Measuring Software Reusability

Jeffrey S. Poulin

Loral Federal Systems–Owego

Abstract

This paper examines various approaches to measuring software reusability. Knowing what makes software "reusable" can help us learn how to build new reusable components and help us to identify potentially useful modules in existing programs. The paper begins by establishing a taxonomy of approaches to reusability metrics based on their empirical or qualitative orientation. The paper then examines the disciplines, theories, and techniques used by numerous existing reusability measurement methods as they relate to the taxonomy. Recognizing that most of these methods focus exclusively on internal characteristics of components and ignore environmental factors, the paper challenges reusability researchers to incorporate domain attributes into their metrics. In fact, the application domain represents a critically important factor in whether or not we find a component reusable. The research, framework, and conclusions should provide a useful reference for persons interested in ways to determine the reusability of software.[1]

Keywords: Software Reuse, Reusability Metrics.

1.0 Overview

1.1 Motivation

Many believe software reuse provides the key to enormous savings and benefits in software development; the U.S. Department of Defense alone could save $300 million annually by increasing its level of reuse by as little as 1% [1]. However, we have yet to identify a reliable way to quantify what we mean by "reusable" software. Such a measure may help us not only to learn how to build reusable components but also to identify reusable components among the wealth of existing programs.

Existing programs contain years of knowledge and experience gained from working in the application domain and meeting the organization's software needs. If we could extract this information efficiently, we could gain a valuable resource upon which to build future applications. Unfortunately, working with existing software poses several significant problems due to its inconsistent quality, style, documentation, and design. We need an economical way to find domain knowledge and useful artifacts within these programs.

1.2 Measuring reusability

This paper examines metrics used to determine the *reusability* of software components. In light of the recent emphasis on software reuse, numerous research efforts have attempted to quantify our ability to use a component in new contexts. Many of the known empirical methods use a version of complexity metrics to measure reusability. However, many other objective and subjective criteria contribute to the question:

What makes software reusable?

One possible measure of a component's reusability comes from its success; how many other application modules access this common code? Other measures come from static code metrics generated automatically by a variety of commercial tools. Additional qualitative measures include: adherence to formatting standards and style guidelines, completeness of testing, and existence of supporting information such as integration instructions and design documentation. This paper examines these various approaches to measuring reusability; note that it distinguishes reusability metrics from those that define reuse [34], the level of reuse in an organization [33], or reuse Return on Investment (ROI) models [11], [35]. Where these reports focus on the level of reuse and its benefits, reusability metrics seek ways to identify the software that will bring those savings.

---

[1] Proceedings of the *Third International Conference on Software Reuse*, Rio de Janeiro, Brazil, 1-4 November 1994.

2.0 A Taxonomy of Reusability Metrics

Approaches to measuring reusability fall into two basic methods: *empirical* and *qualitative.* Because empirical methods depend on objective data, they have the desirable characteristic that a tool or analyst can usually calculate them automatically and cheaply. The qualitative methods generally rely on a subjective value attached to how well the software adheres to some guidelines or principles. Although this allows us to attach a value to an abstract concept, collecting qualitative data often requires substantial manual effort.

Within each method, the metrics tend to focus in one of two areas. In the first, the metrics address attributes unique to the individual module, such as the number of source statements it contains, its complexity, or the tests it successfully passed. In the second approach, the metrics take into account factors external to the module, such as the presence and/or quality of supporting documentation. When we refer to a module and all of these supporting pieces of information, we call it a *component.* In summary, these methods take the form shown in the following taxonomy:

Taxonomy of reusability metrics

- **Empirical methods**
  - Module oriented
    - Complexity based
    - Size based
    - Reliability based
  - Component oriented
- **Qualitative methods**
  - Module oriented
    - Style guidelines
  - Component oriented
    - Certification guidelines
    - Quality guidelines

2.1 Related work

We can relate reusability to *portability* since both involve the use of a component in a new context [31]. A 1990 U.S. Department of Defense report concluded that the ultimate measure of portability comes from the number of source lines of code that a programmer must change to get a module to execute in a different environment. This study recommended developing a mathematical function to convert the amount of changed code to a portability value. However, the study continued to state that it could not make conclusions on reusability factors.

