Deriving Measures of Software Reuse in Object Oriented Systems[*]

James M. Bieman
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523 USA
(303) 491-7096
bieman@cs.colostate.edu

Presented at the British Computer Society
Workshop on Formal Aspects of Measurement, London, 1991
Published in *Formal Aspects of Measurement*, T. Denvir and R. Herman and R. Whitty (editors)
Springer-Verlag 1992, pp 79–82

Technical Report CS–91–112

July 1991

Abstract

The analysis and measurement of current levels of software reuse are necessary to monitor improvements. This paper provides a framework for the derivation of measures of software reuse and introduces several definitions, attributes, and abstractions of potentially measurable reuse properties. The framework is applied to the problem of measuring reuse in object oriented systems which support "leveraged" reuse through inheritance. I describe the importance of the perspective of the observer when analyzing, measuring, and profiling reuse. Three perspectives are examined: the server perspective, the client perspective, and the system perspective. Candidate reuse metrics are proposed from each perspective.

1 Introduction

Research on formal aspects of software measurement tends to focus on properties of measures, measurement theory, measurement scales, and requirements for predictive measures. This paper reports on the practical application of techniques developed in formal studies of software measurement. The overall aim is to derive and measure intrinsic properties of software documents and processes.

Measurement theory is the foundation for practical software measurement. A measure allows us to numerically characterize intuitive attributes of software objects and events. We need a clear definition and understanding of a software attribute before we can define measures. We must be able to determine that one software entity has more or less of some attribute before we can use measurement to assign a numerical quantity of the attribute to the entity. And the orderings of the software entities implied by the measurement must be consistent [BBF$^+$90, MGBB90, FM90].

Software reuse is one important measurable property. Software quality, reliability, and productivity should improve when we reuse our programs, designs, specifications, etc. Software reuse reduces the quantity of software that must be developed from scratch and thus allows a greater focus on quality. The reuse of well-tested software should result in greater reliability. And the high cost of verification of a reused module can be spread out over many products. With reuse, software development becomes a capital investment. The development effort for each product is used to build a software infrastructure which can be utilized in subsequent products [Weg84].

The rigorous measurement of reuse will help developers monitor current levels of reuse and help provide insight into the problem of developing software that is easily reused. Conte [CDS86], Boehm [Boe81], Bailey [BB81], and Fenton [Fen91] describe reuse measures that are based on comparisons between the length or size of reused code and the size of newly written code in particular software products. Modifications to the reused code are not considered by these researchers. The purpose of Conte's reuse measure is estimating coding effort. Reuse reduces coding effort, and this reduction affects effort estimation techniques. In a similar light, Boehm and Bailey use the size of reused code to adjust cost predictors. Fenton also develops a measure of reuse based on the dependencies in an associated call graph. Selby [Sel89] classifies modules into categories based on the percentage of new versus reused code in a module. The categories are (1) completely new modules, (2) reused modules with major revisions ($\geq 25\%$ changed), (3) reused modules with slight revisions ($< 25\%$ changed), and (4) modules that are reused without change. The foregoing reuse measure-

*This research is partially supported by the NATO Collaborative Research Grants Program under RG. 0343/88

ment techniques look at reuse in a one-dimensional fashion. Except for Fenton's measure of private reuse described in Section 3.1, all of the foregoing reuse measurements are based on one attribute, program size (or program length). Only Selby addresses the measurement of reuse with modifications.

Proponents assert that a major benefit of object oriented design and programming is the generation of reusable software components [Mey87]. Components can be reused as is, or modified using subclassing facilities. Class libraries promote the reuse of well tested components of general or special interest. Object oriented systems promote reuse through language features such as data abstraction and inheritance constructs, and through system support such as browsers.

To support or refute claims that object oriented software is easier to reuse, we must be able to measure reuse in these systems. Measuring reuse in object oriented systems requires that we define attributes, abstractions, and measures appropriately for these support mechanisms. Only with meaningful measures can we determine whether object oriented systems really promote reuse.

Ultimately we want to determine whether there are structural properties that make software more likely to be reused. If there are such properties, we want to identify them. We believe that there are – software engineering practices have long supported the notion that certain internal structure such as structured programming, data abstraction, information hiding, etc result in higher quality software products.

The existence of structural properties which improve the level of reuse can only be determined empirically. To investigate this issue, we need to be able to characterize reuse in actual systems. Determining whether object oriented techniques promote greater reuse requires that we be able to measure the quantity of reuse.

