Analyzing and Measuring Reusability in Object-Oriented Designs

Margaretha W. Price
MountainNet, Inc.
2816 Cranberry Square
Morgantown, WV 26505-9289 USA
+1 304 594 9075 ext. 28
mprice@rbse.mountain.net

Steven A. Demurjian, Sr.
The University of Connecticut
Computer Science & Engrg. Dept.
191 Auditorium Rd., U-155
Storrs, CT 06269-3155 USA
+1 860 486 4818
steve@eng2.uconn.edu

ABSTRACT

In this paper, we present a technique to analyze and measure the reusability of object-oriented (OO) designs. The metrics can be incorporated into a design/development environment, so that reusability measurements, analysis, and improvements can be part of "business as usual" for an organization. Design reusability measurements also enable early identification of poor reuse potential, when it is still possible to modify/refine the design. The essential components of our approach are two reuse-specific characterizations of classes and hierarchies, and a set of metrics which objectively measures the dependencies among design components based on those reuse-specific characterizations.

1 INTRODUCTION

Software components that are reused most often tend to be small components, since they are normally less specific (i.e., string functions, abstract data types (ADT), or utility routines), and thus, more likely needed by other systems. However, small code reuse produces minimal savings representing only a small percentage of the final product [2]. Poulin argues that there are three classes of software that make up a typical software application [20]:

- **Domain-independent** (20% of the whole application): This includes ADTs, utility routines, math libraries and other components which are useful in a wide range of problem areas.
- **Domain-specific** (65% of the whole application): This is for software which is only useful within the specific domain. The examples given include high-speed communications device drivers, navigational aids for aircraft, and financial services libraries.
- **Application-specific** (15% of the whole application): This includes software which implements the unique details of an application.

From the above breakdown, we can expect the most savings if we reuse the domain-specific software. Software companies do not have to make their software be reusable in all systems, but they only have to make their software reusable in anticipated future systems in their organizations. Our approach to reusability measurement facilitates domain-specific reuse, particularly, domain-and-organization-specific reuse. This more restrictive goal of reusability makes domain analysis more manageable, thereby minimizing the impact of the actual domain on the potential reuse.

The design of a program is normally described in terms of the program's components and the interactions among them [16]. Our reusability metrics measure the level of interactions of software design components which are expected to be reused together to the components that comprise the rest of the system. Most of the implementation details are not specified in software designs, thus the software designer has a significant amount of flexibility in modifying portions of an existing design to accommodate future systems, thereby attaining design reuse.

Figure 1 illustrates an OO design/development process which incorporates design reusability measurements. A software engineer starts the process by designing a system. After the major components of the system (and the interactions between them) have been determined, our metrics can be used to identify, measure, and provide feedback on the reusability of the OO design. The software designers then have the opportunity to modify their OO design and reevaluate the reusability of their system. The two arrows (1) can be repeated as many times as necessary, in an iterative process that is intended to make the OO design more complete. After the reusable portions have been clearly identified and it has been determined that there exists no interactions which inhibit reuse, we can store the design and document the system architecture (2). Step (3) leads to the implementation process (object coding, testing, and debugging) and the storing of the completed implementation (4). Future software projects will then have the choice to reuse a previous OO design (5) and its corresponding implementation (6) or just reuse the design and write a new implementation. In some cases, it may be necessary to revisit earlier stages of this process after the implementation has commenced.

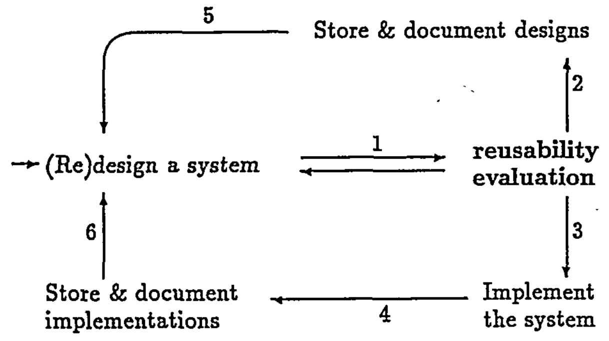


Figure 1. Design and Implementation Process

Design reusability measurements are important for two reasons. First of all, measurements can be automated and they can quickly provide feedback to the software designers. The metrics can also be incorporated into a design and development environment, so that reusability measurements, analysis, and design improvements can be part of "business as usual" for an organization. The second reason for design reusability measurements is to enable early identification of poor reuse potential, when it is still possible to modify/refine the design.

The remainder of this paper is organized into six sections. In Section 2, we present the background concepts for our conceptual model that supports OO design-level reuse. In Section 3, we detail the framework of the OO design-level measurements. Section 4 contains an empirical study that demonstrates our approach via a design reusability evaluations tool that we have developed. In Section 5, we review ongoing research in metrics theory against our approach as presented in Sections 2 and 3. Section 6 examines related work in the areas of reusable, hierarchical, OO models and in reusability measurements. Finally, Section 7 contains the conclusions and a note on our effort to incorporate these measurements into a design/development environment.

2 BACKGROUND CONCEPTS

The major components of our design reuse metrics are two subjective (designer-defined) characterizations of classes and hierarchies. Because of the intellectual nature of the software design process, important components of it must be measured subjectively, while the tangible representation of the design product can be measured objectively for many purposes [4]. Our two characterizations of classes and hierarchies, which are presented in Sections 2.2 and 2.3, are the important reusability properties of a software design, hence they are to be defined subjectively by the software designers. After the characterizations have been defined, we provide a framework in Section 3 which objectively measures the dependencies among design components. These dependencies are the tangible representation of the design product, thus they are measured objectively. This section starts by identifying the unit of abstraction that is used in our reusability measurements.

2.1 Unit of Abstraction

Research efforts in OO metrics [3, 5] are mostly concerned with 'good-design' criteria at the class level. For reusability evaluations, we believe it is more appropriate to evaluate the criteria at the class hierarchy level of abstraction and to study the reusability of a class hierarchy as a whole, portions of a class hierarchy, or a set of related class hierarchies. This has also been argued in another context, namely that the unit of abstraction for OO applications should not only be at the class/object type level, but also at the class hierarchy level [7].

A class hierarchy is the result of an OO mechanism referred to as *inheritance* or *class derivation*. Class inheritance allows members (functions and data) of one class (parent) to be used as if they were members of another class (child or subclass). During system design, class hierarchies are used to group similar classes so that they can have one parent class containing the common operations and/or data. The subclasses will then only need to define operations/data specific to each subclass. Thus, a class is only made to be a child of another class if it needs some members of the parent class. Consequently, it is the nature of OO design with inheritance to migrate more general information and operations up the hierarchy where they can be reused by all descendants while *simultaneously* pushing domain-specific information and operations down the hierarchy where their potential reuse is limited.

From a reuse perspective, since a child class needs members of its parent, if we want to reuse a child class we have to also reuse the parent class. However, if we want to reuse the parent class, we are not required to reuse its subclasses, since the parent class does not use members of its subclasses. As a result, new systems can reuse the top portion of a hierarchy or the whole hierarchy, but they cannot reuse just a lower part of a hierarchy. Reusing just the top portion of a hierarchy is desirable in many cases, since the lower level classes are more specific classes, so with respect to their parents, they are less likely to be needed in other applications.

