Candidate Reuse Metrics For Object Oriented and Ada Software[*]

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

To appear in
Proc. IEEE International Software Metrics Symposium, pp. 120-128, May 1993

Technical Report CS–92–142

November 1992

Abstract: Reuse of existing software components plays an important role in increasing the quality of software products and the productivity of software development. The measurement of levels of software reuse is necessary to monitor improvements in software reuse. This paper presents a set of measurable reuse attributes appropriate to object oriented systems, and a suite of metrics which quantify these attributes. Metrics suitable for the object based language Ada are identified and a prototype measurement tool design is proposed.

1 Introduction

The reuse of software holds the promise of increased quality and productivity in software development and maintenance. Software reuse reduces the amount of software that needs to be produced from scratch and thus allows a greater focus on quality. The reuse of well tested software should result in greater reliability and less testing time for new software. With reuse, software development becomes a capital investment. The measurement of reuse will help developers to monitor current levels of reuse and help provide insight in developing software that is easily reused. The level of reuse must be measured in a way that provides information on a wide range of reuse attributes. Current reuse metrics are generally based on only one attribute namely program size or length. There is also a need for research on measuring reuse when the reused software is modified for new uses.

This paper presents a set of measurable reuse attributes, a suite of metrics which quantify these attributes for object oriented systems and proposes a prototype measurement tool design for Ada software. The paper focuses on object-based Ada systems and static reuse measurements. Metric development follows the

[*]Research partially supported by NASA Langley Research Center, Colorado Advanced Software Institute (CASI) and CTA, Inc. CASI is sponsored in part by the Colorado Advanced Technology Institute (CATI), an agency of the state of Colorado. CATI promotes advanced technology teaching and research at universities in Colorado for the purpose of economic development.

guidelines of scientific measurement principles as applied to software [BBF90]. Attributes of reuse measurement are internal product attributes related to properties of particular software document [Fen91]. In the tool design, we stress flexibility, so that users can tailor their metric use to their own goals.

Our aim is to identify a suite of metrics for primitive reuse attributes. These are attributes that are clearly components of the "level of reuse." We do not attempt to combine these components in to composite metrics. Composite metrics are often less sensitive than the primitive metrics and must be defined very carefully [MGBB90]. Thus we focus on developing primitive reuse metrics which can later be used (or combined) to satisfy specific goals and questions of software developers [BS88].

2 Background

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. The purpose of Conte's reuse measure is estimating coding effort. Boehm and Bailey use the size of reused code to adjust cost predictors. Fenton proposed two types of reuse measure: *public reuse* and *private reuse*. He defines public reuse as "the proportion of a product which was constructed externally." To use such a measure one must be able to clearly distinguish between the components that are from external sources and the components that are completely new. He defines private reuse as the "extent to which modules within a product are reused within the same product." He uses the call graph as an abstraction to capture the private reuse and measures it as $r(G) = e - n + 1$ where e is the number of edges and n is the number of nodes in the call graph. Selby addresses the measurement of reuse with modifications. He 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. Reuse measurement is based on a count of the number of modules in each category.

2.1 Object Oriented Reuse

Proponents assert that a major benefit of object oriented or object based design and programming is the generation of reusable software components [Mey87]. Components can be reused as is, or modified using subclassing facilities. To support or refute claims that object oriented or object based software promotes software reuse, one must be able to measure reuse in these systems. Current reuse measures are not directed toward the object oriented approach. New definitions of attributes, abstractions and measures that support data abstraction, information hiding and inheritance constructs are needed to measure reuse in object oriented systems. Bieman defines classes of software reuse, identifies important perspectives of reuse, proposes relevant reuse abstractions, and suggests reuse attributes and associated metrics applicable to object oriented systems [Bie91]. In this paper, we summarize and extend these definitions and metrics.