Measurement of the current state of industrial practice is of great concern to the software engineering community [GC87]. Only with accurate measures of the "level" of actual reuse can projects be monitored and "improvement" measured.

I want to be able to measure reuse related to object oriented class libraries. Reuse occurs within a class library; library classes may used by other library classes. Library classes may also be reused by a new system. Data from industry can be examined to learn how reuse naturally occurs in object oriented systems. Experiments can help us relate the "level" of reuse to other software properties. But first, the important reuse attributes must be identified.

My goal is to derive a set of measurable, intuitively meaningful reuse attributes, and then use the derived measures in empirical research. This paper describes the process of deriving measurable reuse attributes and is a prerequisite to future empirical investigations. The focus here is on code reuse. In the future, I plan to investigate the derivation of measures of specification and design reuse.

This paper is organized as follows. Section 2 describes the process, borrowed from measurement theory, for deriving new measures of software reuse. Section 3 introduces a number of definitions related to reuse, reuse attributes, and describes prospective abstractions which capture the reuse attributes. In Section 4, the definitions, attributes, and abstractions described in Section 3 are applied to reuse in object oriented systems. An additional abstraction is proposed and a set of candidate reuse measures is introduced. A summary of the results presented in the paper and conclusions are given in Section 5.

2 A Method for Deriving Reuse Measures

Measurement theory provides guidance for deriving reuse measures. Aspects of the "quantity of reuse" are internal product attributes related to properties of particular software documents [Fen91]. Measurement theory suggest the following process [BBF$^+$90]:

1. Identify and define intuitive and well-understood attributes of software reuse. We must qualitatively understand what we want to measure.

2. Specify precisely the documents and the attributes to be measured. We must be able to describe precisely the object that we measure, and the property of the object that the measurement is to indicate.

3. Determine formal models or abstractions which capture these attributes. Formal models are required to unambiguously produce numerical measurement values.

4. Derive mappings from the models of attributes to a number system. The mappings must be consistent with the orderings implied by the model attributes.

The first three steps are critical. Good definitions of reuse attributes, the documents, and abstractions are needed before the attributes can be meaningfully measured.

A long term goal is to be able to use reuse measures to make predictions on external software process attributes such as the development time, maintainability, defects, etc. Industry's need for predictors is often the overriding issue when new measures are introduced and validated. For a measure to be effective in making predictions we need theories relating different measures, and empirical support. Otherwise the use of a measure for prediction is only based on speculation. The requirement that an internal measure effectively predict external process attributes has had a negative impact on software metrics research, especially affecting research directed towards developing measures of structural "complexity" [Fen91]. However, a measure can be useful even if it is not an effective predictor. To be useful, a measure must give a numerical value to a software attribute that is of genuine interest. I defer worrying about the added requirements for using measures for predictions, and simply seek valid and useful reuse measures.

To derive reuse measures we need clear definitions of software reuse attributes, precise definitions of the documents to be measured, formal models of the reuse attributes, and a method for generating consistent numerical values from the documents and attributes.

3 Reuse Definitions, Attributes, and Abstractions

Our derivation of reuse measures concentrates on clear definitions of reuse properties and abstractions which capture these properties.

3.1 Public and Private Reuse

Fenton defines *public reuse* as "the proportion of a product which was constructed externally" [Fen91]. He determines this proportion in terms of program lengths. The following is a modified version of a definition of public reuse from [Fen91]:

$$\text{public reuse} = \frac{\text{length}(E)}{\text{length}(P)}$$

where E is the code developed externally and P is the new system including E. This definition of public reuse makes use of any acceptable abstraction of program length, which may be lines of code or the number of characters. To use such a measure, one must be able to clearly distinguish between the components (lines or characters) that are from the external source and the components that are completely new. Abstractions based on more generic components are also appropriate.

Assume the existence of a library of previously developed software components. Useful components are imported from library when a new system is developed. Components may be modified when imported. The import mechanism may be designed into the environment, or ad hoc. Reuse is determined by the relation between the new system and the library.

In the ensuing discussion in this section, let L = "reuse library" and N = "new system", assuming L and N are sets of software "components." A component may be a source line of code, a statement, a procedure, a module, or an entire sub-system. I can define *reuse* as the use of $e \in L$ in N. (use is not formally defined).

The software entity from the library that is being reused is called the *server* and the software entity that makes use of the the library component is the *client*.

