Software Reuse and Reusability Metrics and Models

William Frakes
Virginia Tech
Computer Science Department
2990 Telestar Ct.
Falls Church, VA 20165 USA
frakes@sarvis.cs.vt.edu

Carol Terry
Applied Expertise, Inc.
1925 N. Lynn St. Suite 802
Arlington, VA 22209
USA
cterry@aecorp.com

Software Reuse and Reusability Metrics and Models

William Frakes
Virginia Tech
Computer Science Department
2990 Telestar Ct.
Falls Church, VA 20165 USA
frakes@sarvis.cs.vt.edu

Carol Terry
Applied Expertise, Inc.
1925 N. Lynn St. Suite 802
Arlington, VA 22209
USA
cterry@aecorp.com

Abstract: As organizations implement systematic software reuse programs to improve productivity and quality, they must be able to measure their progress and identify the most effective reuse strategies. This is done with reuse metrics and models. In this paper, we survey metrics and models of software reuse and reusability, and provide a classification structure that will help users select them. Six types of metrics and models are reviewed: cost-benefit models, maturity assessment models, amount of reuse metrics, failure modes models, reusability assessment models, and reuse library metrics.

Introduction

Software reuse, the use of existing software artifacts or knowledge to create new software, is a key method for significantly improving software quality and productivity. *Reusability* is the degree to which a thing can be reused. To achieve significant payoffs a reuse program must be systematic [Frakes and Isoda 94]. Organizations implementing systematic software reuse programs must be able to measure their progress and identify the most effective reuse strategies.

In this paper, we survey metrics and models of software reuse and reusability. A metric is a quantitative indicator of an attribute of a thing. A model specifies relationships among metrics. In Figure 1, reuse models and metrics are categorized into six types: 1) reuse cost-benefits models, 2) maturity assessment, 3) amount of reuse, 4) failure modes, 5) reusability, and 6) reuse library metrics. *Reuse cost-benefits* models include economic cost/benefit analysis as well as quality and productivity payoff. *Maturity assessment* models categorize reuse programs by how advanced they are in implementing systematic reuse. *Amount of reuse* metrics are used to assess and monitor a reuse improvement effort by tracking percentages of reuse for life cycle objects. *Failure modes analysis* is used to identify and order the impediments to reuse in a given organization. *Reusability* metrics indicate the likelihood that an artifact is reusable. *Reuse library* metrics are used to manage and track usage of a reuse repository. Organizations often encounter the need for these metrics and models in the order presented.

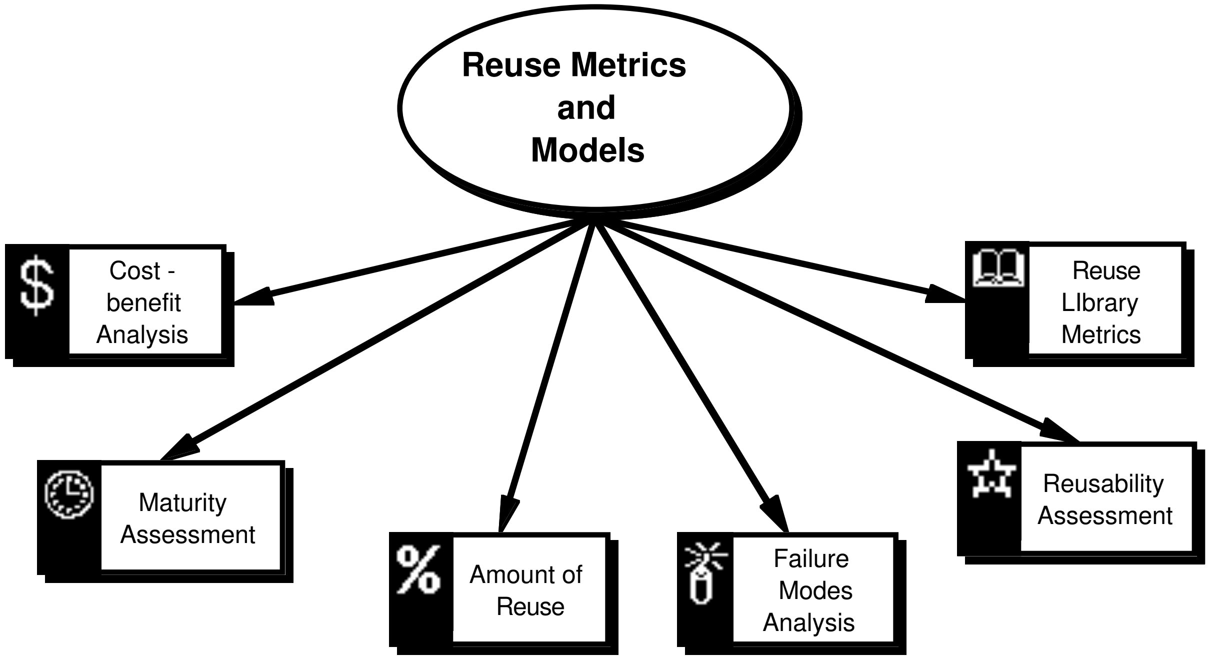

Figure 1: Categorization of Reuse Metrics and Models

Software reuse can apply to any life cycle product, not only to fragments of source code. This means that developers can pursue reuse of requirements documents, system specifications, design structures, and any other development artifact [Barnes and Bollinger 91]. [Jones 93] identifies ten potentially reusable aspects of software projects as follows:

1. architectures	6. estimates (templates)
2. source code	7. human interfaces
3. data	8. plans
4. designs	9. requirements
5. documentation	10. test cases

Table 1: Reusable Aspects of Software Projects

In addition to these lifecycle products, processes (such as the waterfall model of software development and the SEI capability maturity model), and knowledge are also potentially reusable.

Types of Reuse

Clear definitions of types of reuse are necessary prerequisites to measurement. Table 2 provides a faceted classification of reuse definitions gathered from the literature Each column specifies a facet, with the facet name in bold. Below each facet are its corresponding terms. (See Appendix 1 for detailed definitions of the terms in the table.) Terms in parentheses are synonyms. *Development scope* refers to whether the reusable components are from a source external or internal to a project. *Modification* refers to how much a reusable asset is changed. *Approach* refers to different technical methods for implementing reuse. *Domain scope* refers to whether reuse occurs within a family of systems or between families of systems. *Management* refers to the degree to which reuse is done systematically. *Reused entity* refers to the type of the reused object.

Development Scope	**Modification**	**Approach**	**Domain Scope**	**Management**	**Reused Entity**
Internal (Private)	White Box	Generative	Vertical	Systematic (Planned)	Code
External (Public)	Black Box (verbatim)	Compositional	Horizontal	Ad Hoc	Abstract Level
	Adaptive (porting)	In-the-Small			Instance Level
		In-the-Large			Customization Reuse
		Indirect			Generic
		Direct			Source Code
		Carried Over			
		Leveraged			

Table 2: Types of Software Reuse

Reuse in an organization can be defined by selecting appropriate facet-term pairs from this table. For example, an organization might choose to do both internal and external code reuse, allowing only black box modification as part of a compositional approach . They might choose to focus on reuse within their domain (vertical) and to pursue a systematic management approach .

The following sections discuss in detail each of the six types of reuse metrics and models introduced above.

Cost Benefit Analysis

As organizations contemplate systematic software reuse, the first question that will arise will probably concern costs and benefits. Organizations will need to justify the cost and time involved in systematic reuse by estimating these costs and potential payoffs. Cost benefit analysis models include economic cost-benefit models and quality and productivity payoff analyses.