Software *complexity* metrics reveal internal characteristics of a module, collection of modules, or object-oriented programs [9]. Studies indicate that complex modules cost the most to develop and have the highest rates of failure. McCabe and Halstead developed the two most widely known complexity metrics:

- The McCabe Cyclomatic Complexity metric links the number of logical branches (decisions) in a module to the difficulty of programming [27]. McCabe melds a graph theory approach with software engineering: if you represent the logic structure of a module using a flowchart (a graph) and count the regions of the graph caused by program flow statements (*do-while*, *if-then-else*), the number of regions in the graph corresponds to the complexity of the program. If the number of regions, $V(G)$, exceeds 10, the module may have too many changes of control.
- Halstead's Software Science metrics link studies on human cognitive ability to software complexity [15]. Halstead's approach parses the program or problem statement into tokens and classifies the tokens into operators (verbs, functions, procedures) and operands (nouns, variables, files). Equations based on these tokens give program complexity in terms of a variety of indicators including estimated effort, program volume, and size.

Not surprisingly, basic software engineering principles address many of the aspects of software that might make software reusable. This particularly applies to those qualities that make software *maintainable.* One study based on this premise constructs a taxonomy of 92 attributes affecting the ease of maintaining software [46]. Among other findings, the study concludes that maintenance effort correlates highly with the Halstead complexity metrics, a finding corroborated by reusability researchers.

Program comprehension relates closely to program complexity [47]. Criteria and conditions for program comprehension show that we can translate theoretic numerical measures from software complexity back to empirical conditions. This means that we can describe program comprehension with empirical axioms. From the software reuse perspective, we could use these techniques to understand software for reuse and to identify potentially reusable components. In one example, a prototype tool uses candidate criterion to identify abstract

data types in existing program code [7]. The tool applies the criterion in an experiment that analyses five different programs for the purpose of (1) reverse engineering/re-engineering and (2) to identify and extract reusable components from the programs.

Another factor affecting whether a programmer will choose to use an existing component in a new situation depends on how quickly the programmer can assimilate what the component does and how to use it. *Program understanding* methods address this problem. These methods attempt to present the important information about a component to the user in a way the user can quickly assess [28]. For example, recognizing that expert programmers organize the important information about a component into mental templates, Lin and Clancy developed a visual template containing this same information. Their study shows that by using a standard layout, a potential reuser can quickly scan the important aspects of a component, such as text descriptions, pseudocode, illustrations, and implementation information [24]. Understanding how good reusable software works not only helps the programmer learn how to write good reusable software, it increases the chances the programmer will use more of what already exists.

The discussion of what makes software reusable has taken place for a long time. In 1984 Matsumoto stressed qualities such as generality, definiteness (the degree of clarity or understandability), transferability (portability), and retrievability as the major characteristics leading to the reusability of a component [25]. However, the search for quantitative measures remains elusive. Next, we will discuss some of the reasons why.

2.2 Issues

One reason why we find it so hard to develop reusability metrics comes from the fact that no one completely understands “design for reuse” issues [4]. Given that humans often do not agree on what makes a component reusable, obtaining an equation that quantifies the concept offers a significant challenge. To put it simply, we need to define reusability before we can quantify it.

To illustrate this point, Woodfield, Embley, and Scott conducted an experiment where 51 developers had to assess the reusability of an Abstract Data Type (ADT) in 21 different situations [45]. They found developers untrained in reuse did poorly; the developers based their decisions on unimportant factors such as size of the ADT and ignored important factors such as the effort needed to modify the ADT. As a result, the study recommends developing tools and education that can help developers assess components for reuse and suggests a reusability metric based on the effort needed to modify a component as reflected by the number or percent of operations to add or modify.

Not only must a reusability metric involve a wide variety of input parameters, it must define the interrelationships among the parameters and their relative importance. The parameters must exhibit certain qualities such as statistical independence [13]. Well-formed metrics should not have any correlation between the elements that make up the metric [23], an issue Selby addresses as part of his statistical study [40].

Once we identify a potentially reliable metric, we must next look at causality. In other words, if we find smaller modules get reused more often, does this mean the small size made the module more reusable, and we should build all reusable modules small? Metrics must carefully separate the factors leading to the findings.

Finally, some metric values may give conflicting reuse information. For example, do we desire a low value or a high value for module complexity? Complex modules may indicate potentially trouble spots and warrant additional testing, re-design, or further decomposition. However, some algorithms may have high complexity values independent of design. Does the reuser stay away from these modules or does the reuser take advantage of the greater payoff that comes from not having to re-develop complex code?