Reuse can be classified in one of the following ways: *public/private*, *verbatim/generic/leveraged*, and *direct/indirect*. Public reuse is reuse of externally constructed software while private reuse is reuse of software within a product [Fen91]. Verbatim reuse is reuse without modifications. Leveraged reuse is reuse with modifications. These modifications can be either ad hoc modifications (modifications not supported by the programming language) or modifications with some language support. Generic reuse is reuse of generic packages. Generics are simply templates for packages or subprograms. They are general versions of processes that can be modified by parameters at compilation time. Direct reuse is reuse without going through an intermediate entity. Indirect reuse is reuse through an intermediate entity. The levels of indirection is the number of intermediate entities between a client and a server. There may be different possible intermediate entities connecting a client and a server. Object oriented languages support reuse in the following ways:

- verbatim reuse through instantiation and use of previously defined classes

- generic reuse through generic templates
- leveraged reuse through inheritance.

An object oriented software system is a collection of classes which are abstract data types. 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. Objects perform actions in response to messages. If a class *A* uses a class *B* in its method signature or invokes methods of class *B*, then *A* is said to *use B*. A class *A has* a class *B* if an instance of *A* has an instance of *B*. *Use* and *has* relationships allow verbatim reuse. A template or generic class can be instantiated to create new classes. Though it is verbatim in the sense that all the procedures are same, it is modified according to new generic parameter values. Leveraged reuse applies to reuse with any type of modification. Since it is difficult to measure ad hoc leveraged reuse, we consider only leveraged reuse for modifications with some language support. Object oriented support of leveraged reuse via inheritance provides an enhanced ability to analyze and measure leveraged reuse. In object oriented programming a class *B* can inherit from a class *A*. The structure or behavior defined in *A* is shared by *B*. There can be multiple inheritance also. A subclass can augment or redefine the existing structure and behavior of its superclasses. A language without direct support for inheritance is called object based. Ada is object based rather than object oriented and it supports verbatim and generic reuse.

3 Perspectives and Reuse Metrics

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 a server. A program unit being reused is considered a server and the unit accessing that program unit is considered a client. Reuse can be observed from the perspectives of the server, the client, and the system. Each of these perspectives is relevant for the analysis and measurement of reuse in a system. A set of potentially measurable attributes can be derived based on profiles of reuse from each perspective.

In object oriented systems, reuse is not restricted to modules. A class can reuse another class, a global module (or subprogram), a local module, and a class module can reuse another class module. When a class reuses another class, it can inherit from another class, instantiate a generic class, and/or use another class. Thus, measures of the number of servers reused, the number of times a server is reused, the number of clients for a server, size of each server and each client, etc. are important. Size for the relevant metrics can be determined by source lines of code, number of bytes, or any relevant measure. Again, a client can reuse a server either directly or indirectly. When a client A reuses server B and B uses another server C, then A indirectly reuses C, and C was indirectly used by A. Hence, the number of indirect servers, indirect clients, levels of indirection etc., can be measured. All of these types of reuse can be measured from each of the three perspectives. Using each perspective, reuse can be categorized as either verbatim, generic, or leveraged. As a result, we can define numerous measurable attributes. Because of the numerous candidate reuse metrics, we present them in a tabular fashion.

3.1 Client Perspective

The *client perspective* is the perspective of a new system or a new system component. The reuse analysis focuses on how a new class reuses existing library classes or other classes. A client can have verbatim reuse with a server class, can instantiate a generic server class, and/or inherit from a server class. Also, a client can use a global server method and/or repeatedly use a local method that is not exported. Table 1, Table 2, and Table 3 describe reuse measures from the client perspective. Table 1 describes verbatim reuse measures, Table 2 describes generic reuse measures, and Table 3 describes leveraged reuse measures. The definitions of size measures are not given, they can be defined in terms of lines of code, number of bytes, or any relevant size measure.