Several reuse cost-benefit models have been reported. None of these models are derived from data, nor have they been validated with data. Instead, the models allow a user to simulate the tradeoffs between important economic parameters such as cost and productivity. These are estimated by setting arbitrary values for cost and productivity measures of systems without reuse, and then estimating these parameters for systems with reuse. There is considerable commonality among the models, as described below.

Cost/Productivity Models

[Gaffney and Durek 89] propose two cost and productivity models for software reuse. The *simple* model shows the cost of reusing software components. The *cost-of-development* model builds upon the simple model by representing the cost of developing reusable components.

The simple model works as follows. Let C be the cost of software development for a given product relative to all new code (for which C=1). R is the proportion of reused code in the product (R<=1). (Note that R is a type of reuse level; this topic is discussed in detail in the "Amount of Reuse" section.) b is the cost relative to that for all new code, of incorporating the reused code into the new product (b=1 for all new code). The relative cost for software development is,

[(relative cost of all new code) * (proportion of new code)] +
[(relative cost of reused software) * (proportion of reused software)]

The equation for this is,

$$C = (1)\ (1-R) + (b)\ (R) = [\ (b - 1)\ R\ ] + 1$$

and the corresponding relative productivity is,

$$P = 1 / C = 1 / ((b-1)R+1).$$

b must be < 1 for reuse to be cost effective. The size of b depends on the lifecycle phase of the reusable component. If only the source code is reused, then one must go through the requirements, design, and testing to complete development of the reusable component. In this case, Gaffney and Durek estimate b=0.85. If requirements, design, and code are reused as well, then only the testing phase must be done and b is estimated to be 0.08.

The cost of development model includes the cost of reusing software and the cost of developing reusable components as follows. Let E represent the cost of developing a reusable component relative to the cost of producing a component that is not reusable. E is expected to be > 1 since creating a component for reuse generally requires extra effort. Let n be the number of uses over which the code development cost will be amortized. The new value for C (cost) incorporates these measures:

$$C = (b + (E / n) - 1)R + 1$$

Other models help a user estimate the effect of reuse on software quality (number of errors) and on software development schedules. Using reusable software generally results in higher overall development productivity, however the costs of building reusable components must be recovered through many reuses. Some empirical estimates of the relative cost of producing reusable components and for cost recovery via multiple reuses are discussed in the next section.

Applications of Cost/Productivity Models

[Margono and Rhoads 93] applied the cost of development model to assess the economic benefits of a reuse effort on a large-scale Ada project (the United States Federal Aviation Administration's Advanced Automation System (FAA/AAS)) . They applied the model to various types of software categorized by the source (local, commercial, or public) and mode of reuse (reused with or without change). The equation for C in the reuse economics model was modified to reflect acquisition, development, and integration costs. Results show that the cost of developing for reuse is often twice the cost of developing an equivalent non-reusable component.

[Favaro 91] utilized the model from [Barnes et. al. 88] to analyze the economics of reuse. (Note: the [Barnes et. al. 88] model is the same as [Gaffney and Durek 89].) The following variables and formulas are relevant:

Variable	Definition
R	% of code contributed by reusable components
b	integration cost of reusable component as opposed to development cost
RC	relative cost of overall development effort
RP	relative productivity
E	relative cost of making a component reusable
N_0	payoff threshold value (number of reuses needed to recover all component development costs)
Formulas:	
	RC = (b+ (E / N) -1) R+1
	RP = 1/RC
	N_0 = E / (1-b)

Table 3: Barnes' and Bollinger's economic investment model

Favaro's research team estimated the quantities R and b for an Ada based development project They found it difficult to estimate R because it was unclear whether to measure source code or relative size of the load modules. The parameter b was even more difficult to estimate because it was unclear whether cost should be measured as the amount of real time necessary to install the component in the application. and whether the cost of learning should be included.

Favaro classified the Booch components according to their relative complexity. The following categories are listed in order of increasing complexity. (The quoted definitions are from [Booch 87].)

monolithic "Denotes that the structure is always treated as a single unit and that its individual parts cannot be manipulated. Monolithic components do not permit structural sharing; stack, string, queue, deque, ring, map, set, and bag components are monolithic."

polylithic "Denotes that the structure is not atomic and that its individual parts can be manipulated. Polylithic components permit structural sharing; lists, trees and graph components are polylithic."

graph "A collection of nodes and arcs (which are ordered pairs of nodes)."

menu, mask End-products of the project. They were developed as generalized, reusable components so were included in the study.

The following table shows values reported by Favaro for E, the relative cost of creating a reusable component, b, the integration cost of a reusable component, and N_0, the payoff threshold value.

	E	b	N_0 for simple implementations	N_0 for complex implementations
Monolithic	1.0	0.10	1.33	2.56
Polylithic	1.2	0.15	1.69	3.40
Graph	1.6	0.25	2.56	5.72
Menu	1.9	0.30	3.25	7.81
Mask	2.2	0.40	4.40	12.97

Table 4: Costs and payoff threshold values for reusable components ***[Favaro 91]***

The overall costs of reusable components relative to non-reusable components is E+b. b is expected to be less than 1.0 since it should be cheaper to integrate a reusable component than to create the component from scratch. E is greater than or equal to 1.0, showing costs of developing reusable components are higher than costs of developing non-reusable components. Favaro found that the cost of reusability increased with the complexity of the component. Monolithic components were so simple there was no extra cost to develop them as reusable components. In contrast, the cost of the mask component more than doubled as it was generalized. The integration cost b was also high in complex applications. The values for N_0 show that the monolithic and polylithic component costs are amortized after only 2 reuses. However, the graph component must be used approximately 5 times before its costs are recovered, and the most complex form of the mask will require 13 reuses for amortization. In summary, costs rise quickly with component size and complexity.

Quality of Investment

[Barnes and Bollinger 91] examined the cost and risk features of software reuse and suggest an analytical approach for making good reuse investments. Reuse activities are divided into *producer* activities and *consumer* activities. Producer activities are reuse investments, or costs incurred while making one or more work products easier to reuse by others. Consumer activities are reuse benefits, or measures in dollars of how much the earlier reuse investment helped or hurt the effectiveness of an activity. The *total reuse benefit* can then be found by estimating the reuse benefit for all subsequent activities that profit from the reuse investment.

The *quality of investment* (Q) is the ratio of reuse benefits (B) to reuse investments (R): Q = B / R. If Q is less than one for a reuse effort, then that effort resulted in a net financial loss. If Q is greater than one, then the investment provided a good return. Three major strategies are identified for increasing Q: 1) increase the level of reuse, 2) reduce the average cost of reuse, and 3) reduce the investment needed to achieve a given reuse benefit.

Business Reuse Metrics

[Poulin et al 93] presents a set of metrics used by IBM to estimate the effort saved by reuse. The study weighs potential benefits against the expenditures of time and resources required to identify and integrate reusable software into a product. While the measures used are similar to the Cost/Productivity Models [Gaffney and Durek 89] discussed above, metrics are named from a business perspective, and they provide a finer breakdown for some calculations. For example, cost is broken down into development costs and maintenance costs.

The metrics are derived from a set of data elements defined in Table 6.

Data Element	Symbol
Shipped source instructions	SSI
Changed source instructions	CSI
Reused source instructions	RSI
Source instructions reused by others	SIRBO
Software development cost	Cost per LOC
Software development error rate	Error rate
Software error repair cost	Cost per error

Table 5: Observable Data ***[Poulin et al 93]***

Given the above data, the following metrics are defined:

- Reuse percent - reflects how much of the product can be attributed to reuse. (This is equivalent to R in the Gaffney and Durek model.) Poulin et al distinguish the reuse percent of a product, the reuse percent of a product release, and the reuse percent for the entire organization.