Fenton defines *private reuse* (or perhaps more appropriately *internal reuse*) as the "extent to which modules within a product are reused within the same product" [Fen91]. He uses the call graph as an abstraction which captures the "essence" of private reuse. A call graph is a (directed) graph which represents the control flow connections between modules. Call graph nodes represent modules, and nodes are connected with an edge if a module represented by one of the nodes invokes the the module represented by the other node. Figure 1 is an example call graph. Here M0 directly uses M1, M2, and M3. M1 uses M4, M2 uses M3 and M4, and M3 uses M4 and M5. Since M4 is used by M1, M2, and M3, and M3 is used by M0 and M2, M3 and M4 are used by more than one other module. Using Fenton's interpretation [Fen91], M3 and M4 are the only modules used more than once or reused. Fenton's measure of private reuse is:

$$r(G) = e - n + 1$$

where e is the number of edges and n is the number of nodes in the call graph. This measure is exactly the number of modules which are invoked by more than one other module. Thus, the private reuse in the program represented by Figure 1 is $r(G) = 3$. This measure matches an intuitive notion of reuse; M4 is *reused* twice since it is used by three other modules (M1, M2, and M3) and M3 is *reused* once since it is used by two other modules (M0 and M2). In this case, a reuse occurs on the second or subsequent uses.

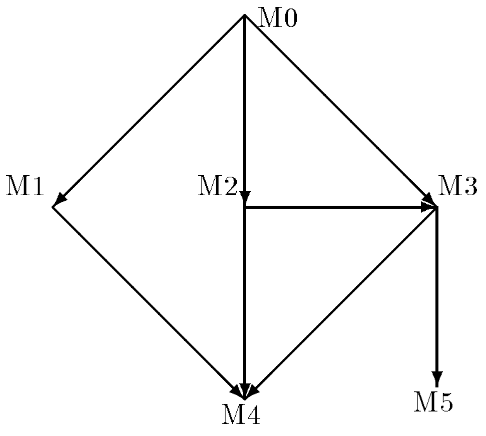

Figure 1: A call graph abstraction

One limitation of the measure is that it is not sensitive to more than one invocation of module A by module B, and is not sensitive to the size of the system. In fact, the call graph abstraction is not an adequate abstraction for developing reuse measures that are sensitive to the number of times that module B invokes module A. An alternative abstraction is a *call multigraph*. A call multigraph has an edge for each invocation of A by B, thus two nodes may be connected by several edges. Figure 2 contains a call multigraph. The figure has two edges connecting M1 and M4, and connecting M3 and M5. These edges represent duplicate calls of M4 by M1 and M5 by M3. Fenton's measure of this graph gives us an $r(G) = 5$ accounting for the two additional reuses.

Both the call graph and call multigraph are not adequate for measuring the relative size or length of the reused software units. If relative size of reused code is important when measuring public reuse, it should be important when measuring private reuse. Perhaps an annotated call multigraph is appropriate.

We can quickly realize that there are many reuse attributes; no one abstraction is appropriate for measuring all of the attributes.

3.2 Verbatim and Leveraged Reuse

Verbatim reuse is reuse without modifications. Importing code from the library without change is verbatim public reuse. Verbatim public reuse is traditionally accomplished with a subroutine activation of program libraries, instantiation of predefined public types, importation of public modules, etc. Using sets of modules as abstractions, we can define verbatim reuse.

$$\forall e[e \in L \wedge e \in N \Rightarrow \text{"}e\text{ is reused verbatim"}\ ]$$

