merging Structural and Behavioral UML Representations", *UML Conference 2003*, pp. 234-248, 2003.

[22] T. Saaty.*The Analytical Hierarchy Process*, McGraw-Hill, 1990.

[23] N. Fenton and S. Pfleeger, *Software Metrics - A Rigorous and Practical Approach, Second Edition*, PWS Publishing Company, 1997.

[24] M. Fowler and K. Scott, *UML Distilled Second Edition*, Addison-Wesley, 2000.

[25] D. Harel, A. Pnueli, J.P. Schmidt, and R. Sherman. "On the formal semantics of statecharts". *In Proceedings of the 2nd IEEE Symposium on Logic in Computer Science*, pp 54-64, Ithaca, New York, June 1987.

[26] C. Jard, S. Pickin, "COTE - Component Testing Using the Unified Modeling Language", ERCIM News, No. 48, pp 49-50, Jan 2002.

[27] J. Kontio, "A Case Study in Applying a Systematic Method for COTS Selection", *Proceedings of the 18th International Conference on Software Engineering*, Berlin, Germany, March 25-30, 1996, pp. 201-209, Los Alamitos, CA: IEEE Computer Society Press, 1996.

[28] J. Kontio, "OTSO: A Systematic Process for Reusable Software Component Selection", CS-TR-3478, 1995, University of Maryland Technical Reports, University of Maryland. College Park, MD.

[29] N. G. Lester, F.G. Wilkie, and D.W. Bustard, "Using UML to Categorise and Specify Criteria for Reusable Artefacts", *UML '98 Beyond the Notation - International Workshop (Preliminary Proceedings)*, pp. 19-24, 1998.

[30] Object Management Group, "UML 2.0 Draft Specification", http://www.omg.org/uml, 2003.

[31] A. Ritzhaupt "Object-Oriented Design Metrics Using UML Class Diagrams" Second Annual CCEC Symposium, pp 221-233, Spring 2004.

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

Appendix A

List of Acronyms

1. CBO - Coupling Between Object classes
2. COMDAG - Class(tuple) Object Method Directed Acyclic Graph
3. DIT - Depth of Inheritance
4. FOL - First Order Logic
5. ICH - Information-flow-based Cohesion
6. ICP - Information-flow-based Coupling
7. LCOM - Lack of Cohesion Metric
8. MDA - Model Driven Architecture
9. NOC - Number of Children
10. OCL - Object Constraint Language
11. OMDAG - Object Method Directed Acyclic Graph
12. OMG - Object Management Group
13. OMMIC - Other Method-Method Import Coupling
14. OSCL - Object Security Constraint Language
15. PPMC - Pearson Product Moment Correlation
16. RFC - Response for Class
17. SSL - Secure Socket Layer
18. UML - Universal Modeling Language

19. WMC - Weighted Method per Class
20. XACML - Extended Access Control Markup Language