  Product Reuse percent = RSI / (RSI+SSI) x 100 percent

- Reuse cost avoidance - measures reduced total product costs as a result of reuse (This is equivalent to 1- RC in the Gaffney and Durek model). Poulin et al estimate that the financial benefit attributable to reuse during the development phase is 80 percent of the cost of developing new code, derived from studies showing that the cost of integrating an existing software element is 20 percent of the cost of new development (b in the Gaffney and Durek model). This study also acknowledges savings that are realized in the maintenance phase since reused software generally contains fewer errors; the total reuse cost avoidance is calculated as the sum of cost avoidance in the development and maintenance activities.

Development cost avoidance = RSI x 0.8 x (new code cost)

Service cost avoidance = RSI x (error rate) x (new code cost)

Reuse cost avoidance = Development cost avoidance +
Service cost avoidance

- Reuse value added - a productivity index that, differs from the previous definitions of relative productivity, by including in the definition of reused code: source code that is reused within the product *and* source code that is reused by others. (This is quite similar to internal and external reuse, discussed below in the section on reuse level) .

  Reuse value added = (SSI + RSI + SIRBO) / SSI

- Additional development cost - increased product costs as a result of developing reusable software (same as E in the Barnes and Bollinger model). This study estimates the cost of additional effort at 50 percent of the cost of new development.

  Additional Development Cost = (relative cost of reuse - 1)
  x code written for reuse by others
  x new code cost

Relative cost of writing for reuse is the cost of writing reusable code relative to the cost of writing code for 1-time use (estimated at 1.5). Code written for reuse by others is the kloc of code written for reuse by the initiating project.

Relation of Reuse to Quality and Productivity

Since systematic software reuse is uncommon, empirical evidence relating software reuse to quality and productivity is limited. However, several researchers have accumulated and published statistics that support the notion that software reuse improves quality and productivity.

[Agresti and Evanco 92] conducted a study to predict defect density, a software quality measurement, based on characteristics of Ada designs. They used 16 subsystems from the Software Engineering Laboratory (SEL) of NASA Goddard Space Flight Center. The SEL project database provided data on the extent of reuse and subsystem identification for each compilation unit, and reported defects and non defect modifications. Collectively, approximately 149 KSLOC (kilo-source lines of code) were analyzed. The project database showed that the reuse ratios (fraction of compilation units reused verbatim or with slight modification, <= 25% of lines changed) were between 26% and 28%. Defect densities were between 3.0 and 5.5 total defects per KSLOC. Four sample rows from a table summarizing the project characteristics of the subsystems show that a high level of reuse correlates with a low defect density (size is in KSLOC units):

Subsystem	Software size	Library units	Compilation units	Reuse	Defect Density
1-5	27.3	38	185	0.44	6.2
1-6	5.7	18	73	0.94	1.6
2-4	6.9	23	60	0.74	1.4
3-3	3.5	12	66	0.09	8.0

Table 6: Characteristics of SEL subsystems [Agresti and Evanco 92]

The Reusability-Oriented Parallel programming Environment (ROPE) [Browne et al 90] is a software component reuse system that helps a designer find and understand components. ROPE is integrated with a development environment called CODE (Computation-Oriented Display Environment), which supports construction of parallel programs using a declarative and hierarchical graph model of computation. ROPE supports reuse of both design and code components, focusing on the key issues of reusability: finding components, then understanding, modifying, and combining them. An experiment was conducted to investigate user productivity and software quality for the CODE programming environment, with and without ROPE.

The experimental design included metrics such as fraction of code in a program consisting of reused components, development time and error rates. Reuse rates were reported as "extremely high" for the 43 programs written using ROPE, with a mean reuse rate for a total program (code and design) of 79%. The researchers used total development time to measure the effect of reusability on productivity. Table 7 shows the development time in hours for subjects programming in the CODE environment and those programming in CODE and ROPE. The data reveals that ROPE had a significant effect on development time for all of the experimental programs.

Program Name	Using CODE only		Using CODE and ROPE	
Convex Hull	12.4	[9.0,15.8]	2.2	[2.0,2.5]
Readers/Writers	4.7	[3.4,6]	1.8	[1.2,2.4]
Producer/Consumer	3.9	[3.6,4.3]	1.9	[1,2.8]
Shortest Path	33.3	[16,51]	1.4	[0.7,2.1]
Parallel Prefix	20	N/A	1.4	[0.9,1.8]
Divide Region	20	N/A	3.5	[2.5,4.5]
Sort/Merge	8.5	N/A	1.5	N/A

Table 7: Mean Development Time and [95% Confidence Intervals in Hours] [Browne et al 90]

Error rates were used to measure quality. Compile errors, execution errors, and logic errors were all counted. The results are shown in Table 8. The use of ROPE reduced error rates, but the data is less clear than that for productivity. The researchers attribute this to the difficulty of collecting the data and to the lack of distinction between design and code errors.

Program Name	Using CODE only		Using CODE and ROPE	
Convex Hull	8.8	[4.3,13.3]	3.1	[1.4,4.4]
Readers/Writers	14.5	[5.4,23.6]	1.2	[.4,2.0]
Producer/Consumer	4.3	[1.9,6.7]	.3	[0,.7]
Shortest Path	10	N/A	4	N/A
Parallel Prefix	20	N/A	2.5	N/A
Divide Region	5	N/A	17	N/A
Sort/Merge	7	N/A	3	N/A

Table 8: Mean Number of Errors and 95% Confidence Intervals [Browne et al 90]

In summary, the CODE/ROPE experiment showed a high correlation between the measures of reuse rate, development time, and decreases in number of errors.

[Card et al 86] conducted an empirical study of software design practices in a Fortran-based scientific computing environment. The goals of the analysis were to identify the types of software that are reused and to quantify the benefits of software reuse. The results were:

- The modules that were reused without modification tended to be small and simple, exhibiting a relatively low decision rate.
- Extensively modified modules tended to be the largest of all reused software (rated from extensively modified to unchanged) in terms of the number of executable statements.
- 98 percent of the modules reused without modification were fault free and 82 percent of them were in the lowest cost per executable statement category.
- These results were consistent with a previous *Software Engineering Laboratory* study [Card et al 82] which showed that reusing a line of code costs only 20 percent of the cost of developing it new.

Matsumura [Frakes 91] describes an implementation of a reuse program at Toshiba. Results of the reuse program show a 60% ratio of reused components and a decrease in errors by 20 to 30%. Managers felt that the reuse program would be profitable if a component was reused at least three times.

A study of reuse in the object-oriented environment was conducted by Chen and Lee [Chen and Lee 93]. They developed an environment, based on an object-oriented approach, to design, manufacture and use reusable C++ components. A controlled experiment was conducted to substantiate the reuse approach in terms of software productivity and quality. Results showed improvements in software productivity of from 30 to 90% measured in lines of code developed per hour (LOC/hr).

The Cost/Productivity Model by [Gaffney and Durek 89] specifies the effect of reuse on software quality (number of errors) and on software development schedules. Results suggest that trade-offs can occur between the proportion of reuse and the costs of developing and using reusable components. In a study of the latent error content of a software product, the relative error content decreased for each additional use of the software but leveled off between 3 and 4 uses. The models show that the number of uses of the reusable software components directly correlates to the development product productivity. Gaffney and Durek believe that the costs of building reusable parts must be shared across many users to achieve higher payoffs from software reuse.