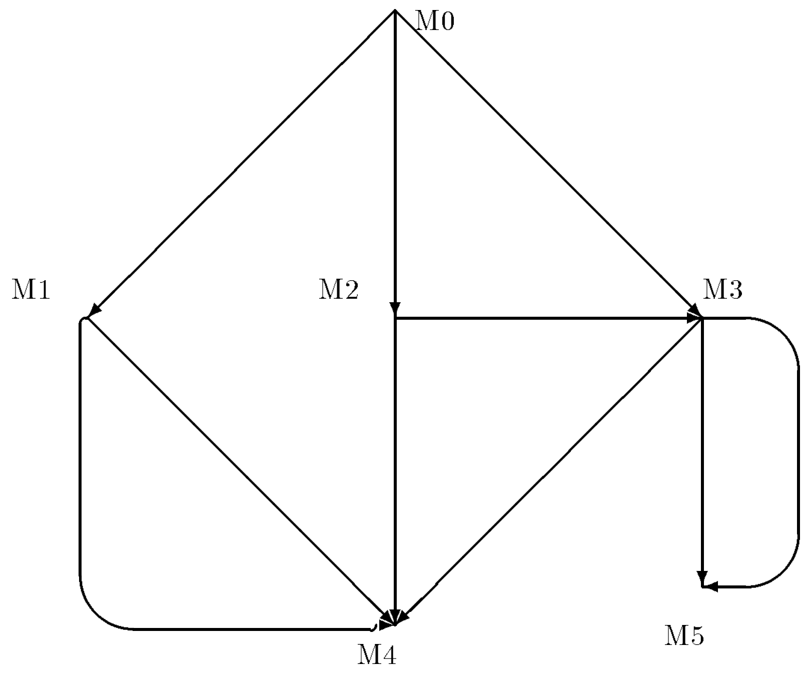

Figure 2: A call multigraph abstraction

We can define measures of reuse in terms of modules (rather than lengths) with the above abstraction. But sets are not adequate abstractions for measuring multiple uses of a library module in the new system. Bags (sets with duplicate entries) may be a better choice.

Verbatim private reuse is essentially the same as verbatim public reuse, except that the reused software entity is not from a library that existed prior to the new product development. Rather, the reused software was developed earlier in the developing the current product. Appropriate abstractions for deriving measures of private verbatim reuse include the call graph and call multigraph as described in Section 3.1.

Leveraged reuse is reuse with modifications. In *leveraged public reuse*, code from L is tailored to a new use. An example of leveraged reuse is when

$$\exists e, f[e \in L \wedge f \in N \wedge f \text{ “is derived from” } e].$$

Leveraged reuse applies to taking an old piece of code and “hacking away at it” until it does what the new requirements demand. We refer to this undisciplined reuse as *ad hoc leveraged reuse*. Leveraged reuse also applies to reuse with more disciplined modifications, and modifications with some language support. In order to measure leveraged reuse we need abstractions which capture the “is derived from” relationship. Again, sets are not adequate. We need to relate components of a new module to components from the library. Thus, we must be able to break modules into smaller units. Here, code statements may be the correct level of abstraction, and reuse can be measured through statement comparisons. However this will not capture modifications to individual statements or expressions. Perhaps only process information — transcripts recording program behavior is effective for measuring ad hoc leveraged reuse. We will see in Section 4 that there are abstractions that can effectively capture leveraged reuse, when leveraged reuse is supported by the language or programming environment.

3.3 Direct and Indirect Reuse

Direct reuse is reuse without going through an intermediate entity. Subroutine calls, instantiations of a type, or importations of modules are direct reuses (or uses). “Direct reuse” is used to distinguish from “indirect reuse.” The previous discussion in this section really refers to direct reuse.

Indirect reuse is reuse through an intermediate entity. When module A invokes module B, and B invokes module C. Then A indirectly reuses C, and C was indirectly reused by A. For *private indirect reuse* a call graph or call multigraph are appropriate abstractions for deriving measures.

The *Distance (or level) of indirection* is the number of intermediate entities between user (client) and used (server). For example assume that A uses B, B uses C, and C uses D. Then for the pair (A, D) there is a *2-level* indirection, or the distance of indirection is 2. It follows that when M_0 uses M_1, ..., M_n uses M_{n+1} then (M_0, M_{n+1}) has an *n-level* indirect reuse. The concept of “distance” or “level” of indirection can also be applied to both verbatim and leveraged reuse. There may be different possible paths connecting two call graph nodes. Thus, there may not be a unique distance of indirect reuse between two modules in a system. For example, in Figure 1, M0 indirectly uses M5 with two distinct reuse paths. The indirect reuse of M5 via the **direct** use of M3 is a *1-level* indirect use, while the reuse of M5 through M2 and M3 is a *2-level* indirect use.

3.4 Reuse Perspectives

Different reuse attributes are visible when reuse is examined from different perspectives. Consider a system where individual modules access some set of existing software entities. When module M uses program unit S, M is a client and S is the server. Thus we consider the program unit being reused as the server, and the unit accessing the server as the client. One can observe reuse from the perspective of the server, the client, and the system. Each of these perspectives is relevant for the analysis and measurement of reuse in a system. We can derive a set of potentially measurable attributes based on profiles of reuse from each perspective. Rather than one measure of one component of a reuse attribute, we end up with measurements of many reuse attributes.