2.2 General versus Specific Classes

From our discussion in Section 2.1, it is clear that the individual classes in any inheritance hierarchy can be characterized based on their overall position. In our approach, we require the software designer to identify individual classes to be either General or Specific with respect to its purpose in the overall design. A *General* class is one that is expected to be reused in other applications. A *Specific* class is a class that is only applicable in this application. Abstract classes, which are a design technique used to define templates for specifying subclasses, are examples of classes which would normally be defined as General classes. However, not all General classes have to be defined as abstract classes; rather, a class is General if it is recognized by the software designer as being able to solve problems in addition to the context (inheritance hierarchy and underlying application) that it is defined within.

To further explain this characterization, we use a simple software design of a Health Care Application (HCA) system [13]. Figure 2 presents the class hierarchies of HCA. The Person hierarchy is used to represent and process information common to all people in a hospital application. The Record hierarchy is used for various record processing, the Item hierarchy is used to represent general physical items used in a hospital, and the Organization hierarchy is used to represent the various entities in a hospital.

This Health Care Application system however, is not only applicable in hospitals. Portions of this system might

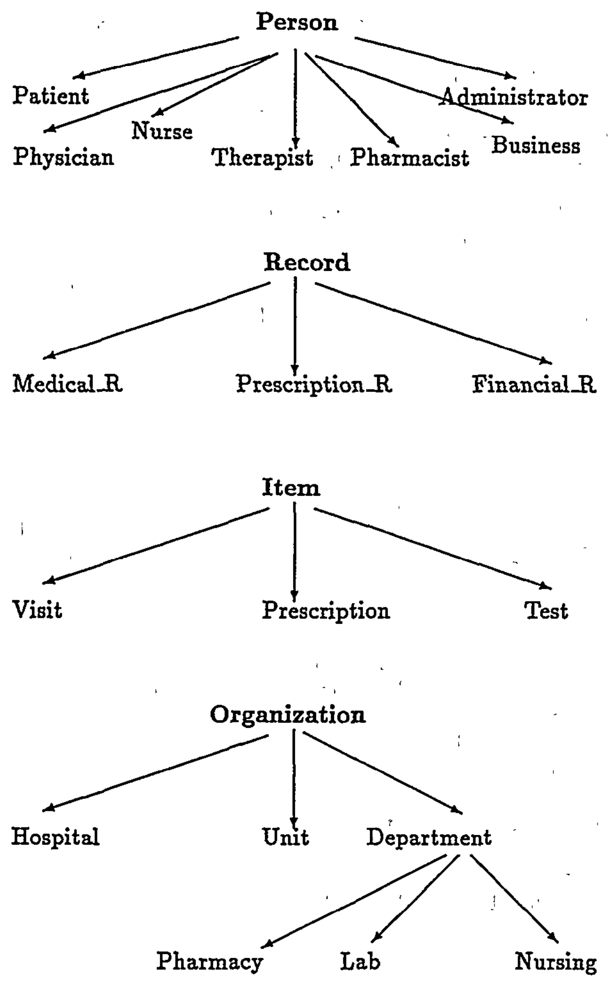


Figure 2. Health Care Application (HCA) Classes

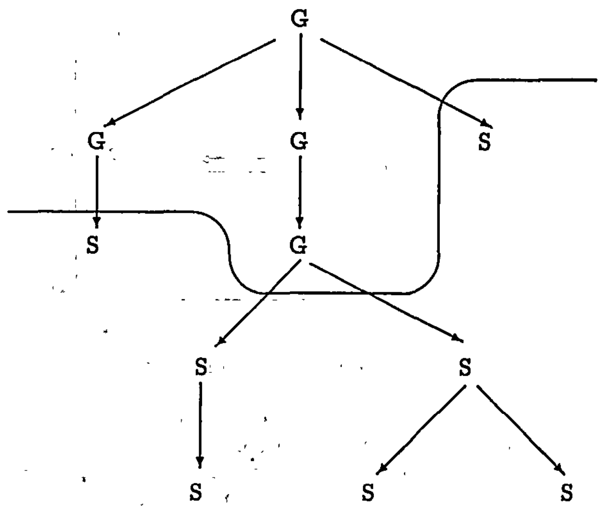


Figure 3. General(G)/Specific(S) Classes of a Hierarchy

be reusable in other facets of health care, both in large and small scales (e.g., dental office, eye care center, etc.). If our organization's future projects are expected to target those smaller health care establishments, we can define the following classes as General classes: Person, Patient, Physician, Business, Record, Medical_R, Prescription_R, and Financial_R. The other classes can be defined as Specific classes, since it is likely that they will not be needed in our future systems.

As stated previously, we cannot reuse just a lower part of a hierarchy. Thus, as illustrated in Figure 3, the General classes must be towards the top and the Specific classes must be towards the bottom of the hierarchy. A General class cannot be a descendant of a Specific class, since to reuse a class, we also have to reuse its parents. A design tool can be used to enforce this: when creating a root class, this class is first defined as a General class. If at some point of a hierarchy, a class is defined as a Specific class by the software designer, descendants of that Specific class, will also have to be Specific. Thus, every class hierarchy has a line that divides all of the General classes from all of the Specific classes.

It has been known that the generalization of classes is an important software development activity, and should be included as an integral part of a software development process. Generalizations arise in systems in two ways, during the initial design phase and during the maintenance phase. Work in program restructuring [9, 10, 15] discusses generalizations in terms of structural modifications during the maintenance of completed applications. On the other hand, our metrics bring up the issue of generalizations during the initial design process in an effort to maximize the reusable portions of applications..

2.3 Related Class Hierarchies

While General and Specific classes provide a characterization mode within an inheritance hierarchy, from a design-reuse perspective, it is also important to identify the interactions between the hierarchies that comprise an OO application. These interactions provide the important first step in discerning the couplings between classes when viewed from the perspective of entire hierarchies. Thus, to augment the General/Specific classes, the software designer is asked to define the class hierarchies that are related to one another in an OO application. A hierarchy is defined as *related* to another hierarchy if they are related in concept and are expected to be reused together in future systems. Relating class hierarchies encourages the designers to group their components into reusable portions at the earliest stages in the design process.

OO frameworks [17] is a similar technique in codifying design knowledge to produce a generic design. Like OO frameworks, related hierarchies also provide a means to describe the interactions between objects of a program. OO frameworks are domain specific, hence they require the software designer to have a solid understanding about the application domain. In determining related hierarchies, it is also important for software designers to understand the application domain, but more importantly, they need to have some ideas on the kinds of systems they expect to build in the future. Thus, the determination of related hierarchies is not only domain specific, but also organization specific.

In the HCA design given in Figure 2, a software designer may decide that the hierarchies with root classes 'Person' and 'Record' will always be needed together in future projects. The reason can be that in any health care system, we will always need to represent the people involved and to process patients' records. Moreover, current or future subclasses of Person will be the ones that manipulate records. In this case, the software designer knows that there are (will be) many couplings between these two hierarchies, but these couplings will not affect the reusability of HCA, or portions of HCA, in this organization's future projects. In our metrics, these two hierarchies can be defined as related to each other.

A dependency to a related hierarchy is not a hindrance to reuse, because the related hierarchy will also be reused together; thus, the dependency will always be satisfied. On the other hand, a *dependency to an unrelated* hierarchy is a hindrance to reuse and can arise later in design or implementation phase. This characterization of related hierarchies, if done during the initial design phase, can prevent the occurrence of unrelated dependency. Requiring software designers to define related hierarchies provides a way to differentiate between couplings that do affect reuse and those that do not.