3.0 Empirical Methods

The following methods primarily use objective, quantifiable attributes of software as the basis for a reusability metric. Most use module-oriented attributes, but the methods to interpret the attributes vary greatly.

3.1 Prieto-Diaz and Freeman

In their landmark paper on faceted classification, Prieto-Diaz and Freeman identify five program attributes and associated metrics for evaluating reusability [37]. Their process model encourages white-box reuse and consists of finding candidate reusable modules, evaluating each, deciding which module the programmer can modify the easiest, then adapting the module. In this

model they identify four module-oriented metrics and a fifth metric used to modify the first four. The following list shows the five metrics and gives a description of each:

1. *Program size.* Reuse depends on a small module size, as indicated by lines of source code.
2. *Program structure.* Reuse depends on a simple program structure as indicated by fewer links to other modules (low coupling) and low cyclomatic complexity.
3. *Program documentation.* Reuse depends on excellent documentation as indicated by a subjective overall rating on a scale of 1 to 10.
4. *Programming language.* Reuse depends on programming language to the extent that it helps to reuse a module written in the same programming language. If a reusable module in the same language does not exist, the degree of similarity between the target language and the one used in the module affects the difficulty of modifying the module to meet the new requirement.
5. *Reuse experience.* The experience of the reuser in the programming language and in the application domain affects the previous metrics because every programmer views a module from their own perspective. For example, programmers will have different views of what makes a "small" module, depending on their background. This fifth metric serves to modify the values of the other metrics.

3.2 Selby

To derive measures of reusability, we must look at instances where reuse succeeded and try to determine *why*. Selby provides a statistical study of reusability characteristics of software using data from a NASA software environment [40]. NASA used the production environment to develop ground support software in FORTRAN for controlling unmanned spacecraft. The study provides statistical evidence based on non-parametric analysis-of-variance on the contributions of a wide range of code characteristics. The study validated most of the findings listed below at the .05 level of confidence, showing that most modules reused without modification:

- Have a smaller size, generally less than 140 source statements.
- Have simple interfaces.
- Have few calls to other modules (low coupling).
- Have more calls to low-level system and utility functions.
- Have fewer input-output parameters.
- Have less human interaction (user interface).
- Have good documentation, as shown by the comment-to-source statement ratio.
- Have experienced few design changes during implementation.
- Took less effort to design and build.
- Have more assignment statements than logic statements per source statement.
- Do not necessarily have low code complexity.
- Do not depend on project size.

3.3 Chen and Lee

Although Selby's evidence did not find a statistically significant correlation between module complexity and reusability, other studies show such a link. In one example, Chen and Lee developed about 130 reusable C++ components and used these components in a controlled experiment to relate the level of reuse in a program to software productivity and quality [8]. In contrast to Selby, who worked with professional programmers, Chen and Lee's experiment involved 19 students who had to design and implement a small data base system. The software metrics collected included the Halstead size, program volume, program level, estimated difficulty, and effort. They also included McCabe complexity and the Dunsmore live variable and variable span metrics [10]. They found that the lower the values for these complexity metrics, the higher the programmer productivity.

3.4 Caldiera and Basili

Caldiera and Basili [6] state that basic reusability attributes depend on the qualities of correctness, readability, testability, ease of modification, and performance, but they acknowledge we cannot directly measure or predict most of these attributes. Therefore, the paper proposes four candidate measures of reusability based largely on the McCabe and Halstead metrics. This module-oriented approach has an advantage in that tools can automatically calculate all of the four metrics and a range of values for each:

1. *Halstead's program volume.* A module must contain enough function to justify the costs of retrieving and integrating it, but not so much function as to jeopardize quality.
2. *McCabe's cyclomatic complexity.* Like Halstead's volume, the acceptable values for the McCabe metric must balance cost and quality.

3. *Regularity.* Regularity measures the readability and the non-redundancy of a module implementation by comparing the actual versus estimated values of Halstead's two length metrics. A clearly written module will have an actual Halstead length close to its theoretical Halstead length.
4. *Reuse frequency.* Reuse frequency indicates the proven usefulness of a module and comes from the number of static calls to the module.

The paper continues by calculating these four metrics for software in nine example systems, and noting that the four metrics show a high degree of statistical independence.