	Candidate Measures	Definition
	Verbatim	**Using relationship measure**
1.	# Direct server classes	No. of classes included for 'use' and 'has' relationships.
2.	# Indirect server classes	No. of classes that direct servers have the indirect relationships: Using, Instantiation, Inheritance.
3.	# Server instance creations	No. of object instances of each server.
4.	# Server methods	No. of distinct server methods called by clients.
5.	# Server method instances	No. of calls to or usage of each server method.
6.	Size of each server method	
7.	Size of server interface	
8.	Size of global definitions in server interface	
9.	Size of global definitions in server body	
10.	Size of each client method	
11.	Size of client interface	
12.	Size of global definitions in client interface	
13.	Size of global definitions in client body	
14.	# Paths to indirect servers	No. of paths connecting client and indirect servers.
15.	Length of paths to indirect servers	No. of edges in a path connecting client and indirect servers.
16.	# Local server methods	No. of non-generic procedures/functions local to client.
17.	# Local server instance	No. of usage of each local servers.
18.	# Global server methods	No. of non-generic global procedures/functions.
19.	# Global server instance	No. of usage of each global servers.

Table 1: Verbatim reuse measures from client perspective

3.2 Server Perspective

The *server perspective* is the perspective of the library or a particular library component. Given a class, the analysis focuses on how a server is being reused by all of the client classes in the system. A set of reuse measurements can be taken from the server perspective. These measurement can help determine which library components are being reused and in what manner (verbatim, generic, leveraged, directly, indirectly). Table 4 describes reuse measures from the server perspective.

3.3 System Perspective

The system perspective is a view of reuse in the overall system, both servers and clients. It includes both system-wide private reuse, and system-wide public reuse and characterizes overall reuse of library classes in the new system. The measurable system reuse attributes include

- Percentage of the new system source text imported from the library.
- Percentage of new system classes used verbatim from the library and percentage of those library classes that are imported.
- Percentage of new system classes derived from library templates and percentage of those library templates that are imported.
- Percentage of new system classes derived from library classes through inheritance and percentage of those library classes that are imported.
- The average number of verbatim, generic and leveraged clients for servers, and conversely, the average number of servers for clients.

	Candidate Measures	Definition
	Generic	**Generic Instantiation measure**
1.	# Direct server classes	No. of generic classes included for instantiation.
2.	# Indirect server classes	No. of classes that direct servers have the indirect relationships: Using, Instantiation, Inheritance.
3.	# Server instance creations	No. of generic class instances of each generic server.
4.	# Object instance creations	No. of object instances of generic class instances.
5.	# Server methods	No. of distinct server methods called by clients.
6.	# Server method instances	No. of calls to or usage of each server method.
7.	Size of each server method	
8.	Size of server interface	
9.	Size of global definitions in server interface	
10.	Size of global definitions in server body	
11.	Size of generic declarations in server	Size of generic parameters declaration.
12.	Size of each client method	
13.	Size of client interface	
14.	Size of global definitions in client interface	
15.	Size of global definitions in client body	
16.	# Paths to indirect servers	No. of paths connecting client and indirect servers.
17.	Length of paths to indirect servers	No. of relations or edges in a path connecting client and indirect servers.
18.	# Local server methods	No. of generic procedures/functions local to client.
19.	# Local server instances	No. of usage of each local servers.
20.	# Global server methods	No. of global generic procedures/functions.
21.	# Global server instances	No. of usage of each global servers.

Table 2: Generic reuse measures from client perspective

	Candidate Measures	Definition
	Leveraged	**Inheritance relationship measure**
1.	# Direct server classes	No. of direct superclasses.
2.	# Indirect server classes	No. of classes that direct servers have the indirect relationships: Using, Instantiation, Inheritance.
3.	# Indirect parent servers	No. of indirect super classes for client.
4.	# Direct server methods inherited	No. of methods from server class available for client.
5.	# Direct server methods extended	No. of methods in client extended from corresponding server methods.
6.	# Direct server methods overidded	No. of methods in client overridding corresponding server methods.
7.	# Direct server overloaded methods	No. of methods in client overloading server methods.
8.	Size of each direct server method that is reused / extended	
9.	Size of direct server interface	
10.	Size of global definitions in server interface	
11.	Size of global definitions in server body	
12.	Size of each client method	
13.	Size of client interface	
14.	Size of global definitions in client interface	
15.	Size of global definitions in client body	
16.	Paths to indirect servers	No. of paths connecting client and indirect servers.
17.	Length of paths to indirect servers	No. of edges in a path connecting client and indirect servers.
18.	Paths to indirect parent servers	No. of paths connecting client and indirect parent servers.
19.	Length of paths to indirect parent servers	No. of edges in a path connecting client and indirect parent servers.