Maturity Assessment

Reuse maturity models support an assessment of how advanced reuse programs are in implementing systematic reuse, using an ordinal scale of reuse phases. They are similar to the Capability Maturity Model developed at the Software Engineering Institute (SEI) at Carnegie Melon University [Humphrey 89]. A maturity model is at the core of planned reuse, helping organizations understand their past, current, and future goals for reuse activities . Several reuse maturity models have been developed and used, though they have not been validated.

Koltun and Hudson Reuse Maturity Model

[Koltun and Hudson 91] developed the reuse maturity model shown in Table 9. Columns indicate phases of reuse maturity, assumed to improve along an ordinal scale from 1 (Initial/Chaotic) to 5 (Ingrained). Rows correspond to dimensions of reuse maturity such as Motivation/Culture and Planning for Reuse. For each of the ten dimensions of reuse, the amount of organizational involvement and commitment increases as an organization progresses from initial/chaotic reuse to ingrained reuse. Ingrained reuse incorporates fully automated support tools and accurate reuse measurement to track progress.

	1 Initial/ Chaotic	2 Monitored	3 Coordinated	4 Planned	5 Ingrained
Motivation/ Culture	Reuse discouraged	Reuse encouraged	Reuse incentivized re-enforced rewarded	Reuse indoctrinated	Reuse is the way we do business
Planning for reuse	None	Grassroots activity	Targets of opportunity	Business imperative	Part of strategic plan
Breadth of reuse	Individual	Work group	Department	Division	Enterprise wide
Responsible for making reuse happen	Individual initiative	Shared initiative	Dedicated individual	Dedicated group	Corporate group with division liaisons
Process by which reuse is leveraged	Reuse process chaotic; unclear how reuse comes in.	Reuse questions raised at design reviews (after the fact)	Design emphasis placed on off the shelf parts	Focus on developing families of products	All software products are genericized for future reuse
Reuse assets	Salvage yard (no apparent structure to collection)	Catalog identifies language and platform specific parts	Catalog organized along application specific lines	Catalog includes generic data processing functions	Planned activity to acquire or develop missing pieces in catalog
Classification activity	Informal, individualized	Multiple independent schemes for classifying parts	Single scheme catalog published periodically	Some domain analyses done to determine categories	Formal, complete, consistent timely classification
Technology support	Personal tools, if any	Many tools, but not specialized for reuse	Classification aids and synthesis aids	Electronic library separate from development environment	Automated support integrated with development environment
Metrics	No metrics on reuse level, pay-off, or costs	Number of lines of code used in cost models	Manual tracking of reuse occurrences of catalog parts	Analyses done to identify expected payoffs from developing reusable parts	All system utilities, software tools and accounting mechanisms instrumented to track reuse
Legal, contractual, accounting considerations	Inhibitor to getting started	Internal accounting scheme for sharing costs and allocating benefits	Data rights and compensation issues resolved with customer	Royalty scheme for all suppliers and customers	Software treated as key capital asset

Table 9: Hudson and Koltun Reuse Maturity Model

To use this model, an organization will assess its reuse maturity before beginning a reuse improvement program by identifying their placement on each dimension. (In our experience, most organizations are between Initial/Chaotic and Monitored at the start of their program.) The organization will then use the model to guide activities that must be performed to achieve higher levels of reuse maturity. Once an organization achieves Ingrained reuse, reuse becomes part of the business routine and will no longer be recognized as a distinct discipline.

SPC Reuse Capability Model

The reuse capability model developed by the Software Productivity Consortium [Davis 93] has two components: an assessment model and an implementation model.

The assessment model consists of a set of categorized critical success factors (stated as goals) that an organization can use to assess the present state of their reuse practice. The factors are organized into four primary groups: management, application development, asset development, and process and technology factors. For example, in the group "Application Development Factors/Asset Awareness and Accessibility" is the goal "Developers are aware of and reuse assets which are specifically acquired/developed for their application."

The implementation model helps prioritize goals and build successive stages of implementation. The SPC identifies four stages in the risk-reduction growth implementation model:

1. Opportunistic	The reuse strategy is developed on the project level. Specialized reuse tools are used and reusable assets are identified.
2. Integrated	A standard reuse strategy is defined and integrated into the corporation's software development process. The reuse program is fully supported by management and staff. Reuse assets are categorized.
3. Leveraged	The reuse strategy expands over the entire life cycle and is specialized for each product line. Reuse performance is measured and weaknesses of the program identified.
4. Anticipating	New business ventures take advantage of the reuse capabilities and reusable assets. High payoff assets are identified. The reuse technology is driven by customers' needs.

The SPC continues to evolve the model. It has been used in pilot applications by several organizations, but no formal validation has been done.

Amount of Reuse

Amount of reuse metrics are used to assess and monitor a reuse improvement effort by tracking percentages of reuse of life cycle objects. over time. In general, the metric is:

$$\frac{\text{amount of lifecycle object reused}}{\text{total size of lifecycle object}}$$

A common form of this metric is based on lines of code as follows:

$$\frac{\text{lines of reused code in system or module}}{\text{total lines of code in system or module}}$$

[Frakes 90], [Terry 93] and [Frakes and Terry 94] extend the basic "amount of reuse" metric by defining *reuse level* metrics that include factors such as abstraction level of the lifecycle objects and formal definitions of internal and external reuse. Work is also underway ([Bieman and Karunanithi 93], [Chidamber and Kemerer 94]) to define metrics specifically for object-oriented systems. The following sections discuss this work in detail. To date, little work has been done on amount of reuse metrics for generative reuse, though [Biggerstaff 92] implicitly defines such a metric by reporting a ratio of generated source lines to specification effort for the Genesis system—40 KSLOC for 30 minutes specification effort.

Reuse Level

The basic dependent variable in software reuse improvement efforts is the level of reuse[Frakes 93]. Reuse level measurement assumes that a system is composed of parts at different levels of abstraction. The levels of abstraction must be defined to measure reuse. For example, a C based system is composed of modules (.c files) that contain functions, and functions that contain lines of code. The reuse level of a C based system, then, can be expressed in terms of modules, functions, or lines of code. A software component (lower level item) may be internal or external. An internal lower level component is one developed for the higher level component. An external lower level component is used by the higher level component , but was created for a different item or for general use.

The following quantities can be calculated given a higher level item composed of lower level items:

L= the total number of lower level items in the higher level item.

E = the number of lower level items from an external repository in the higher level item.

I = the number of lower level items in the higher level item which are not from an external repository.

M = the number of items not from an external repository which are used more than once.

These counts are of unique items (types) not tokens (references).

Given these quantities, the following reuse level metrics are defined[Frakes 90]:

External Reuse Level: E / L

Internal Reuse Level: M / L

Total Reuse Level: External Reuse Level + Internal Reuse Level

Internal, external, and total reuse levels will assume values between 0 and 1. More reuse occurs as the reuse level value approaches 1. A reuse level of 0 indicates no reuse.

The user must provide information to calculate these reuse measures. The user must define the abstraction hierarchy, a definition of external repositories, and a definition of the "uses" relationship. For each part in the parts-based approach, we must know the name of the part, source of the part (internal or external), the level of abstraction, and the amount of usage.

[Terry 93] extended this formal model, by adding a variable reuse *threshold level* which had been arbitrarily set to one in the original model The variables and reuse level metrics are:

ITL=	internal threshold level, the maximum number of times an internal item can be used before it's reused.
ETL=	external threshold level, the maximum number of times an external item can be used before it's reused.
IU=	number of internal lower level items which are used more than ITL.
EU=	number of external lower level items which are used more than ETL.
T =	total number of lower level items in the higher level item, both internal and external.