Server Perspective

The *server perspective* is the perspective of the library or a particular library component. Given a particular program entity, the analysis focuses on how this entity is being reused by all of the clients.

A set of reuse measurements can be taken from the server perspective. Such a profile of server reuse describes how the library is reused by clients. The *server reuse profile* can help determine which library components are being reused and in what manner (verbatim, leveraged, directly, indirectly). Potential measures of server reuse attributes include the number of times a library component is reused, average for library, etc

Client Perspective

The *client perspective* takes the point of view of the new system or a new system component. Here we are interested in how a particular program entity reuses other program entities.

Reuse measurements can also be taken from the perspective of the client entity. The *client reusing profile* describes how a new system uses entities in the library.

We can determine the extent that a software entity takes advantage of the reuse library. The reusing profile is focused on the previously implemented code that a program unit, say unit A, takes advantage of. The client reusing profile can include an analysis of the verbatim reuse in A, and profiles the instantiation and references of externally defined program entities. Potential measures include the number of reuse occurrences in new system, the percentage of the new system that is reused code, kinds of reuse, etc.

System Perspective

The *system perspective* is a view of reuse in the overall system, both servers and clients. The analysis is of the reuse throughout an entire system. This may include a synthesis of the reuse in individual clients and servers in the system.

A *system reuse profile* includes both system-wide private reuse, and system-wide public reuse. It includes measurement of indirection, direct, and verbatim reuse. We are interested in both the absolute number of occurrences of various kinds of reuse, and the reuse related to the size of the system (reuse density).

3.5 Language Support & Leveraged Reuse Measurement

Given a library L and new system N, How do we measure the quantity or percentage of ad hoc leveraged reuse? If the language or programming environment does not support reuse, then reuse can be examined either through automatic text comparisons, or records of programmer behavior. Both techniques are not dependable.

With support mechanisms, reuse attributes can be analyzed and measured accurately. Object oriented languages and environments provide such support. In the following section we examine the effect of the object oriented paradigm on reuse attributes, abstractions, and measurement.

4 Object Oriented Reuse

Object oriented languages support reuse through the class system, as well as through the instantiation and use of previously defined entities. Leveraged reuse is supported through inheritance. In object oriented systems it is difficult to distinguish between public and private reuse; a new system is built as an extension to an existing class library. Thus, the ensuing discussion does not distinguish between public and private reuse, and the term "reuse" refers to either one.

4.1 Object Oriented Terminology

Object oriented programming and object oriented languages make use of terminology which is somewhat unique. Here, I introduce some object oriented terminology especially as it relates to software reuse. For a more comprehensive introduction to object oriented software see the excellent texts by Meyer and by Booch [Mey88, Boo91].

Essentially, an object oriented software system is a collection of abstract data types called classes. A *class* is an encapsulated specification of both the persistent state of an abstract data type and its operations. An instantiation or *instance* of a class is an *object*. There may be several concurrently active objects of one class; each instantiation is a different object. Suppose a class defines a stack abstract data type. We can instantiate several stack objects, and each object may contain different values in their stack frames.

Objects perform actions in response to *messages*. These messages are essentially procedure calls. Upon receiving a message an object responds by sending other messages, changing state, and/or returning a message to the object which sent the message. The only way to affect an object is by sending a message. To change the internal state of an object, a message with a specified state changing effect must be sent to the object. Responses to messages are specified via *methods* which are components of classes. Methods are essentially procedures which are local to a class. They may have parameters, assignments to local variables and persistent class variables, and may send other messages. A stack class will contain methods for common stack operations such as *push, pop, top, isempty,* etc.

To reuse an unmodified (verbatim reuse) object oriented software entity, rather than invoke a procedure several times, one sends several messages to the same object. Instantiating a class several times is also a form of verbatim reuse. Verbatim reuse of object oriented software entities is very similar to verbatim reuse of more traditional software. Both the call graph and call multigraph abstractions are appropriate for deriving measures of reuse in object oriented software. Object oriented support of leveraged reuse provides an enhanced ability to analyze and, hopefully, to measure leveraged reuse. I will not address ad hoc leveraged reuse in object oriented systems. Inheritance is a powerful support tool for leveraged reuse and I assume that this mechanism is used.