Table 3: Leveraged reuse measures from client perspective

	Candidate Measures	Definition
	Verbatim	**Using Relationship measure**
1.	# Direct clients	No. of classes that use this server.
2.	# Indirect clients	No. of classes that have the indirect relationships-Using, Instantiation, Inheritance, with direct clients.
3.	# Paths to indirect clients	No. of paths connecting server and indirect clients.
4.	Lengths of paths to indirect clients	No. of edges in a path connecting server and indirect clients.
5.	# Direct client invocations of server	No. of object instances in all clients.
6.	# Client invocations of server method	No. of calls to each server method in all clients.
7.	Size of server interface	
8.	Size of each method in server	
10.	Size of global definitions in server interface	
11.	Size of global definitions in server body	
	Generic	**Generic Instantiation measure**
1.	# Direct clients	No. of client classes instantiating this server.
2.	# Indirect clients	No. of classes that have the indirect relationships-Using, Instantiation, Inheritance with direct clients.
3.	# Paths to indirect clients	No. of paths connecting server and indirect clients.
4.	Lengths of paths to indirect clients	No. of edges in a path connecting server and indirect clients.
5.	# Server instantiations	No. of class instances in all direct clients.
6.	# Direct client invocations of server	No. of object instances in all direct clients.
7.	# Client invocations of server method	No. of calls to each server method in all clients.
8.	Size of server interface	
9.	Size of each method in server	
10.	Size of global definitions in server interface	
11.	Size of global definitions in server body	
12.	Size of generic declarations in server	Size of generic parameters declarations.
	Leveraged	**Inheritance Relationship measure**
1.	# Direct clients	No. of direct subclasses.
2.	# Indirect clients	No. of classes that have the indirect relationships-Using, Instantiation, Inheritance with direct clients.
3.	# Indirect child clients	No. of clients that have inherited indirectly from server.
4.	# Client invocations of server method	No. of times a method in server is reused in all its clients.
5.	Size of server methods	
6.	Size of server interface	
7.	Size of global definitions in server interface	
8.	Size of global definitions in server body	
9.	Paths to indirect clients	No. of paths connecting server and clients.
10.	Length of paths to indirect clients	No. of edges in a path connecting client and server.
11.	Paths to indirect child clients	No. of paths connecting server and child clients.
12.	Length of paths to indirect child clients	No. of edges in a path connecting a server and child client.

Table 4: Reuse measures from server perspective

- The average number of verbatim, generic and leveraged indirect clients for servers, and conversely, the average number of indirect servers for clients.
- The average length and number of paths between servers and clients for verbatim, generic and leveraged reuse.

4 Reuse Measurement in Ada

Ada is designed to support large scale development with reusable components. It supports software portability and reuse. It supports reuse through the following features:

- program units: packages, subprograms, and tasks.
- information hiding: private clauses that allow separation of visible interface specifications and hidden bodies.
- strong type checking: compile time data type checking that improves the ability to reuse components without introducing context errors for data elements in the reused components.
- generic program unit: parameterized templates that allow the generation of software components.
- program libraries: separately compiled reusable program units.

The current version of Ada does not support inheritance. All of the measures defined in Section 3 except leveraged reuse measures are applicable for Ada. However, the expected new version of Ada, Ada 9X [Coh90] provides more kinds of generic parameters, overloading, and inheritance. This paper addresses only object based Ada systems. We want to differentiate between the reuse of library classes from external systems and reuse from within the new system. However Ada does not provide any constructs to differentiate between public library packages and local library packages. Ada packages serve the purpose of object oriented classes.