Internal reuse level:	IU / T
External reuse level:	EU / T
Total reuse level:	(IU + EU) / T

The *reuse frequency* metric is based on references (tokens) to reused components rather than counting components only once as was done for reuse level. The variables and reuse frequency metrics are:

IUF=	number of references in the higher level item to reused internal lower level items.
EUF=	number of references in the higher level item to reused external lower level items.
TF =	total number of references to lower level items in the higher level item, both internal and external.

Internal reuse frequency:	IUF / TF
External reuse frequency:	EUF / TF
Total reuse frequency:	(IUF + EUF) / TF

Program size is often used as a measure of complexity The complexity weighting for internal reuse is the sum of the sizes of all reused internal lower level items divided by the sum of the sizes of all internal lower level items within the higher level item. An example size weighting for internal reuse in a C system is the ratio of the size (calculated in number of lines of non-commentary source code) of reused internal functions to the size of all internal functions in the system.

The software tool *rl* calculates reuse level and frequency for C code . Given a set of C files, rl reports the following information:

1. internal reuse level
2. external reuse level
3. total reuse level
4. internal reuse frequency
5. external reuse frequency
6. total reuse frequency
7. complexity (size) weighting for internal functions

The user may specify an internal and external threshold level. The software allows multiple definitions of higher level and lower level abstractions. The allowed higher level abstractions are system, file or function. The lower level abstractions are function and NCSL (Non-Commentary Source Lines of code).

Reuse Metrics for Object-Oriented Systems

[Bieman 92][Bieman and Karunanithi 93] have proposed reuse metrics for object-oriented systems. [Bieman 92] identifies three perspectives from which to view reuse: the server perspective, the client perspective, and the system perspective. The server perspective is the perspective of the library or a particular library component. The analysis focuses on how the entity is reused by the clients. From the client perspective, the goal is knowing how a particular program entity reuses other program entities. The system perspective is a view of reuse in the overall system, including servers and clients.

The server reuse profile of a class will characterize how the class is reused by the client classes. The verbatim server reuse in an object oriented system is basically the same as in procedural systems, using object oriented terminology. Leveraged server reuse is supported through inheritance. A client can reuse the server either by extension, adding methods to the server, or by overload, redefining methods. (*Note*: [McGregor and Sykes 92] offers good definitions of the object-oriented terminology used in this section.)

The client reuse profile characterizes how a new class reuses existing library classes. It too can be verbatim or leveraged, with similar definitions to the server perspective.

Measurable system reuse attributes include:

- % of new system source text imported from the library
- % of new system classes imported verbatim from the library
- % of new system classes derived from library classes and the average % of the leveraged classes that are imported
- average number of verbatim and leveraged clients for servers, and servers for clients
- average number of verbatim and leveraged indirect clients for servers, and indirect servers for clients
- average length and number of paths between indirect servers and clients for verbatim and leveraged reuse

[Bieman and Karunanithi 93] describe a prototype tool which is under development to collect the proposed measures from Ada programs. This work recognizes the differences between object oriented systems and procedural systems, and exploits those differences through unique measurements.

[Chidamber and Kemerer 94] proposes a metrics suite for object oriented design. The paper defines six metrics based on measurement theory:

1. Weighted methods per class
2. Depth of inheritance tree
3. Number of children
4. Coupling between object classes
5. Response for a class

Among these, the most significant to reuse is the metric Depth of Inheritance Tree. This metric calculates the length of inheritance hierarchies. Shallow hierarchies forsake reusability for the simplicity of understanding, thus reducing the extent of method reuse within an application. Chidamber and Kemerer assert that this metrics suite can help managers manage reuse opportunities by measuring inheritance.

Reuse Predictions for Lifecycle Objects

[Frakes and Fox 95] present models that allow the prediction of reuse levels for one lifecycle object based on reuse levels for other lifecycle objects. Such models can be used to estimate reuse levels of later lifecycle objects such as code from earlier ones like requirements. The authors define both temporal and non-temporal models, and discuss methods for tailoring the models for a specific organization. The models are based on reuse data collected from 113 respondents from 29 organizations, primarily in the US. The study found that there were significant correlations between the reuse levels of lifecycle objects, and that prediction models improved as data from more lifecycle objects were added to the models.

Software Reuse Failure Modes Model

Implementing systematic reuse is difficult, involving both technical and non-technical factors. Failure modes analysis provides an approach to measuring and improving a reuse process based on a model of the way a reuse process can fail. The reuse failure modes model reported by [Frakes and Fox, 94] can be used to evaluate the quality of a systematic reuse program, to determine reuse impediments in an organization, and to devise an improvement strategy for a systematic reuse program.

Given the many factors that may affect reuse success, how does an organization decide which ones to address in its reuse improvement program? This question can be answered by finding out why reuse is not taking place in the organization. This can be done by considering reuse failure modes—that is, the ways that reuse can fail.

The reuse failure modes model has seven failure modes corresponding to the steps a software engineer will need to complete in order to reuse a component. The failure modes are:

No Attempt to Reuse
Part Doesn't Exist
Part Isn't Available
Part Isn't Found
Part Isn't Understood
Part Isn't Valid
Part Can't Be Integrated

Each failure mode has failure causes associated with it. “No attempt to reuse”, has among its failure causes, for example, resource constraints, no incentive to reuse, and lack of education. To use the model, an organization gathers data on reuse failure modes and causes, and then uses this information to prioritize its reuse improvement activities.

Reusability Assessment

Another important reuse measurement area concerns the estimation of reusability for a component. Such metrics are potentially useful in two key areas of reuse: reuse design and re-engineering for reuse. The essential question is, are there measurable attributes of a component that indicate its potential reusability? If so, then these attributes will be goals for reuse design and re-engineering. One of the difficulties in this area is that attributes of reusability are often specific to given types of reusable components, and to the languages in which they are implemented. In this section we review work in this area.

In a study of NASA software, [Selby, 1989] identified several module attributes that distinguished black box reuse modules from others in his sample. The attributes included:

- fewer module calls per source line
- fewer I/O parameters per source line
- fewer read/write statements per line
- higher comment to code ratios
- more utility function calls per source line
- fewer source lines

This data suggests that modules possessing the above attributes will be more reusable.

[Basili et al 90] reported on two reuse studies of systems written in Ada. The first study defined measures of data bindings to characterize and identify reusable components. Data binding is the sharing of global data between program elements [Hutchins and Basili 85]. [Basili et al 90] proposed a method to identify data bindings within a program. After identification of the bindings, a cluster analysis computes and weights coupling strengths. A coupling is based on references to variables and parameters (data bindings). Aliasing, or referencing, is not taken into account; only one level of data bindings is considered. The cluster analysis identifies which modules are strongly coupled and may not be good candidates for reuse, and which modules are found to be independent of others and are potentially reusable. A similar use of module coupling information to identify potential objects in C code has been reported by [Knight and Dunn, 1991]

The second study defines an abstract measurement of reusability of Ada components. Potentially reusable software is identified, and a method to measure distances from that ideal is defined. By measuring the amount of change necessary to convert an existing program into one composed of maximally reusable components, an indication of the reusability of the program can be obtained.

Reuse Library Metrics

As can be seen in Figure 2, a reuse library is a repository for storing reusable assets, plus an interface for searching the repository. Library assets can be obtained from existing systems through re-engineering, can be designed and built from scratch, or can be purchased. Components then are usually *certified*, a process for assuring that they have desired attributes such as testing of a certain type. The components are then classified so that users can effectively search for them. The most common classification schemes are enumerated, faceted, and free text indexing [Frakes and Gandel, 90]. The evaluation criteria for indexing schemes of reuse libraries are: costs, searching effectiveness, support for understanding, and efficiency.