4.2 Leveraged Reuse via Inheritance

Inheritance provides language support for specifying "I want something just like that except ... ." A developer can modify a particular class to create a new class which behaves somewhat differently than the parent class. The original class is the *superclass* and the new leveraged class is the *subclass.* The subclass can be modified in several basic ways:

- adding new state variables.
- adding new methods.
- changing existing methods.

The first two modifications extend the specification of the superclass, and can be called *extension modifications.* The third modification changes the behavior of a superclass method and can be dubbed an *overload modification.* When creating a subclass, a developer need only specify the differences from the superclass. Clearly, the inheritance mechanism supports reuse of previously written classes.

The class, superclass, subclass hierarchy can be represented by a class hierarchy graph. Figure 3 shows an example class hierarchy graph from a object oriented data base of university records. The figure shows that both "faculty" and "student" inherit certain common data fields (perhaps information such as name, identification number, age, etc.) and operations (such as add/delete person) from the parent or superclass "person." The subclasses "undergrad" and "grad" inherit information and operations from their superclass "student." The subclasses reuse these operations and data through inheritance. However, the reuse is leveraged since (language supported) changes are made to the earlier functionality and data structures. For example, fields for course grades and an operation to create a transcript may be added to Class Person when creating Class Student. The inheritance graph abstraction can be used to describe potential reuse attributes and derive their measures. (Note that many object oriented languages, such as $C++$ support *multiple inheritance*, and so one class may have several superclasses.)

Unfortunately, the class hierarchy graph cannot be used to distinguish between extension and overload modifications. We need to look inside the classes to distinguish between the classes of modifications. The abstraction used to model overload and extension reuse must include the distinction, perhaps through colored edges, in order to derive measures reflecting these leverage categories. For now, I defer the extension of the inheritance graph abstraction to model the classes of modification. In this paper, the inheritance graph serves as the abstraction for the derivation of object oriented leveraged reuse attributes and measures.

As in more traditional software, we can analyze object oriented reuse from the server, client, and system perspective. In each case, we suggest measurable attributes.

4.3 OO Server Reuse Profile

The server reuse profile of a particular class (say class A) will characterize how the class is being reused by the client classes in the system.

The *verbatim server reuse* in an object oriented system is essentially the same as in non object oriented systems. However, we use object oriented terminology. We can measure number of instance creations of objects of class A in a new system and the number of instance references of class A objects in a new system. These attributes can be determined either statically or dynamically.

Leveraged server reuse is supported through inheritance, and is characterized by an analysis of the subclass hierarchy using the inheritance hierarchy graph as the abstraction. We can count the number of subclasses of A, analyze the size and shape of the inheritance hierarchy graph. We can examine the *indirect leveraged server reuse* through an measurements of the inheritance graph; we can find the number of indirect clients and calculate the average distance of indirect leveraged reuses. A count of the number of paths in the inheritance graph between a particular server and its clients is a measure of the instances of indirect reuse. With an extended inheritance graph abstraction we can examine and categorize each instance or reuse. A client can reuse the server either by *extension*, adding methods to the server, or by *overload*, redefining methods.

4.4 OO Client Reusing Profile

The client reusing profile characterizes how a new class A reuses existing library classes. This reuse can be verbatim reuse within A, and measures include the number of instance creations of library objects, the number of instance references of library objects, and the percentage of objects created or referenced by A which are from the library. These verbatim reuses can also be modeled and measured using the call multigraph abstraction. Potential indirect client reuse measures include the number of servers which are indirectly reused, the number of paths to indirect servers, and the lengths of these paths.

The client can leverage its reuse of library class(es) via inheritance. Again, the inheritance hierarchy graph is an appropriate abstraction. We can count the number of servers, the number of indirect servers, the number of paths to indirect servers, and the average lengths of such paths.

Some potential client reuse attributes require an extended inheritance hierarchy graph to distinguish between the classes of reuse. Measures of such attributes include

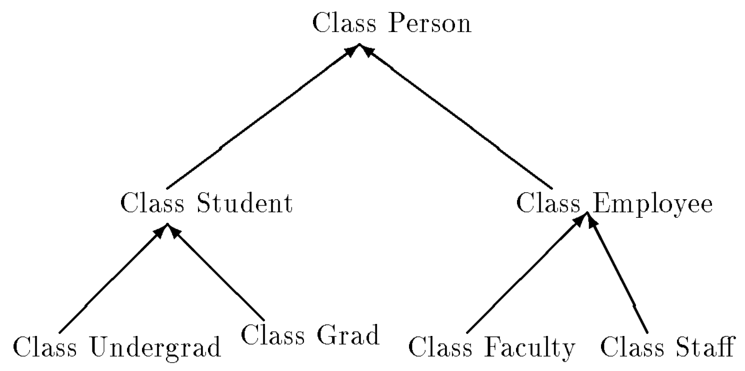

Figure 3: Inheritance hierarchy graph abstraction

the percentage of server methods changed or unchanged, measures of the the kinds of modifications made (whether extension or overloading).