3 A Design Reusability Measurement Framework

This section discusses the objective dependency measurements which are based on the subjective characterizations of classes and hierarchies described in Section 2. Unlike the characterizations of General/Specific classes and related hierarchies, the measurements presented here are domain-independent. The first subsection details the types of couplings between General and Specific classes, which are foundational in identifying those portions of the design that have the greatest reuse potential. The next subsection discusses couplings between hierarchies and provides suggestions on either eliminating them (if they are a hindrance to reuse) or moving them to locations where they can add value to the reusable design. The final subsection presents the design-reusability metrics which are based on subjective characterizations of Sections 2.2 and 2.3 and the objective concepts of Sections 3.1 and 3.2.

3.1 Coupling between General and Specific Classes

The first aspect of our design-reusability measurements involves an understanding of the different types of couplings that can exist between General and Specific classes, and their positive or negative impact on reuse. Figure 4 illustrates the four types of coupling between General (G) and Specific (S) classes of two hierarchies: a General class can depend on another General class (1), a General class can depend on a Specific class (2), a Specific class can depend on a General class (3), and a Specific class can depend on another Specific class (4). According to [5], inter-class coupling occurs when methods of one class use methods or instance variables of another class. Since the unit of abstraction used here is at the hierarchy level, we define coupling as inter-hierarchy coupling for when methods of one hierarchy use methods or instance variables of another hierarchy. There are actually eight types of couplings. The four couplings illustrated in Figure 4 can be either directed to a related

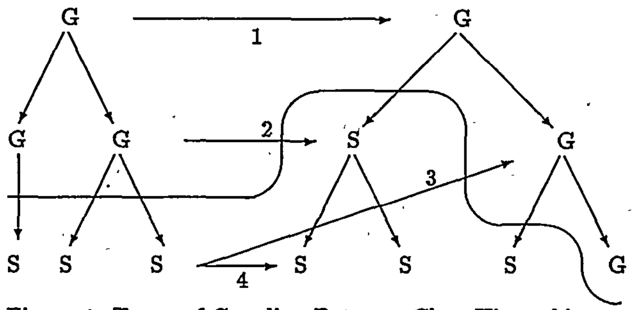


Figure 4. Types of Coupling Between Class Hierarchies

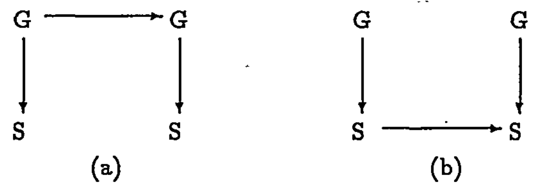


Figure 5. Favorable Couplings between: (a) Related Hierarchies and (b) Unrelated Hierarchies.

hierarchy or to an unrelated hierarchy. Our metrics measure these eight types of dependencies separately, as we will justify in Section 5.

Not all dependencies are bad for reuse. Dependencies between classes which are expected to be reused together (i.e., related hierarchies) are not a hindrance to reuse. In fact, they add more value to the design, because a larger portion of the design is reused. Our metrics separate the measurements for couplings which are good for reuse, bad for reuse, and those which are neither but might be able to increase the value of the reusable classes. As illustrated in Figure 5, our goal is to have a design where:

- there exists many couplings between the related General classes (a), and
- the couplings among unrelated hierarchies are only between their Specific classes (b).

Using our previous assumptions of classes in Figure 2, the hierarchy with root class 'Person' is related to the one with root class 'Record.' Moreover, all classes of 'Record' are General, while only 'Person,' 'Patient,' 'Physician,' and 'Business' are General in the other hierarchy. If the Specific class 'Therapist' has a method 'Give_Therapy' which calls a method of 'Medical_R' named 'Get_Last_Medications' as shown in the code segment below:

```
class Therapist {
 ...
 void   *Give_Therapy(PatientID){
   char        *medication;
   Medical_R   *record;
   ...
   record = new Medical_R(PatientID);
   medication = (char *)
     record->Get_Last_Medication();
   if (strcmp(medication, "abc")==0){ ... }
   ...
 }
 ...
}
```

then, this coupling to 'Medical_R' is not a hindrance to reuse, since class 'Therapist' is not expected to be reused in future projects (a Specific class). However, our future system for smaller health care establishments will also benefit from this coupling. In a smaller health care system, the 'Business' class will also need to keep track of prescriptions. For example, if the 'Physician' orders a 'Patient' to buy more of a certain medication, the 'Business' will need to catch cases where the 'Patient' still has enough of that medication to avoid the opportunity to overdose. In HCA, 'Business' does not need to do this, since there is a 'Pharmacist' class which takes care of avoiding overdose. To accommodate reuse, we can move the coupling from 'Therapist' (for Get_Last_Medication) to 'Person,' such as in the following:

```
class Person {
  protected char  *medication;
  Medical_R       *record;
  ...
  void Get_Last_Medication(PatientID){
    record = new Medical_R(PatientID);
    medication =
      (char *) record->Get_Last_Medication();
  }
}

class Therapist : public Person {
  ...
  void  *Give_Therapy(PatientID){

    // using Person's method
    Get_Last_Medication(PatientID);

    // using Person's protected variable
    if (strcmp(medication, "abc")==0){ ... }

  }
  ...
}
```

The above allows the coupling to 'Medical_R' to be reused in our next project and leaves only those methods and operations specific to a 'Therapist,' such as 'Give_Therapy' and all other detailed operations, in the 'Therapist' class.

3.2 Understanding Coupling between Hierarchies

The second aspect of our design-reusability measurements involves an understanding of the couplings that exist between General/Specific classes when the hierarchies that they are in are related and unrelated. The four coupling types of Figure 4, when combined with related/unrelated hierarchies, yields eight types of reusability couplings. Each of these coupling types is discussed in turn with the intent to provide suggestions on eliminating non-desirable couplings and/or moving couplings to add value to the design whenever possible. The changes in the characterization of a class from General to Specific (or vice versa) and of hierarchies from related to unrelated (or vice versa) are not included in the suggested actions. These types of changes can greatly affect the overall design and the resulting reusable design components. Thus, these subjective characterizations should only be modified after a thorough review of the design, and not done just to eliminate an undesirable coupling.

1. **G—>G among related hierarchies.**
   A dependency from a General class to another General class in a related hierarchy is not a hindrance to reuse. Metrics to count this kind of coupling may denote the value of reuse. Increasing these couplings in a design yields a potential for more reuse.
   Action: None.

2. **G—>G among unrelated hierarchies.**
   A dependency from a General class to another General class in an unrelated hierarchy is undesirable because the source and destination are not expected to be reused together.
   Action: Attempt to move the dependency to their Specific descendant classes that are most relevant. Create new classes if necessary.

3. **G—>S among related hierarchies.**
   A dependency from a General class to a Specific class, even if they are among related hierarchies, is undesirable. This is because the General class, which is expected to be reused, depends on a class which is not expected to be reused. There are two possible movements: either move the source to a Specific descendant class or move the destination to an appropriate General ancestor. Since this coupling is between related hierarchies, the second option is better since it will become type #1.
   Action: Attempt to move the destination to an appropriate General ancestor class.

4. **G—>S among unrelated hierarchies.**
   A dependency from a General class to a Specific class is undesirable because the source is expected to be reused but the destination is not expected to be reused. There are two possible ways to eliminate this coupling: move the source to its Specific descendant class or move the destination to its General ancestor class. Since this is between unrelated hierarchies, the first option is better since if the second option is chosen, it would introduce another kind of undesirable coupling (type #2).
   Action: Attempt to move the source to an appropriate Specific descendant class.

5. **S—>G among related hierarchies.**
   A dependency from a Specific class is not a hindrance to reuse, since the source is not expected to be reused. However, we might be able to increase the value of reuse. The two options in moving this coupling are either to move the source to a General ancestor or to move the destination to a Specific descendant. The first option is better since the coupling will then be between two classes which are expected to be reused together (type #1). Hence, this type of move increases the value of the reusable design components.
   Action: Attempt to move the source to an appropriate General ancestor.