Reuse can not be measured by a single reuse measure. We need to measure more specific attributes that contribute to the general notion of reuse. So, in Section 3, we define numerous measures. These measures are primitive and are flexible enough to be used in various analyses. To support flexible use of the primitive metrics in future analyses, we specify the construction of a number of tables of primitive metrics. These tables allow users to tailor their metric analysis to their own reuse goals. After creation of these tables, a user can use a spread sheet to analyze reuse.

4.1 Class Using Class Table (CUCT).

The CUCT table supports the measurement of verbatim reuse in classes from both the client and server perspectives. When a class *A* has verbatim use of another class *B*, *A* can instantiate *B* and access the methods of *B*. In the CUCT, each row represents a class in the measured system and each column represents a class that is used by a row class. Each table entry indicates both whether a row class 'uses' the associated column class, and the number of times a column class is instantiated in the row class. By counting the number of entries in each row we can compute the number of direct server classes for each client. The sum of the count of instance creations for each row is the total number of server instance creations. The sum of entries for each column is the number of direct clients of a server class. The sum of the count of instances for each column is the number of direct client invocations of a server. The number of uses of each method of the server class can be obtained from the MMRT table which is described in Section 4.3. Figure 1 is an example call multigraph and Figure 2 is the corresponding CUCT table for that call graph.

4.2 Class Instantiating Generic Class Table (CIGCT).

The CIGCT table supports the measurement of generic reuse from both the client and server perspectives. When a class *A* instantiates a generic class *B*, *A* reuses *B*. In the CIGCT, each row represents a class in the

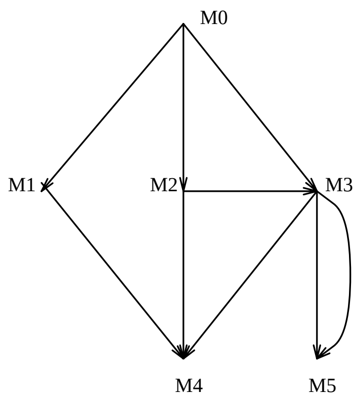

Figure 1: A Call multigraph abstraction

	M0		M1		M2		M3		M4		M5	
M0	0		1	1	1	1	1	1	0		0	
M1	0		0		0		0		1	1	0	
M2	0		0		0		1	1	1	1	0	
M3	0		0		0		0		1	1	1	2
M4	0		0		0		0		0		0	
M5	0		0		0		0		0		0	

Figure 2: A CUCT table

measured system and each column represents a generic class. Each cell has three fields and the first field indicates if a row class instantiates a column class. The second field indicates the number of times a generic column class is instantiated in a row class. The third field indicates the number of times a generic object is instantiated.

The row sum of first field gives the number of direct generic server classes, the row sum of second field gives the number of server instance creations and the row sum of third field gives the number of object instance creations. The column sum of first field gives the number of direct clients for each generic server, the column sum of second field gives the the number of server instantiations in direct clients and the column sum of third field gives the number of client invocations of server. The number of use each method of the column class can be obtained from the MMRT table which is described in Section 4.3.

4.3 Method-Method Reuse Table(MMRT).

The tables CUCT, and CIGCT indicate only class level verbatim and generic reuse. But a class method can reuse another class method, a global method, and/or a local method of that class. The table MMRT indicates the method level reuse from the client and server perspectives. In MMRT (except the first row and column), rows and columns are identified by method names. A method name can be a global procedure/function name or a local procedure/function name or a procedure/function name in a class. Each cell in the first row indicates the size of the corresponding column methods and each cell in the first column indicates the size of the corresponding row methods. The methods of a class that are exported can be identified by the combination of class name, method name. The local methods in a class can be identified by the combination of class name, method name and an identification mark ‘*’. Local methods are accessed only by methods of its corresponding class. The global methods can be identified by the method name and an identification mark ‘**’. Each table entry indicates both whether a row method calls a column method and the number of times a column method is called in the row method.