4.5 OO System Reuse Profile

The system profile characterizes overall reuse of library classes in the new system.

Measurable system reuse attributes include

- Percentage of the new system source text imported from the library. This requires information not contained in the call multigraph, information related to class lengths.
- Percentage of new system classes imported verbatim from library.
- Percentage of the new system classes derived (leveraged reuse) from library classes, and the average percentage of these leveraged classes that are imported.
- The average number of verbatim and leveraged clients for servers, and, conversely, the average number of servers for clients.
- The average number of verbatim and leveraged indirect clients for servers, and the average number of indirect servers for clients.
- The average length and number of paths between indirect servers and clients for both verbatim and leveraged reuse.

I am interested in both the reuse of library classes and reuse within the new system. That is, classes developed for the new system that may be completely new or derived from library classes may be reused within the new system. They may be reused verbatim or leveraged. The "shape" of the class/superclass hierarchy is an attribute of the system reuse profile.

Connections between verbatim and leveraged reuse are also part of the system reuse profile. Leveraged server B of client C may have made verbatim use of module A. Thus, there is a reuse chain connecting module A to module C. This chain can be analyzed by examining the two abstractions. Any appropriate measures will make use of both graphs.

4.6 Observations

The process of identifying meaningful and measurable attributes of reuse in object oriented software has not provided just one or two appropriate measures. Rather, I find that there are many reuse properties to examine and to potentially measure. The specific questions that one is trying to answer will help determine which reuse properties to measure. Reuse attributes also depend on the particular perspective of the observer. One can use the point of view of the server, client , or system.

Two abstractions seem especially appropriate for deriving object oriented reuse measures: the call multigraph and the inheritance hierarchy graph. The call multigraph is used to derive measures of verbatim reuse, while the inheritance hierarchy graph is used to derive measures of leveraged reuse. Table 1 shows several measures derived from the two abstractions and three perspectives. Additional abstractions will be necessary to derive reuse measures that are sensitive to the kind of modifications made in leveraged reuse.

The analysis (and measurement) of leveraged reuse is clearly eased by the support of inheritance in object oriented software. With reuse supported by inheritance, we can identify exactly the library source(s) of leveraged reuse and the kind of modifications made to the server(s). This analysis can be made by examining the software documents directly. With ad hoc leveraged reuse, either the programmers must supply the information or we need an elaborate system to track access to the library and editing changes made to library components.

We can still speculate about the relationship between the quantity of reuse and "reusability," assuming that reusability is an internal property of a software document. The level of reuse may be related to internal software properties such as

- Class size: A large class may be harder to understand, and thus more difficult to reuse. Leveraged reuse of a larger sized class may be especially difficult.
- Nature and "complexity" of control flow: A class with a complicated decision structure in the implementation of a class method may also be more difficult to reuse, especially if modifications are necessary.

reuse class	*abstraction*	*perspective*	*candidate measures*
verbatim	call multigraph (CMG)	server	# direct clients # client invocations of server # indirect clients # paths to indirect clients lengths of indirect paths
		client	# direct servers # server instance creations # distinct server instance creations # indirect servers # paths to indirect servers lengths of indirect paths
		system	"size" & "shape" of CMG: $r(CMG) = e - n + 1$ # nodes/edges in CMG # paths in CMG # connected node pairs average indirect distance
leveraged	inheritance hierarchy graph (IHG)	server	# direct clients # direct client uses # indirect clients # paths to indirect clients lengths of indirect paths
		client	# direct servers # server uses # indirect servers # paths to indirect servers lengths of indirect paths
		system	"size" & "shape" of IHG: $r(IHG) = e - n + 1$ # edges in IHG # paths in IHG # connected node pairs average indirect distance

Table 1: Object oriented reuse measures from two abstractions

- Nature and "complexity" of data flow: Many data dependencies may also make reuse more difficult.
- Size and "complexity of interfaces: Many speculate that a large and complicated interface makes reuse more difficult. I suspect that interface complexity will affect direct reuse of a server entity more than the above internal attributes.