6. **S—>G among unrelated hierarchies.**
   This is not a hindrance to reuse because the source of the coupling is not expected to be reused. In this case, there is nothing we can do to increase the value of the reusable design components, because the dependency is between two hierarchies which are not expected to be reused together. Moving the source to its General ancestor would create another undesirable

coupling (type #2) and moving the destination to a Specific descendant does not add value to the reusable design components (type #8).
Action: None.

7. **S—>S among related hierarchies.**
This is also not a hindrance to reuse because the source of coupling is not expected to be reused. However, we might be able to increase the value of the reusable design components if both the source and destination are moved to their General ancestors. The dependency would then be between two classes which are expected to be reused together (type #1).
Action: Attempt to move both the source and destination to appropriate General ancestors.

8. **S—>S among unrelated hierarchies.**
This is not a hindrance to reuse, rather, it represents the desired situation for couplings between unrelated classes: they need to be among the Specific classes.
Action: None.

Overall, our goal is to direct the software designer to strive for maximum reuse by organizing all couplings into G—>G, if they are in related hierarchies, or S—>S, if they are in unrelated hierarchies.

Before we define the metrics in Section 3.3, it is important that we formalize the idea of related hierarchies. One hierarchy (H1) is defined as related to another hierarchy (H2) if they are expected to be reused together in one or more future systems. Let us use the operator '$\triangleright$' to define this binary relation. We can now express the relation between H1 and H2 as H1 $\triangleright$ H2. This relation is transitive but not commutative:

- if H1 $\triangleright$ H2 and H2 $\triangleright$ H3, then H1 $\triangleright$ H3
- H1 $\triangleright$ H2 does not imply H2 $\triangleright$ H1

This means that if we only have the following relations H1 $\triangleright$ H2 and H2 $\triangleright$ H3, we only expect to reuse one of the following sets of hierarchies: {H1,H2,H3}, {H2,H3} or {H3}. In this case, neither {H1} nor {H2} nor {H1,H2} can be reused in isolation.

3.3 Software Design Reusability Metrics

The metrics are defined in eight summations that correspond to the eight types of couplings given in Section 3.2. Coupling is defined as an inter-hierarchy dependency that results when methods of one hierarchy use methods or instance variables of another hierarchy. We use the term Coupling Counts, CC, to represent these interactions between hierarchies. These reusability measurements for a class hierarchy are then defined as:

$$\begin{array}{ll}
CC_1 = \sum_{i=1}^{m} \sum_{j=1}^{x} G_jG_i & CC_5 = \sum_{i=1}^{m} \sum_{j=1}^{y} S_jG_i \\
CC_2 = \sum_{i=1}^{n} \sum_{j=1}^{x} G_jG_i & CC_6 = \sum_{i=1}^{n} \sum_{j=1}^{y} S_jG_i \\
CC_3 = \sum_{i=1}^{m} \sum_{j=1}^{x} G_jS_i & CC_7 = \sum_{i=1}^{m} \sum_{j=1}^{y} S_jS_i \\
CC_4 = \sum_{i=1}^{n} \sum_{j=1}^{x} G_jS_i & CC_8 = \sum_{i=1}^{n} \sum_{j=1}^{y} S_jS_i
\end{array}$$

where

m: # of hierarchies which are related to this one
n: # of hierarchies which are not related to this one
x: # of General classes in this hierarchy
y: # of Specific classes in this hierarchy

G_jG_i: # of couplings from the j-th General class to all General classes in the i-th hierarchy
G_jS_i: # of couplings from the j-th General class to all Specific classes in the i-th hierarchy
S_jG_i: # of couplings from the j-th Specific class to all General classes in the i-th hierarchy
S_jS_i: # of couplings from the j-th Specific class to all Specific classes in the i-th hierarchy

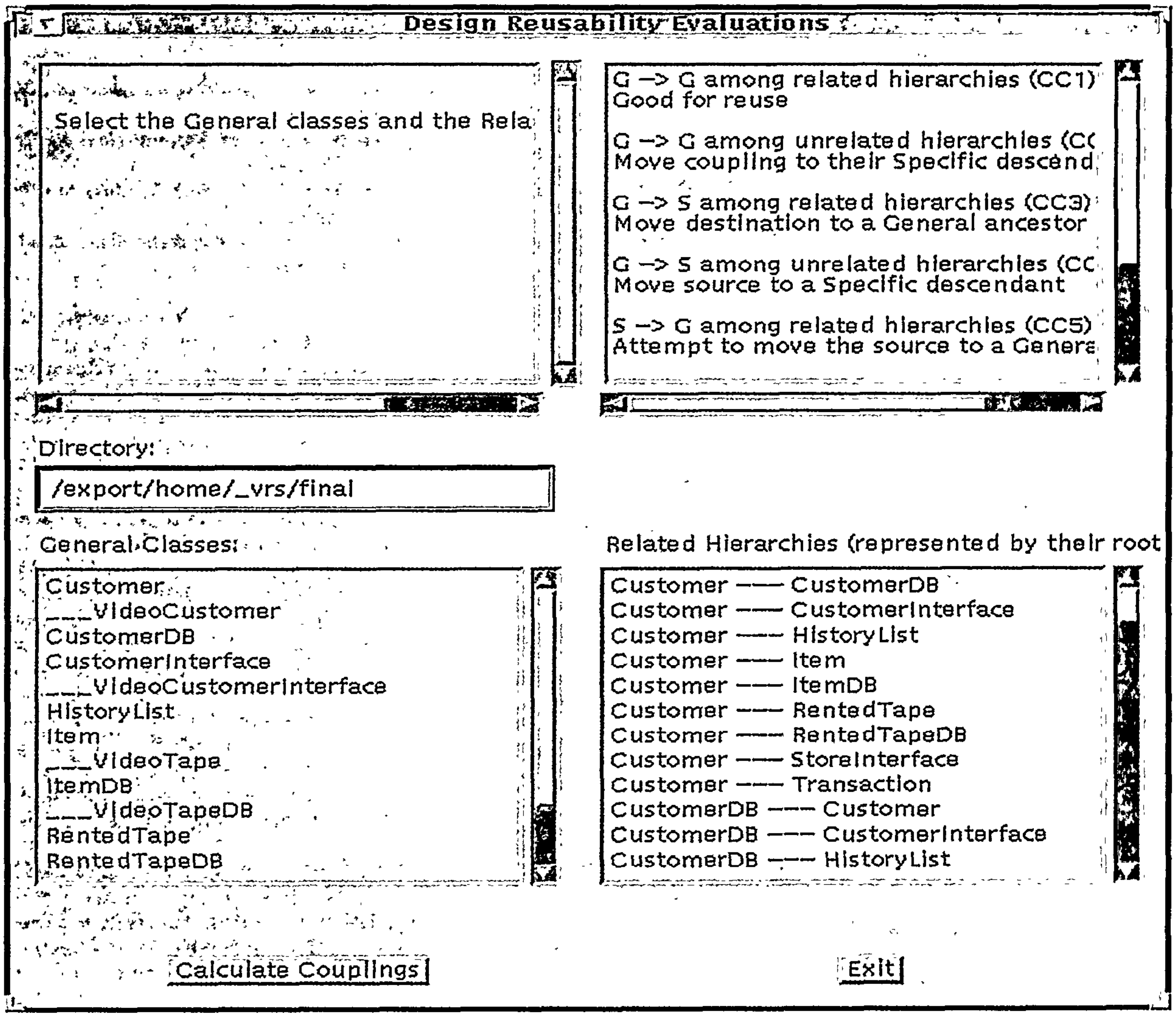
To understand these counts, we provide a plausible scenario of the way that they can be utilized in practice.