3.5 REBOOT

The ESPRIT-2 project called REBOOT (Reuse Based on Object-Oriented Techniques) developed a taxonomy of reusability attributes. As part of the taxonomy, they list four reusability factors, a list of criteria for each factor, and a set of metrics for each criteria [22]. Although some of the metrics depend on subjective items such as checklists, an analyst can compute many of the metrics directly from the code, such as complexity, fan-in/out, and the comment-to-source-code ratio. The analyst combines the individual metric values into an overall value for reusability. The following list defines the four reusability factors; Table 1 gives the criteria and metrics for each factor.

- *Portability.* The ease with which someone can transfer the software from one computer system to another. Criteria include:
  - Modularity
  - Environment independence
- *Flexibility.* The number of choices a programmer has in determining the use of the component; also referred to as "generality." Criteria include:
  - Generality
  - Modularity
- *Understandability.* The ease with which a programmer can understand the component. Criteria include:
  - Code complexity
  - Self descriptiveness
  - Documentation quality
  - Module complexity
- *Confidence.* The subjective probability that a component will perform without failure over a specified time in a new environment. Criteria include:
  - Module complexity
  - Observed reliability
  - Error tolerance

Table 1. The REBOOT reusability metrics for the four reusability factors

Criteria	**Metric**
Generality	Generality checklist
Modularity	Code / number of methods
Environment independence	Machine-dependent code / executable code System-dependent code / executable code
Code complexity	Cyclomatic complexity
Self descriptiveness	Comments / source code Self-descriptiveness checklist
Documentation quality	Documentation / source code Documentation checklist
Module complexity	Fan-in, fan-out Cyclomatic complexity
Reliability	Total number of tests Number of observed errors
Error tolerance	Error tolerance checklist

3.6 Hislop

Hislop discusses three approaches to evaluating software; function, form, and similarity [19]. Evaluating software on *function* helps to select components based on what the component does. In fact, many part collections use a hierarchical organization based on function. The second approach, *form*, characterizes the software based on observable characteristics such as size or structure. This approach lends itself well to code analysis tools that report on numbers of source statements or code complexity. The third approach, *similarity*, compares modules and groups modules based on shared attributes.

Hislop's work uses ideas drawn from plagiarism detection, where instructors seek to identify cases of students "reusing" each others programs. However, this type of analysis also helps in the study of reusability in a number of ways. First, a tool can identify potentially reusable modules in existing software by finding modules with a similarity metric close to those of successfully reused modules. Second, identifying groups of similar modules can help in domain analysis by showing

opportunities for reuse. Third, the method can help automate reuse metrics by locating instances of informal reuse with or without modification.

Hislop's prototype tool, *SoftKin*, consists of a data collector and a data analyzer. The collector parses the software and calculates measures of form for each module. The analyzer computes the similarity measures based on a variety of form metrics such as McCabe complexity and structure profile metrics.

3.7 Boetticher and Eichmann

Recognizing the difficulty of defining what humans seem to accept as an intuitive notion of reusability, Boetticher and Eichmann take an alternative approach to reusability metrics. They base their work on the training of neural networks to mimic a set of human evaluators [4], [5]. They conducted experiments that varied several neural network configurations and each network's ability to match the output provided by a group of four humans. Using an Ada repository, the study used commercial metric tools to generate over 250 code parameters with the goal of determining the best possible association between the parameters and the human assessments.

The input parameter selection contained code parameters representing module complexity, adaptability, and coupling. The experiment proceeded in three phases, using input parameters significant in *black box* reuse (reuse without modification), *white box* reuse (reuse with modification), and *grey box* reuse (a combination of black box and white box reuse). Black box parameters included source statements, physical size, file size, and number of inputs/outputs. White box parameters included Halstead volume, cyclomatic complexity, coupling, and size. Grey box metrics combined the black box and white box inputs.

The experiment strived for a best fit through sensitivity analysis on parameters selected for the training vectors, the neural network configuration, and extent of network training. Although the black box and white box results showed little correlation to the expected outputs (.18 and .57, respectively), the grey box results correlated very well (.86) with the expert ratings. The experiment concluded that neural networks could serve as an economical, automatic tool to generate reusability rankings of software.

3.8 Torres and Samadzadeh

Torres and Samadzadeh conducted a study to determine if information theory metrics and reusability metrics correlate [42]. Information theory measures information content as the level of entropy in a system (entropy reflects the degree of uncertainty or unpredictability in a system). This study examined the effects of two information theory metrics, *entropy loading* and *control structure entropy*, on software reusability.