The first row of the MMRT indicates the size of server methods and the first column indicates the size of the column methods. This enables a user to measure the size of reused code in a new method. The count of the number of entries in each row is the number of server methods for each client method. The sum of the count of server instances for each row gives the total number of calls to server methods. The sum of entries for each column is the number of direct client methods. The sum of the count of server instances for each column is the number of direct client invocations of a server method.

4.4 Class Size Table (CST).

In addition to recording counts of various attributes, we also record the size of the software entities being examined. Thus, we need to measure the size of the reused and reusing code. In the MMRT table, we record the size of methods. In addition, we record the size of the packages or classes using the CST table. All rows in the CST are identified by class names. The table has four columns, one each for the size of class interface, the size of global definitions in the class interface, the size of global definitions in the class body, and the size of generic parameter declarations respectively. The last column is needed only for generic classes. Because visible interface specifications are separated from the hidden bodies in Ada systems, we separate these size measures.

4.5 Flexibility of the Proposed Metrics Tables

We stress flexibility in the tables, so that users can tailor metric analysis for their specific needs. For a given client class, a user can determine the amount of verbatim reuse from the CUCT table, the amount of generic reuse from the CIGCT table, and the amount of both verbatim and generic method reuse from the MMRT table in terms reuse attribute count. Users can also determine the size of the reused code from the MMRT and CST tables. Similarly, a user can measure the reuse of a library class from the tables.

4.6 Proposed Tool

We are developing a prototype tool (in Ada) to collect the proposed measures defined in Section 3 from Ada programs. A first step in automated measurement of metrics analysis is the collection of data item counts. We count the number of occurrences of a specific Ada language feature using a parser. The second step in the automated analysis is the calculation of metric scores through the formulation of the CUCT, CIGCT, MMRT, and CST tables. The third step is the production of various reports based on obtained metrics. The form and content of these reports can be controlled by the user.

Rather than develop a parser and semantic analyzer from scratch, we are reusing the *Anna-I* tool [Men92] to develop our measurement system. Anna is a language extension of Ada which includes features for formally specifying the intended behavior of Ada programs. The Anna-I tool set contains the following:

- Intermediate Representation Toolkit: defines the common internal representation used by all the Anna-I tools.
- Ada/Anna Parser: parses Anna text files (hence Ada text files) generating an Abstract Syntax Tree (AST) representation.
- Semantic Checker: checks the static-semantics of the Ada and Anna code.
- Pretty Printer: generates an ASCII text given an AST.
- AST Browser: It is an X-window based tool for graphically traversing and examining an Anna AST.

We are currently implementing the measurement tool. Once the tool is developed we will collect reuse metrics data to begin empirical studies of reuse. Ada reuse repositories for analysis are available.

5 Conclusions

In this paper, we have identified the client, server and system perspectives for the analysis of software reuse. We have defined an extensive set of primitive reuse metrics from each perspective. These metrics are especially suited to object oriented and object based systems. The metrics can be recorded in a tabular form (CUCT, CIGCT, MMRT, CST) that allows analysts to tailor their metrics use for a wide variety of goals. We are now implementing a tool to generate the reuse measures and metric tables from Ada programs.

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.

[BBF90] 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.

[Bie91] J.M. Bieman. Deriving measures of software reuse in object-oriented systems. *Proc. British Computer Society Workshop on Formal Aspects of Measurement*, 1991, in Formal Aspects of Measurement, pp. 63-83, T. Denvir, R. Herman, and R. W. Whitty (Eds.), Springer-Verlag, 1992.

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

[BS88] V. R. Basili, and H. D. Rombach. The TAME project: Towards improvement-oriented software environments. *IEEE Trans. Software Engineering,* SE-14(6):758-773, June 1988.

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

[Coh90] S. Cohen. Ada 9X project report: Ada support for software reuse. Technical Report SEI-90-SR-16, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PAi 15213, October 1990.

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

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

[Men92] G. O. Mendal. The Anna-I User's Guide and Installation Manual. Stanford University, Computer Systems Lab, ERL 456, Stanford, CA, version 1.4 edition, Sep 1992.

[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] R.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.