Suppose that a software designer has characterized all of his/her classes as either General or Specific classes (see Section 2.2 again), and the related classes have also been defined (see Section 2.3 again). CC_1 through CC_8 can initially be calculated. High CC_1 values indicate that there are many couplings between classes which are expected to be reused together. This is very good in terms of reuse since we will be reusing many design components. Low CC_1 values denote that there are not many couplings to reuse, which may indicate that the software designer needs to review couplings for possible changes. If any of CC_2, CC_3, or CC_4 have values greater than 0, the software designer will need to either remove or move these dependencies. This is because these coupling sources are expected to be reused, but the coupling destinations are either not expected to be reused or belong to unrelated hierarchies. To accomplish this, the actions defined in Section 3.2 need to be consulted. The couplings in CC_3 can actually be used to create more couplings of type # 1 in Section 3.2, which increases the value of CC_1, the desirable kind of coupling.

Any value in CC_5 and CC_7 does not indicate a hindrance to reuse, since they are counting dependencies where the source is not expected to be reused. However, the designer might want to look into these couplings more closely, since they can be utilized to create more desirable couplings and thereby increase CC_1. Any value in CC_6 and CC_8 are also not indicating a hindrance to reuse because the coupling source is not expected to be reused. These values denote the dependencies which are specific to this application. There is no action needed for any value of CC_6 and CC_8; rather they are provided for the next time the software designer does an overall design review. At this time, the software designer can only move or remove dependencies, and when the couplings are changed, CC_1 to CC_8 can be automatically recalculated to provide a current view of the reuse potential of the OO design. However, during an overall design review, the characterizations of classes (General/Specific) and hierarchies (related/unrelated) can be reviewed and modified to increase the reusability of the design.

4 EMPIRICAL STUDY

We have conducted an experiment to study the effectiveness of our reusability measurement framework. This was done as part of a joint graduate/undergraduate project. The graduate student, M. Price, has been conducting work to verify the framework presented in Sections 2 and 3 as part of her dissertation efforts, by developing a tool that can be utilized to analyze C++ code when given information on the General and Specific classes, and the Related hi-



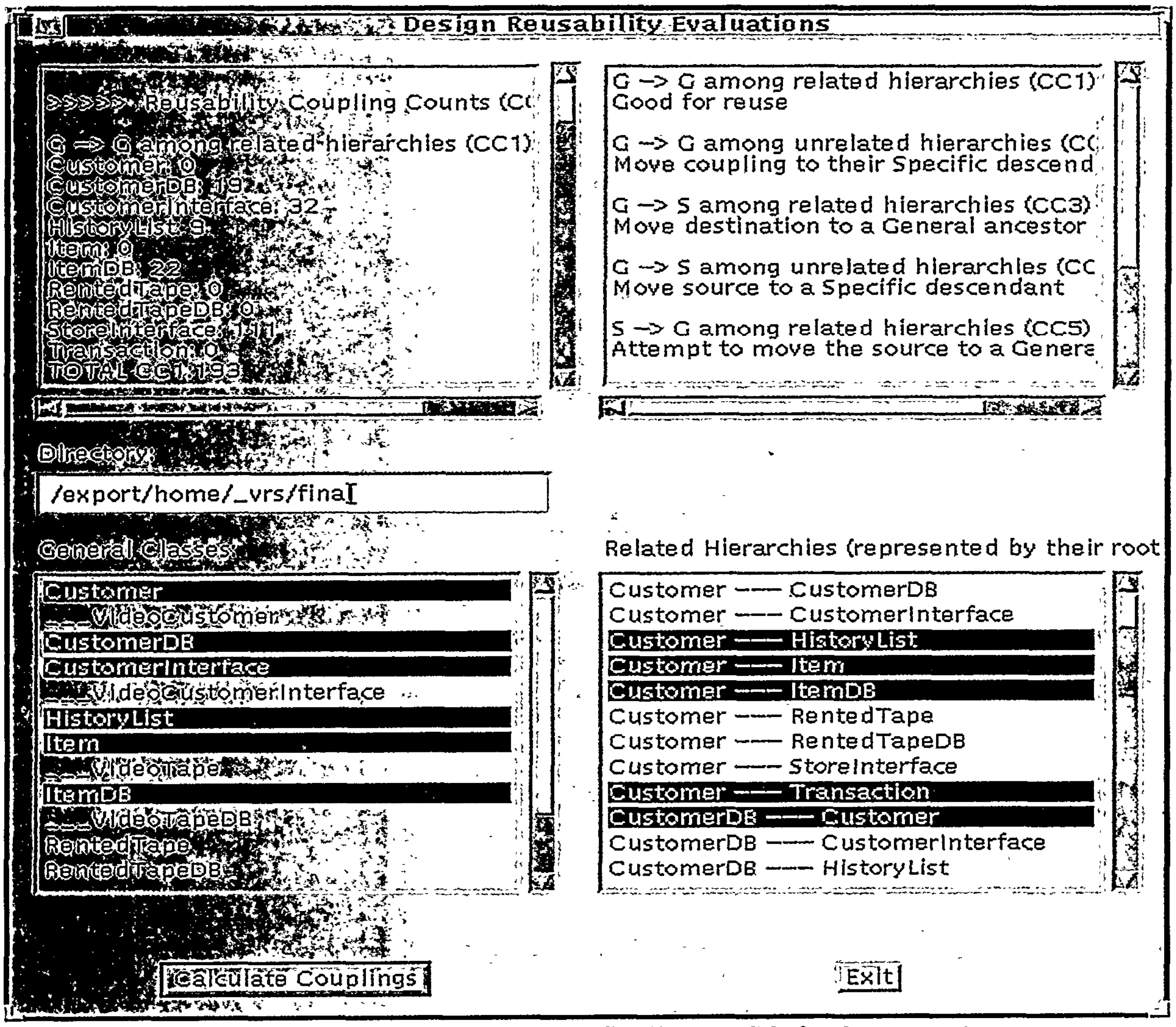
```
Select the General classes and the Rela

G -> G among related hierarchies (CC1)
Good for reuse

G -> G among unrelated hierarchies (CC
Move coupling to their Specific descend

G -> S among related hierarchies (CC3)
Move destination to a General ancestor

G -> S among unrelated hierarchies (CC
Move source to a Specific descendant

S -> G among related hierarchies (CC5)
Attempt to move the source to a Genera

Directory:
/export/home/_vrs/final

General Classes:
Customer
___VideoCustomer
CustomerDB
CustomerInterface
___VideoCustomerInterface
HistoryList
Item
___VideoTape
ItemDB
___VideoTapeDB
RentedTape
RentedTapeDB

Related Hierarchies (represented by their root
Customer ---- CustomerDB
Customer ---- CustomerInterface
Customer ---- HistoryList
Customer ---- Item
Customer ---- ItemDB
Customer ---- RentedTape
Customer ---- RentedTapeDB
Customer ---- StoreInterface
Customer ---- Transaction
CustomerDB ---- Customer
CustomerDB ---- CustomerInterface
CustomerDB ---- HistoryList
```

Figure 6. Design Reusability Evaluations (DRE).

erarchies. The undergraduate student, Kevin Jin, a senior, was responsible for designing and developing two applications that have significant overlap in spite of their domain differences. Each of these efforts contributes to the overall goal of providing empirical results and an automated framework to support the analyses of C++ code, which has been derived from object-oriented designs according to our reuse approach.