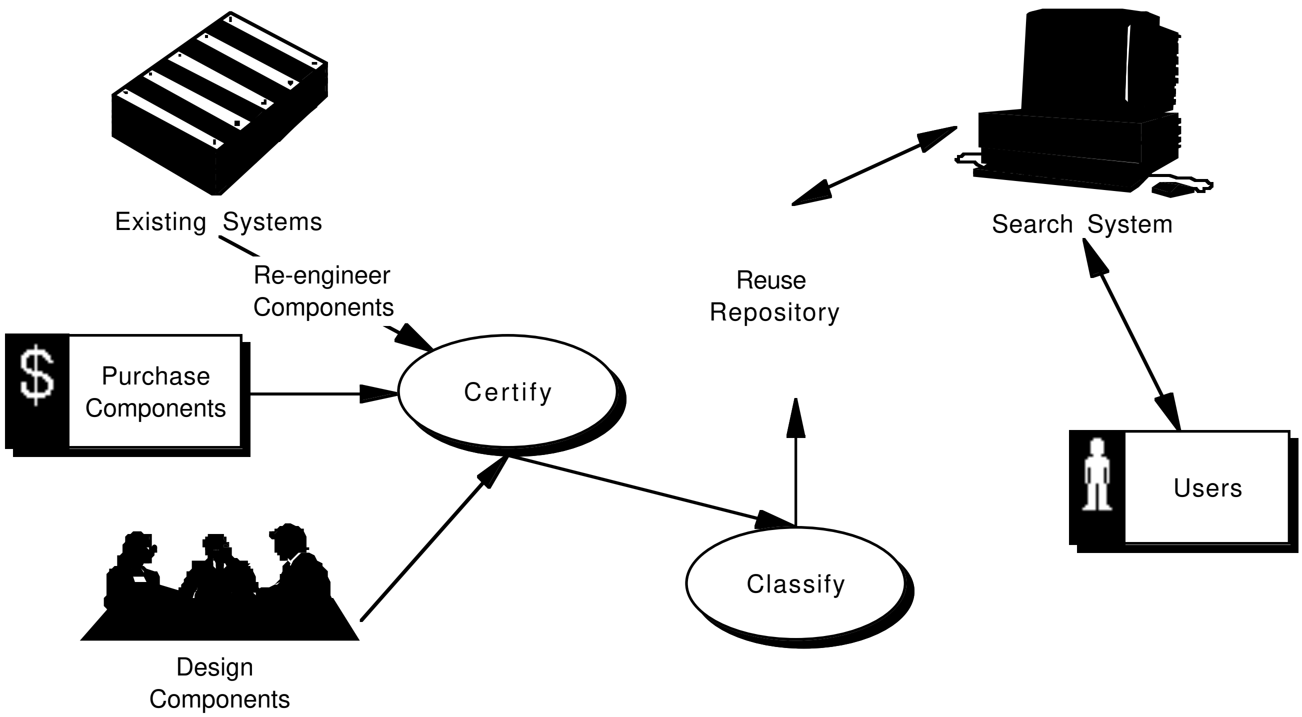

Figure 2: A Reuse Library System

Indexing costs include the cost of creating a classification scheme, maintaining the classification scheme, and updating the database for the scheme. These concerns are negligible for a small collection, but become more important as the collection grows. Each of these costs can be measured in terms of dollars or effort. These costs can be normalized by dividing total costs by the number of components handled by the library.

Searching effectiveness addresses how well the classification methods help users locate reusable components. Searching effectiveness is usually measured with recall and precision. Recall is the number of relevant items retrieved divided by the number of relevant items in the database. The denominator is generally not known and must be estimated using sampling methods. Precision is the number of relevant items retrieved divided by total of items retrieved. Another effectiveness measure is overlap, which reports the percentage of relevant documents retrieved jointly by two methods. [Frakes and Pole, 94], in a study of representation methods for reusable software, reported no statistically significant difference in terms of recall and precision between enumerated, faceted, free text keyword, and attribute value classification schemes. They reported high overlap measures ranging from .72 to .85 for all pairs of the classification methods.

To reuse a component, a software engineer must not only find it, but also understand it. *Understanding metrics* measure how well a classification method helps users understand reusable components. [Frakes and Pole, 94] used a 7 point ordinal scale (7 = best) to measure support for understanding. They reported no significant difference between the classification methods using this metric.

In order to be useful, a reuse library must also be efficient. Library efficiency deals with non-functional requirements such as memory usage, indexing file size, and retrieval speed. Memory usage can be measured by the number of bytes needed to store the collection. Indexing overhead ratios, calculated by dividing the sum of the size of the raw data and indexing files by the size of the indexing files. Retrieval speed is usually calculated by measuring the time it takes the system to execute a search of a given query on a given database.

Quality of assets is another important aspect of a reuse library. There is considerable anecdotal evidence that this is the most important factor in determining successful use of a reuse library. [Frakes and Nejmeh, 87] proposed the following metrics as indicators of the quality of assets in a reuse library.

- Time in Use - the module should have been used in one or more systems that have been released to the field for a period of three months.
- Reuse Statistics - the extent to which the module has been successfully reused by others is perhaps the best indicator of module quality.
- Reuse Reviews - favorable reviews from those that have used the module is a good indication that the module is of high quality.
- Complexity - overly complex modules may not be easy to modify or maintain.
- Inspection - the module should have been inspected.

- Testing - the modules should have been thoroughly tested at the unit level with statement coverage of 100% and a branch coverage of at least 80%.

Another class of reuse library metrics is used to measure usage of a reuse library system. The following list was supplied by the ASSET system, an on-line commercial reuse library system.

- Time on-line (system availability) - this is a measure of the number of hours the system is available for use.
- Account Demographics - this assigns users to the following categories:
- Gov't/Contractor, Commercial, Academia, Non-US
- Type of library function performed : Searches, Browses, Extracts
- Asset Distribution: whether electronic or via a human librarian
- Number of Subscriber Accounts
- Available Assets by type: Interoperation , Supplier Listings, Local Components
- Number of Extractions by Collection: Number of Assets Extracted by Collection, Number of Assets Extracted by Evaluation Level
- Listing of Assets by Domain
- Request for Services by: Telnet Logins, Modem Logins, FTP, World Wide Web

These metrics can provide good management information for a library system. They can be used to demonstrate the value of the library to management as well as to provide information for continuous quality improvement.

Summary

A reuse program must be planned, deliberate, and systematic if it is to give large payoffs. As organizations implement systematic software reuse programs in an effort to improve productivity and quality, they must be able to measure their progress and identify the most effective reuse strategies. In this paper, we surveyed metrics and models of software reuse. A metric is a quantitative indicator of an attribute. A model specifies relationships between metrics.

Table 10 presents a summary of the reuse metrics and models discussed in this paper. As is typical in an emerging discipline such as systematic software reuse, many of these metrics and models still lack formal validation. Despite this, they are being used and are being found useful in industrial practice.