The foregoing are all attributes from the server perspective, the perspective of the entity being reused.

The amount of reuse of a particular class is also related to external properties such as the "usefulness" of a class in particular application domains. If the particular functionality of a class is often needed, then the class is more likely to be reused often. Another external attribute is related to the nature of the software development process. If reuse is encouraged by management and/or if reuse is supported by good browsers and information retrieval systems, then reuse is more likely. Thus, the quantity of reuse is not a direct function of the internal properties or internal "reusability" of a class. Because of the numerous external factors, "reusability" is not an attribute of a software document.

Measuring external attributes is generally more difficult than measuring internal attributes. Often needed are measures related to the "ability" of the personnel, the "quality" of the environment, and the "usefulness" of a particular potentially reusable software entity. Yet these are generally subjective attributes, and are not amenable to scientific measurement.

5 Conclusions

In this paper, I introduce the problem of deriving measures of software reuse from a measurement theory perspective. I define several reuse terms including verbatim, leveraged, direct, and indirect reuse. I describe three important perspectives of reuse: the server perspective, the client perspective, and the system perspective. One needs to use well defined perspectives to ensure that reuse measures are taken of the desired attribute.

Using the measurement theory approach, I derive reuse metrics applicable to object oriented systems. Two abstractions are used: a call multigraph and an inheritance hierarchy graph. The call multigraph can be used to define measures of verbatim reuse in both traditional and object oriented systems. The inheritance hierarchy graph is appropriate for defining measures of leveraged reuse when leverage is supported by inheritance. Several candidate measures of object oriented software reuse are proposed. The inheritance hierarch graph abstraction can be extended to allow the definitions of measures of leveraged reuse that take into account the nature of the modifications.

A long term goal of this research is to determine what makes software reusable. Before we can satisfy this ambitious goal, we need to be able to measure both the quantity of reuse and many "independent" variables. These variables include structural attributes of the software entities, and external attributes of the software process. The internal attributes can be readily defined and measured, while measuring the external attributes is problematic. Assuming that we can measure the independent variables, we must derive theories relating the quantity of reuse to the independent variables, and then conduct empirical investigations to support or refute the theories.

This work directed towards measuring the quantity of reuse allows us to monitor reuse levels and is a first step towards our ultimate aim. Determining what makes software reusable remains a long term goal. Measures of the quantity of reuse have immediate, practical applications. Current levels of reuse can be measured and improvements can be monitored.

References

[BB81] J.W. Bailey and V.R. Basili. A meta-model for software development resource expenditures. *Proc. Fifth Int. Conf. Software Engineering*, pages 107–116, 1981.

[BBF+90] A.L. Baker, J.M. Bieman, N. E. Fenton, A. C. Melton, and R.W. Whitty. A philosophy for software measurement. *Journal of Systems and Software*, 12(3):277–281, July 1990.

[Boe81] B. W. Boehm. *Software Engineering Economics.* Prenntice-Hall, Englewood Cliffs, NJ, 1981.

[Boo91] G. Booch. *Object Oriented Design with Applications.* Benjamin/Cummings, 1991.

[CDS86] S.D. Conte, H.E. Dunsmore, and V.Y. Shen. *Software Engineering Metrics and Models.* Benjamin/Cummings, Menlo Park, California, 1986.

[Fen91] Norman Fenton. *Software Metrics A Rigorous Approach.* Chapman & Hall, London, 1991.

[FM90] N. Fenton and A. Melton. Deriving structurally based software measures. *Journal of Systems and Software*, 12(3):177–187, July 1990.

[GC87] R.B. Grady and D.L. Caswell. *Software Metrics: Establishing a Company-wide Program.* Prentice Hall, NJ, 1987.

[Mey87] B. Meyer. Reusability: The case for object-oriented design. *IEEE Software*, 4(2):50–64, March 1987.

[Mey88] B. Meyer. *Object-oriented Software Construction.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[MGBB90] A.C. Melton, D.A. Gustafson, J.M. Bieman, and A.L. Baker. A mathematical perspective for software measures research. *IEE Software Engineering Journal*, 5(5):246–254, 1990.

[Sel89] Richard W. Selby. Quantitative studies of software reuse. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability Vol. II Applications and Experiences*, pages 213–233. Addison-Wesley, 1989.

[Weg84] P. Wegner. Capital-intensive software technology. *IEEE Software*, 1(3):7–45, July 1984.