Specifically, we have developed the first prototype of a tool to calculate C++ couplings using our measurement framework, which we call Design Reusability Evaluations (DRE). DRE takes a directory containing C++ classes as input that is parsed to return the hierarchical list of classes and the list of all combinations of root classes as shown in Figure 6. Users can then select the classes which are to be characterized as General classes (all classes not chosen are assumed to be specific) and those hierarchies which are supposed to be Related. After the choices have been made and the "Calculate Coupling" button is clicked, the resulting metrics (CC1 through CC8) are displayed in the upper left window as shown in Figure 7. The upper right window identifies the actions for the various dependencies as specified in Section 3.2. The software engineer can then utilize the measurements to identify those portions of the code that need to be changed, or to rethink which classes should be General/Specific and which hierarchies should be Related. DRE can be used to analyze the reusability of completed C++ code, and is intended to be incorporated into either design or development environments. Its key purpose is to promote an iterative process that evolves design/code to a more reusable state.

Given this tool, it was then necessary to provide input for at least two applications, that while different, have the potential to share significant portions of both design and code. This was the responsibility of the undergraduate student, who was asked to design and develop a video rental system (VRS) and an auto service center system (ASCS). VRS maintains customer and video databases, keeps track of each transaction (borrow/return), and logs the tapes that have been rented. There are two interfaces to VRS: a Customer Interface, which lets customers browse and search the video tapes; and a Store Interface, which lets store atten-


Design Reusability Evaluations

```
>>>>> Reusability Coupling Counts (C

G -> G among related hierarchies (CC1)
Customer: 0
CustomerDB: 19
CustomerInterface: 32
HistoryList: 9
Item: 0
ItemDB: 22
RentedTape: 0
RentedTapeDB: 0
StoreInterface: 111
Transaction: 0
TOTAL CC1: 193
```

```
G -> G among related hierarchies (CC1)
Good for reuse

G -> G among unrelated hierarchies (CC
Move coupling to their Specific descend

G -> S among related hierarchies (CC3)
Move destination to a General ancestor

G -> S among unrelated hierarchies (CC
Move source to a Specific descendant

S -> G among related hierarchies (CC5)
Attempt to move the source to a Genere
```

Directory:

/export/home/_vrs/final

General Classes:

```
Customer
___VideoCustomer
CustomerDB
CustomerInterface
___VideoCustomerInterface
HistoryList
Item
___VideoTape
ItemDB
___VideoTapeDB
RentedTape
RentedTapeDB
```

Related Hierarchies (represented by their root

```
Customer ---- CustomerDB
Customer ---- CustomerInterface
Customer ---- HistoryList
Customer ---- Item
Customer ---- ItemDB
Customer ---- RentedTape
Customer ---- RentedTapeDB
Customer ---- StoreInterface
Customer ---- Transaction
CustomerDB ---- Customer
CustomerDB ---- CustomerInterface
CustomerDB ---- HistoryList
```

Figure 7. DRE: After the Couplings are Calculated.

dants manipulate tapes, manipulate customers' information, and process the borrowing/returning of tapes.

The undergrad was advised that we also had to build another application at a later time in the semester, ASCS, which is to be used to keep track of an auto parts inventory, customer information, and the services that are performed on the cars. While VRS and ASCS have many similarities, they also have some interesting differences. They both need a database of items, which can be either tapes or auto parts. Both types of items have names, categories, descriptions, number of available items, and suppliers. Video tapes have specific information such as running time and rating. Auto parts have specific information, such as the minimum number of items (before they need to reorder) and the price of each item. Similarities and differences also exist in the customer database. The auto service customer database needs to keep information on the car owned by each customer, such as the year and model. Both systems need to keep track of customers' account balances and the history of transactions. But the transaction history is very different. In ASCS, a customer who needs a certain part installed on their car, buys the item without needing to return it. Thus, when a customer buys an item, we need to increase his/her balance and log the transaction. In VRS, the customers borrow the tapes, so that in addition to increasing the balance and logging the transaction, we also need to keep track of the tapes being borrowed.

These applications were chosen because they are easily understood. The undergrad was able to define and interpret the requirements. This allowed us to concentrate on the concepts, since the contexts are well understood. Moreover, there are many similarities, so we can expect many reusable dependencies. During the design of VRS, the object-oriented CRC based approach was utilized. The characterizations of the General/Specific classes and the Related hierarchies were incorporated into the CRC approach. The design process of the undergrad was closely monitored and discussed in several walkthroughs.

During the design stage, the student first identified the major classes and determined their General/Specific char-

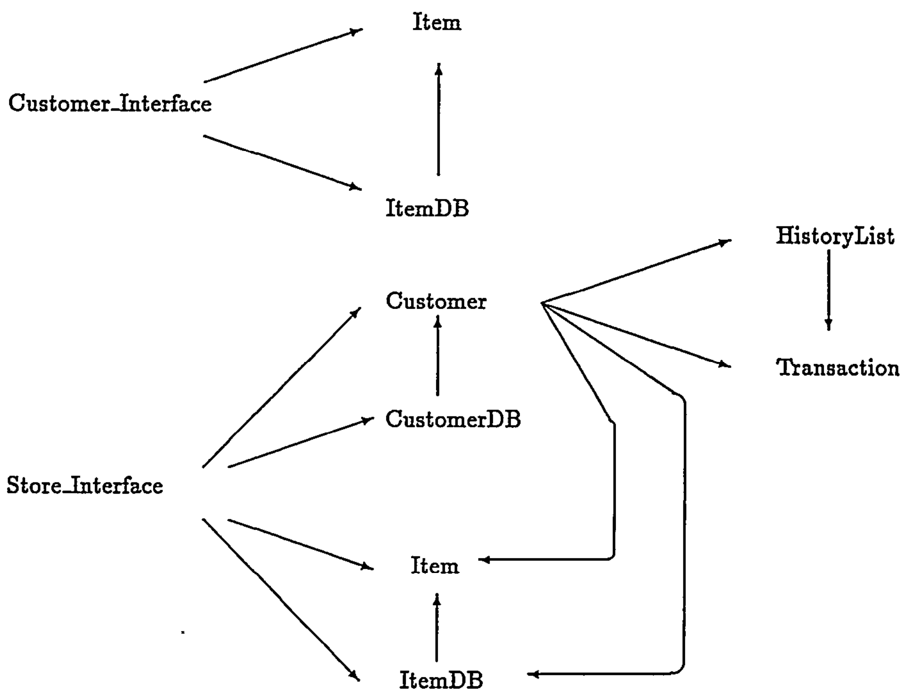


Figure 8. General Root Classes and Related Hierarchy Relationships of the Video Rental System.

Type	Metric	I	II	III	IV	V	VI	VII
Good for Reuse	CC1	8	54	79	109	166	167	193
Bad for Reuse	CC2	0	0	0	0	0	0	0
	CC3	6	0	0	0	4	4	0
	CC4	0	0	1	1	0	0	0
Can Improve	CC5	0	19	28	39	94	92	75
Reuse (if moved)	CC7	0	6	6	18	28	30	25
No Impact	CC6	0	0	0	0	0	5	0
on Reuse	CC8	0	0	11	11	21	22	26
	# of lines	410	1386	2088	2695	3784	3824	3867

Table 1. Reusability Measurements of VRS.

acterizations. He first identified all of the classes as General classes, except for those used to keep track of the rented tapes. These General classes are shown in Figure 8. This is reasonable, since these classes are needed by the auto service center. However, when he started to define the attributes, he realized that he could not include some of the attributes in the General classes, such as the running time and rating of video tapes. This resulted in the creation of subclasses of Item, ItemDB, Customer, CustomerInterface, and StoreInterface to support VRS initially, and ASCS subsequently.