Table 10: Summary of Software Reuse Metrics and Models

	Source	Description
Cost-benefit Analysis		
Cost/Productivity Models	Gaffney, Durek [Gaffney and Durek 89]	*Simple model:* Let C=cost of software development. R=proportion of reused code in the product. b=cost relative to that of all new code of incorporating reused code into the product. Then C=(b-1)R + 1 and productivity P=1/C. *Cost of development model:* Let E=cost of developing a reusable component relative to the cost of producing a component that is not to be reused. Let n be the number of uses over which code cost will be amortized. Then C (cost) is C=(b + E/n-1)R+1.
Quality of Investment	Barnes, Bollinger [Barnes and Bollinger 91]	Quality of investment (Q) is the ratio of reuse benefits (B) to reuse investments (R): Q = B/R. If Q<1 then the reuse effort resulted in a net loss. If Q>1 then the investment provided good returns.
Business Reuse Metrics	Poulin, Caruso, Hancock [Poulin et al 93]	Reuse metrics that distinguish the savings and benefits of reuse are defined: 1. reuse percent 2. reuse cost avoidance 3. reuse value added 4. additional development cost The metrics are used to estimate return on investment.
Maturity Assessment		
Reuse Maturity Model	Koltun, Hudson [Koltun and Hudson 91]	Levels an organization proceeds through working toward effective software reuse: 1. Initial/Chaotic 4. Planned 2. Monitored 5. Ingrained 3. Coordinated
Reuse Capability Model	Software Productivity Consortium [Davis 93]	The Software Productivity Consortium identifies four stages in the risk-reduction growth implementation model for software reuse: 1. Opportunistic 3. Leveraged 2. Integrated 4. Anticipating
Amount of Reuse		
Reuse Level	[Frakes 90] [Terry 93] [Frakes, Terry 94]	Assume a system consists of parts where a higher level item is composed of lower level items. The internal reuse level of a higher level item is defined as the number of reused internal lower level items divided by the total number of lower level items in the higher level item. External reuse level, total reuse level, reuse frequency, and a complexity weighting are also defined.

Reuse Fraction	Agresti, Evanco [Agresti and Evanco 92]	The variable FNEMC is defined as the fraction of new or extensively modified software units. FNEMC is the number of new components plus the number of extensively modified components divided by the total number of components. FNEMC is equal to one minus the "reuse fraction."
Object-Oriented Metrics and Models	Bieman, Karunanithi [Bieman 92] [Bieman and Karunanithi 93]	There are three perspectives from which to view reuse in the object-oriented environment: server perspective, client perspective, and system perspective. Measurable o-o system reuse attributes include percentages of code and classes that are new or derived and specific client/server relationships.
Failure Modes Analysis	[Frakes and Fox 93]	A method for measuring and improving a reuse process based on the ways a reuse process can fail. The failure modes are: No Attempt to Reuse, Part Doesn't Exist, Part Isn't Available, Part Isn't Found, Part Isn't Understood, Part Isn't Valid, Part Can't Be Integrated
Reusability Assessment		
Measuring Reusability for Ada	Basili, Rombach, Bailey, Delis [Basili et al 90]	Two studies address reuse for Ada systems. The first defines a means of measuring data bindings to characterize and identify reusable components. The second defines an abstract measurement of reusability of software components by identifying reusable software and measuring the distance from that ideal.
Reuse Predictions for Lifecycle Objects	Frakes, Fox [Frakes and Fox 93]	Models allow the prediction of reuse levels for one lifecycle object based on reuse levels for other lifecycle objects.
Reuse Library Metrics		
Indexing Costs, Searching Effectiveness and Efficiency	[Frakes and Gandel 90] [Frakes and Pole 94]	Indexing costs include the cost of creating, maintaining, and updating the asset classification scheme. Searching effectiveness measures include recall, precision, overlap, understanding. Searching efficiency measures include memory usage, indexing overhead, retrieval time.
Asset Quality Metrics	[Frakes and Nejmeh, 87]	Indicators of the quality of the assets in the library include: time in use, successful reuses, reuse reviews, complexity, inspection, testing
Reuse Library Usage	[Lillie 95]	Indicators of library usage include: time on-line (system availability), account demographics, type of library function performed : (searches, browses, extracts) asset distribution: number of subscriber accounts, available assets by type: number of extractions by collection: number of assets extracted by collection, number of assets extracted by evaluation level, listing of assets by domain, request for services by: Telnet logins, modem logins, FTP, world wide web

References

[Agresti and Evanco 92] Agresti, W. and Evanco, W. "Projecting software defects in analyzing Ada designs." IEEE Transactions on Software Engineering 18 (11 1992): 988-997.

[Barnes 88] Barnes, B. el al. "A Framework and Economic Foundation for Software Reuse." In IEEE Tutorial: Software Reuse - Emerging Technology, ed. W. Tracz. Washington, D.C.: IEEE Computer Society Press, 1988.

[Barnes and Bollinger 91] Barnes, B. and Bollinger, T. "Making software reuse cost effective." IEEE Software (1 1991): 13-24.

[Basili et al 90] Basili, V. R., Rombach, H.D., Bailey, J., and Delis, A. "Ada reusability and measurement." Computer Science Technical Report Series, University of Maryland. May (1990).

[Bieman 92] Bieman, J. "Deriving measures of software reuse in object oriented systems." In BCS-FACS Workshop on Formal Aspects of Measurement in Springer-Verlag, 1992.

[Bieman and Karunanithi 93] Bieman, J. and Karunanithi, S. "Candidate reuse metrics for object oriented and Ada software." In IEEE-CS 1st International Software Metrics Symposium, 1993.

[Biggerstaff 92] Biggerstaff, Ted. "An Assessment and Analysis of Software Reuse." In *Advances in Computers*, ed. Marshall Yovits. 34. New York: Academic Press, 1992.

[Booch 87] Booch, G. Software components with Ada. Menlo Park, CA: Benjamin / Cummings, 1987.

[Browne et al 90] Browne, J., Lee, T., and Werth, J. "Experimental evaluation of a reusability-oriented parallel programming environment." IEEE Transactions on Software Engineering 16 (2 1990): 111-120.

[Card et al 82] Card, D., McGarry, F., Page, G., et. al. "The software engineering laboratory." NASA/GSFC (2 1982).

[Card et al 86] Card, D., Church, V., and Agresti, W. "An empirical study of software design practices." IEEE Transactions on Software Engineering 12 (2 1986): 264-270.

[Chen and Lee 93] Chen, D. and Lee, P. "On the study of software reuse: using reusable C++ components." Journal of Systems and Software 20 (1 1993): 19-36.

[Chidamber and Kemerer 94] Chidamber, S. and Kemerer, C. "A metrics suite for object oriented design." IEEE Transactions on Software Engineering 20 (6 1994): 476-493.

[Davis 93] Davis, T. "The reuse capability model: a basis for improving an organization's reuse capability." In 2nd International Workshop on Software Reusability in Herndon, VA, 1993.

[Dunn and Knight 91] Dunn, M. F., & Knight, J. C. (1991). Software Reuse in an Industrial Setting: A Case Study. In 13th International Conference on Software Engineering, (pp. 329-338). Austin, TX: IEEE CS Press.

[Favaro 91] Favaro, J. "What price reusability? A case study." Ada Letters (Spring 1991): 115-124.

[Fenton 91] Fenton, N. Software Metrics A Rigorous Approach. Chapman & Hall, London, 1991.

Frakes, W.B. and Nejmeh, B.A., "Software Reuse Through Information Retrieval", Proceedings of the Twentieth Annual Hawaii International Conference on Systems Sciences, Kona, pp. 530-535, January 1987

[Frakes and Gandel 90] Frakes, W. and Gandel, P. "Representing reusable software." Information and Software Technology 32 (10 1990): 653-664.

[Frakes 90] Frakes, W. "An empirical framework for software reuse research." In Third Workshop on Tools and Methods for Reuse in Syracuse, NY, 1990.