After all of the attributes and classes were identified, the undergrad defined not only the methods, but also, for each method, the other methods that are required to realize the needed functionality. By defining these dependencies, the Related hierarchies could be identified. Even though this process is subjective, the student was able to define the classes which need to be reused together. We asked him to define hierarchy A to be related to hierarchy B if he expected the methods in A to use many methods in B and if both A and B will be needed in ASCS. The General root classes and the Related hierarchy relationships (the arrows) are also shown in Figure 8.

Couplings between the methods in the design were examined carefully and changed according to actions specified in Section 3.2. Even though we have clearly defined the General classes and the Related hierarchies and their purpose, many undesirable couplings occurred during both the design and the coding stages. In this paper, it is more interesting to show the results of the coding phase, because the design is relatively small. More complicated systems have many more dependencies which can be identified during the design phase and they can expect the same benefit in automating the measurements.

During our seven code walkthroughs, we utilized the DRE tool and discussed its results. We have included the results in Table 1 to illustrate the improvements gained by

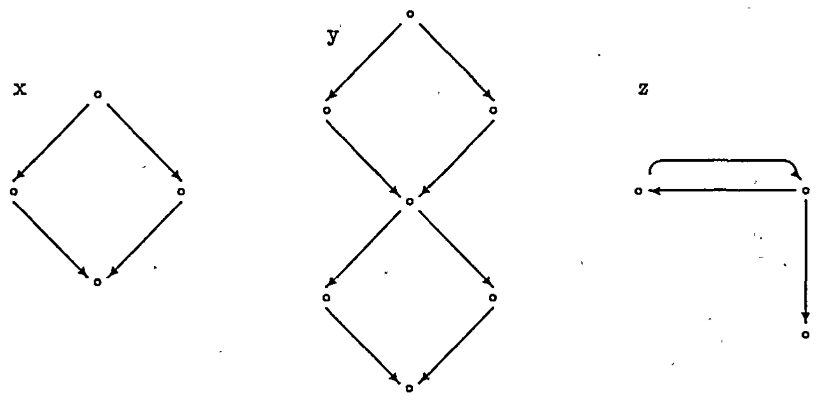


Figure 9. Incomparable Program Complexity [11]

using an automated framework. The reusable portion of the VRS system was then used in ASCS. The VRS code contains a total of 3867 lines, among which 2485 of them are reusable as is for ASCS. The number of lines are relatively small, because we use web browsers as our interface. VRS is written as Common Graphics Interface (CGI) code which responds to requests from any web browser. The number of lines reused are only provided for completeness; we believe that the value of reuse relies more on the number of good dependencies that are reused, as reflected in the CC1 count.

Because of the subjectivity aspect of this approach, it is necessary for the software engineers/designers to have a good understanding about:

- the application domain, and
- the types of systems that they expect to build in the future.

If software engineers continue to design in a vacuum, they will continue to produce code with minimal or no reuse. Our General/Specific classes and Related hierarchies are intended to provide a framework for software engineers to think about reuse at early and all stages of the design and development process. The resulting metrics of this framework can be used to provide automatic guidance during those time periods in between walkthroughs. Whenever a design walkthrough is conducted, the subjective characterizations of classes and hierarchies can then be reviewed and modified if necessary. These characterizations are very critical in the success of this approach, thus they should only be changed after a careful analysis of the overall system.

5 CONSIDERING METRICS THEORY

There is not yet an agreement on the set of properties which make some software more reusable than others. Poulin states that in general, most sets of reusability guidelines reflect the same properties as those promoted by good software engineering principles [20]. These good software engineering principles include low coupling. However, as we have seen in Sections 3.2 and 3.3, not all couplings are bad for reuse. Software designs are composed of the system's components and the interactions between those components. These interactions (or couplings) are valuable to the design, and many of them tie those reusable classes together to create a meaningful design.

In the context of software complexity, Fenton showed that the search for a general-purpose real-valued complexity measure is doomed to failure [11]. In dozens of proposed complexity measures, there is a minimum assumption that the empirical relation system for complexity of programs leads to at least an ordinal scale. An ordinal scale involves a ranking, from best to worst. But there is a problem with this, because some programs are "incomparable." In his paper, Fenton provides the example given in Figure 9 (which is Figure 1 in [11]). Most would agree that x is less complex than y. However, when people are asked which is more complex between x and z or y and z, they end up asking questions like "what is meant by complexity" before attempting to answer. From the measurement theory perspective, it is good enough if most programmers agree on the complexity order of x, y and z; but there is no such agreement in the order of their complexity.

Software reusability metrics have the same problems. We cannot try to put an empirical value on a poorly understood attribute [20]. Software engineers working on different domains will have different opinions on the reusability of a component. Some components are more reusable in one domain and less reusable in others, so they are incomparable with respect to reusability. This means that we should stop searching for a general reusability metric, and instead look for the specific properties or aspects of reuse. This is one goal of our work as presented in this paper.

In his recent book, Fenton states that measurements of internal software attributes can be useful when restricted to locally specified, commonly accepted definitions of the underlying terms [12]. We believe that the metrics defined in this paper are restricted to specific properties of object-oriented software. They are not only specific to couplings, but they are also separated by the kinds of couplings with respect to the reusability of a set of related hierarchies. It is not correct to combine the measurements of those couplings which are good for reuse and those which are bad for reuse. We also believe that they are based on a commonly accepted understanding of reusability: a set of related components in a system is more reusable if it has less dependencies to other parts of the system.

6 RELATED WORK

There are several proposals on the ways to make OO software systems more reusable. One of them is by Batory and O'Malley [1] which is a domain-independent model of hierarchical systems, based on domain modeling and building-block technologies. They show that complex domains can be expressed by an elementary metamodel of interchangeable and plug-compatible components. Their model is aimed specifically at mature software technologies, where standardization makes sense. Standardization is possible for certain domains, such as in communication protocols, where there is a set of operations which have to occur in a certain order. Strayer has built a set of classes called the Meta-Transport Library which provides a set of protocol-inspecific base classes for transport layer protocols [22]. Specific transport layer protocols can be built through derivations from these classes.

On the other hand, our techniques can assist the design and development process of reusable systems which do not yet have a standard set of operations. Extensive domain analysis tends to be expensive and not always possible. Moreover, our measurement techniques are aimed at organizations which have some ideas on what kinds of systems they would like to build in the future. Software companies do not have to make their software reusable for any system in the domain, but they only have to make their software designs be reusable in anticipated future systems in their organizations.

Once there is a reusable design, we create other specific systems by extending the base classes. Our form for extending these classes is by inheritance. Another way to extend the behavior of a class hierarchy is by combining the base hierarchy with one or more extension hierarchies [19]. The extension hierarchies can extend or overwrite the behavior of the existing classes, so that the existing classes do not have to be changed. However, overwriting the behavior of existing classes has the same impact as changing those existing classes, since it is possible that the behaviors of the combined hierarchies are incompatible. The combined hierarchies will need to be thoroughly re-tested. They define conditions that the hierarchies have to be non-conflicting, non-interfering, and semantically compatible. The definition of semantic compatibility is still an area of research. We are in full agreement with the authors that in large systems, extending classes by inheritance becomes unmanageable. However, if the original classes are designed to support reusability, extension by subclassing has the potential to be very manageable.

In the area of reusability measurements, there are various empirical and qualitative methods which have been proposed [20]. Prieto-Diaz and Freeman identify five reusability metrics: size (favors small module size), structure (favors simple structure, low coupling, and low complexity), good documentation (subjective rating), programming language (favors same language), and reuse experience (in the domain and the programming language) [21]. We agree with the last three. The first two are arguable. Smaller size and less complex code are easier to understand, which makes them easier to reuse. But they do not produce high savings. The best way to approach this comprehensibility problem is by making the large components more understandable by having well-defined interfaces and good documentation.

Another technique, which is also reviewed in [20], is developed by the Reuse Based on Object-Oriented Techniques (REBOOT) project. They define four reusability factors (portability, flexibility, understandability and confidence) and many criteria and metrics under each factor. Some of the metrics are empirical and some are qualitative which are measured using checklists. They define reusability by normalizing all metrics to a value between 0 and 1 and taking the average. They also multiply each metric value with its weight to describe the relative importance of each metric. This method may not work as well for reusability since as discussed earlier, certain systems are incomparable with respect to their reusability. However, their comprehensive list of criteria provides a good set of those properties which are also promoted by good software engineering principles. Each of their criteria can be used to provide an idea of software reusability with respect to that specific property, but they probably should not be combined into a general reusability metric.

7 CONCLUSIONS & AN ONGOING EFFORT

We have presented a framework for reusability measurements which facilitates large-scale OO design reuse. The following summarizes the contributions of this measurement framework:

1. It provides an effective and subjective method to distinguish between what is expected to be reused and what is not. This is accomplished by the differentiation between General and Specific classes as presented in Section 2.2.

2. It provides an effective and subjective method to group related components into reusable portions. This is accomplished by the differentiation between related and unrelated class hierarchies as presented in Section 2.3.

3. It presents specific metrics based on the above concepts, which can work at any level of design detail. This is achieved by the mathematical formulas for CC_i in Section 3.3, which are based on the types of couplings and refinement actions presented in Sections 3.1 and 3.2. This is demonstrated in Section 4 in our empirical study via the design reusability evaluations tool.

The first two items listed above are the important components of a software design, and they are to be determined subjectively by the software designer. The last item is the metrics that objectively measure the couplings of design components; couplings are the tangible representation of a design product, thus they can be measured objectively. The last item includes a set of suggestive actions to automatically advise software designers/developers on the ways to improve their products' reusability, and follows through with an actual tool that evaluates reuse potential in C++ code according to our framework.

Our approach relies heavily on subjective decisions and it requires the software designers to have a good understanding about the application domain and the types of systems that they expect to build in the future. When changes are made to the subjective characteristics of an OO design, the objective measures are recalculated to provide the software designer with an evolving and incremental perspective on those portions of the OO design that have the most potential to be reused. This was illustrated in Section 4 with our experimental work on the reuse between video rental and auto service center systems (see Figures 6 and 7 and Table 1 again). Our coupling measurements also conform with the results and recommendations of research in metrics theory, which is described in Section 5.

We are working on integrating this reusability evaluation framework in the Active Design and Analysis Modeling (ADAM) environment [6, 8, 14, 18]. ADAM supports a language-independent design process, where software designs can be entered and code can be generated in various languages (Ada 83, Ada 95, C++, Ontos C++ and Eiffel). We are inserting this reusability evaluation technique into ADAM to support the subjective aspects of our framework that can then be utilized to automatically warn the software designer whenever a non-reusable coupling is introduced and provide suggestions on the way to eliminate the coupling and/or move it to add value to the reusable design.

References

[1] D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components," *ACM Transactions on Software Engineering and Methodology*, Vol. 1, No. 4, October 1992, pp.355-398.

[2] T. Biggerstaff and A. Perlis, editors, *Software Reusability*, ACM Press, New York, NY, 1989, volume I, pp. xv-xxv.

[3] L. Briand, S. Morasca, and V. Basili, "Defining and Validating High-Level Design Metrics," University of Maryland, Technical Report number CS-TR-3301.

[4] D. Card and R. Glass, *Measuring Software Design Quality*, Prentice-Hall, Englewood Cliffs, NJ, 1990.

[5] S. Chidamber and C. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, June 1994, pp. 476-493.

[6] S. Demurjian, T. Daggett, T.C. Ting, and M.-Y. Hu, "URBS Enforcement Mechanisms for Object-Oriented Systems and Applications," in *Database Security, IX: Status and Prospects*, D. Spooner, S. Demurjian, and J. Dobson (eds.), Chapman Hall, 1995.

[7] S. Demurjian and T.C. Ting, "The Factors that Influence Apropos Security Approaches for the Object-Oriented Paradigm," *Workshops in Computing*, Springer-Verlag, 1994.

[8] H. Ellis and S. Demurjian, "Object-Oriented Design and Analyses for Advanced Application Development - Progress Towards a New Frontier," *Proceedings of the 21st Annual ACM Computer Science Conference*, February 1993.

[9] W. Griswold, "Program Restructuring as an Aid to Software Maintenance," Department of Computer Science and Engineering, University of Washington, Technical Report number 91-08-04, August 1991.

[10] W. Griswold and D. Notkin, "Automated Assistance for Program Restructuring," Department of Computer Science and Engineering, University of Washington, Technical Report number 90-08-05, August 1990.

[11] N. Fenton, "Software Measurement: A Necessary Scientific Basis," *IEEE Transaction on Software Engineering*, Vol. 20, No.3, March 1994, pp. 199-206.

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

[13] M.-Y. Hu, S. Demurjian, and T.C. Ting, "User-Role Based Security Profiles for an Object-Oriented Design Model," in *Database Security, VI: Status and Prospects*, C. Landwehr and B. Thuraisingham (eds.), North-Holland, 1993.

[14] M.-Y. Hu, S. Demurjian, and T.C. Ting, "Unifying Structural and Security Modeling and Analyses in the ADAM Object-Oriented Design Environment," in *Database Security, VIII: Status and Prospects*, J. Biskup, C. Landwehr, and M. Morgenstern (eds.), Elsevier Science, 1994.

[15] R. Johnson and W. Opdyke, "Refactoring and Aggregation," *Object Technologies for Advanced Software*, November 1993, volume 742, pp. 264-278.

[16] R. Johnson and B. Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming*, June/July 1988.

[17] R. Johnson and V. Russo, "Reusing Object-Oriented Designs," University of Illinois, Technical Report number UIUCDCS 91-1696, May 1991.

[18] D. Needham, S. Demurjian, K. El Guemhioui, T. Peters, P. Zemani, M. McMahon, and H. Ellis, "ADAM: A Language-Independent, Object-Oriented, Design Environment for Modeling Inheritance and Relationship Variants in Ada 95, C++, and Eiffel," *Proceedings of 1996 TriAda Conference*, Philadelphia, PA, December 1996.

[19] H. Ossher and W. Harrison, "Combination of Inheritance Hierarchies," *OOPSLA 1992 Conference Proceedings*, pp. 25-40.

[20] J. Poulin, *Measuring Software Reuse - Principles, Practices and Economic Models*, Addison-Wesley, 1997.

[21] R. Prieto-Diaz and P. Freeman, "Classifying Software for Reusability," *IEEE Software*, Vol. 4, No. 1, January 1987, pp. 6-16.

[22] T. Strayer, "A Class-Chest for Deriving Transport Protocols," *Proceedings of the 21st Local Computer Networks Conference*, Minneapolis, MN, October 13-16, 1996.