[Frakes 91] Frakes, W. (moderator). "Software reuse: is it delivering?" In 13th International Conference on Software Engineering in Los Alamitos, CA, IEEE Computer Society Press, 1991.

[Frakes 93] Frakes, W. "Software reuse as industrial experiment." American Programmer (September 1993): 27-33.

[Frakes and Fox 94] Frakes, William and Christopher Fox. Quality Improvement Using a Software Reuse Failure Modes Model. Software Engineering Guild, Technical Report 1994.

[Frakes and Isoda 94] Frakes, William and Sadahiro Isoda. "Success Factors of Systematic Reuse." IEEE Software 11 (5 1994): 14-19.

[Frakes and Pole 94] Frakes, William and Thomas Pole. "An Empirical Study of Representation Methods for Reusable Software Components." IEEE Transactions on Software Engineering (1994):

[Frakes and Terry 94] Frakes, William and Carol Terry. "Reuse Level Metrics." In Third International Conference on Software Reuse in Rio de Janeiro, edited by W. Frakes, IEEE CS Press, 139-148, 1994.

[Frakes and Fox 95] Frakes, W. and C. Fox., "Modeling Reuse Across the Software Lifecycle", Journal of Systems and Software (to appear).

[Gaffney and Durek 89] Gaffney, J.E. and Durek, T.A. "Software reuse - key to enhanced productivity: some quantitative models." Information and Software Technology 31 (5 1989): 258-267.

[Humphrey 89] Humphrey, W. (1989). Managing the Software Process. Reading, MA: Addison-Wesley.

[Hutchins and Basili 85]. Hutchens, D.H. and V. Basili, "System Structure Analysis: Clustering with Data Bindings", IEEE Transactions on Software Engineering 11(8): 749-757.

[Jones 93] Jones, C. "Software return on investment preliminary analysis." Software Productivity Research, Inc., 1993.

[Koltun and Hudson 91] Koltun, P. and Hudson, A. "A reuse maturity model." In 4th Annual Workshop on Software Reuse in Herndon, VA, 1991.

[Lillie 95] Personal Communication

[Margono and Rhoads 93] Margono, T. and Rhoads, T. "Software reuse economics: cost-benefit analysis on a large-scale Ada project." In International Conference on Software Engineering ACM, 1993.

[McGregor and Sykes 92] McGregor, J. and Sykes, D. Object-Oriented Software Development: Engineering Software for Reuse. New York: Van Nostrand Reinhold, 1992.

[Ogush 92] Ogush, M. "A Software Reuse Lexicon", Crosstalk, December 1992.

[Poulin et al 93] Poulin, J.S., Caruso, J.M., and Hancock, D.R. "The business case for software reuse" IBM Systems Journal 32 (4 1993): 567-594.

[Prieto-Diaz 93] Prieto-Diaz, R. "Status report: software reusability."
IEEE Software, May 1993, p 61-66.

[Selby 89] Selby, Richard W., "Quantitative Studies of Software Reuse," in "Software Reusability, Volume II," Ted J. Biggerstaff and Alan J. Perlis (eds). Addison-Wesley, Reading, MA, 1989.

[Terry 93] Terry, C. "Analysis and Implementation of Software Reuse Measurement." Master's Project and Report, Virginia Polytechnic Institute and State University, 1993.

Appendix 1: Definitions of Types of Reuse

The terms in the table describe various reuse issues. Some terms in the table overlap in meaning. For example, the terms *public* and *external* both describe the part of a product which was constructed externally; *private* and *internal* describe the part of a product which was not constructed externally but was developed and reused within a single product. The terms *verbatim* and *black-box* both describe reuse without modification; *leveraged* and *white-box* describe reuse with modification. The final four terms in the table describe levels of reuse which can occur in the object-oriented paradigm.

Type of Reuse	Description
abstract-level reuse	Abstract-level reuse is the use of high-level abstractions within an object-oriented inheritance structure as the foundation for new ideas or additional classification schemes [McGregor and Sykes 92].
ad-hoc	Ad-hoc reuse refers to the selection of components which are not designed for reuse from general libraries; reuse is conducted by the individual in an informal manner [Prieto-Diaz 93].
adaptive	Adaptive reuse is a reuse strategy which uses large software structures as invariants and restricts variability to low-level, isolated locations. An example is changing arguments to parameterized modules [Barnes and Bollinger 91].
black-box	Black-box reuse is the reuse of software components without any modification [Prieto-Diaz 93]. See *verbatim*..
Carry-Over Reuse	Software that occurs when one version of a software component is taken to be used as is in a subsequent version of the same system. [Ogush 92]
compositional	Compositional reuse is a reuse strategy which uses small parts as invariants; variant functionality links those parts together. Programming in a high level language is an example [Barnes and Bollinger 91]. Compositional reuse is the use of existing components as building blocks for new systems. The Unix shell is an example [Prieto-Diaz 93].
customization reuse	Customization reuse is the use of object-oriented inheritance to support incremental development. A new application may inherit information from an existing class, overriding certain methods and adding new behaviors [McGregor and Sykes 92].
direct	Direct reuse is reuse without going through an intermediate entity[Bieman and Karunanithi 93].
external	External reuse level [Frakes 90] is the number of lower level items from an external repository in a higher level item divided by the total number of lower level items in the higher level item. See *public*.
generative	Generative reuse is reuse at the specification level with application or code generators. Generative reuse offers the "highest potential payoff." The Refine and MetaTool systems are state of the art examples [Prieto-Diaz 93].
generic	Generic reuse is reuse of generic packages, such as templates for packages or subprograms [Bieman and Karunanithi 93].

horizontal scope	Horizontal reuse is reuse of generic parts in different applications. Booch Ada Parts and other subroutine libraries are examples [Prieto-Diaz 93].
In-the-large	Reuse-in-the-large is the use of large, self-contained packages such as spreadsheets and operating systems [Favaro 91].
In-the-small	Reuse-in-the-small is the reuse of components which are dependent upon the environment of the application for full functionality. Favaro asserts that component-oriented reuse is reuse-in-the-small [Favaro 91].
indirect	Indirect reuse is reuse through an intermediate entity. The level of indirection is the number of intermediate entities between the reusing item and the item being reused [Bieman and Karunanithi 93].
instance-level reuse	Instance-level reuse is the most common form of reuse in an object-oriented environment. It is defined as simply creating an instance of an existing class [McGregor and Sykes 92].
internal	Internal reuse level [Frakes 90] is the number of lower level items not from an external repository which are used more than once divided by the total number of lower level items not from an external repository. See *private*.
leveraged	Bieman and Karunanithi define leveraged reuse as reuse with modifications [Bieman and Karunanithi 93].
private	Fenton [Fenton 91] defines private reuse as "the extent to which modules within a product are reused within the same product." See *internal* .
public	Fenton [Fenton 91] defines public reuse as "the proportion of a product which was constructed externally." See *external* .
source-code reuse	Source code reuse is the low-level modification of an existing object-oriented class to change its performance characteristics.
systematic (planned mode)	Planned reuse is the systematic and formal practice of reuse as found in software factories [Prieto-Diaz 93].
verbatim	Bieman and Karunanithi define verbatim reuse as reuse of some item without modifications [Bieman and Karunanithi 93]. See *black-box*.
vertical scope	Vertical reuse is reuse within the same application or domain. An example is domain analysis or domain modeling [Prieto-Diaz 93].
white-box	White-box reuse is the reuse of components by modification and adaptation [Prieto-Diaz 93]. See *leveraged*.