Entropy loading reflects the degree of inter-process communication. Since the amount of communication required between parts of any system drives entropy up, information theory seeks ways to reduce entropy by designing systems with minimal communication between sub-systems. Applying this concept to software, the researchers measure the amount of communication required between modules and assign a value for entropy loading to each module. Entropy loading corresponds to the software engineering concepts of coupling and cohesion; programs that possess small values for entropy loading should also possess properties consistent with good program structure and reusability.

Control structure entropy seeks to measure the complexity of a modules's logic structure, as reflected by the number of *if-then* statements in the module. Like cyclomatic complexity, control structure entropy provides a value for module complexity.

An experiment that calculated the two information theory metrics on six programs (three in Ada, three in C) found that high entropy loading (coupling) had an negative effect on reuse, while low control structure entropy (complexity) had a positive effect. The study concluded that there exists a possible relationship between information theory metrics and reusability metrics. Consequently, these metrics might help select the optimum reuse case among different reuse candidates.

3.9 Mayobre

To help identify reusable workproducts in existing code, Mayobre describes a method called Code Reusability Analysis (CRA) [26]. CRA melds three reusability assessment methods and an economical estimation model to identify reusable components, resulting in a combination of automatic analysis and expert analysis to determine both technical reusability and economic value. CRA uses the Caldiera and Basili method as one of the

three methods. The second method, called Domain Experience Based Component Identification Process (DEBCIP) depends on an extensive domain analysis of the problem area and uses a decision graph to help domain experts identify reusable components. The primary output of the DEBCIP provides an estimate of the expected number of reuse instances for the component.

The third method, called Variant Analysis Based Component Identification Process (VABCIP), also uses domain knowledge but to a lesser degree. It uses cyclomatic complexity metrics to estimate the specification distance between the existing module and the required module giving an estimate of the effort needed to modify the component. The last step of the reusability analysis consists of estimating the Component Return On Investment, a process consisting of comparing the estimated costs of reuse with the expected benefits.

A test of this method with 7-8 engineers and 40k source statements in the data communication domain showed a very high correlation of about 88% between the Caldiera and Basili metrics and expert analysis. The full CRA including VABCIP had better results, but took up to four weeks to complete and required a domain expert, a domain analyst, and a software engineer familiar with software metrics.

3.10 NATO

The NATO Standard for Software Reuse Procedures recommends tracking four metrics as indicators of software quality and reusability [30]:

- *Number of inspections.* The number of times someone has considered the module for reuse.
- *Number of reuses.* The number of times someone actually has reused the module.
- *Complexity.* The complexity of the code, normally based on the McCabe complexity metric.
- *Number of problem reports.* The number of outstanding defects in the module.

The standard suggests these metrics present a rough estimate of the reusability of a component and can help eliminate unsuitable candidates. For example, potential reusers should look for components with a high number of prior reuses, but remain skeptical of components with a high number of inspections and few actual reuses. The standard recommends reusing components with low complexity values and fewer known defects.

3.11 US Army Reuse Center

The Army Reuse Center (ARC) inspects all software submitted to the Defense Software Repository System (DSRS) [38]. As part of that inspection, each component undergoes a series of reusability evaluations [32]. The preliminary screening consists of coarse measures of module size in source statements, the estimated effort to reuse the component without modification, the estimated effort needed to develop a new component rather than reuse one, the estimated yearly maintenance effort, and the number of expected reuses of the component.

Following this screening, the ARC conducts several other assessments and tests. They calculate an initial and a final reusability metric using a commercially available Ada source code analysis tool. The initial analysis uses 31 metrics supplied by the tool; the final analysis uses a total of 150 metrics. Table 2 lists a subset of the 31 metrics used in the initial analysis by metric category, and gives the metric threshold values required by the ARC.

Table 2. Partial list of ARC reusability metrics

Category	**Metric**	**Threshold Value**
Anomaly Management	Normal_Loops	95%
	Constrained_Subtype	80%
	Contraint_Error	80%
	Constrained_Numerics	90%
	Constraint_Error	0%
	Program_Error	0%
	Storage_Error	0%
	Numeric_Error	0%
	User_Exception_Raised	100%
Independence	No_Missed_Closed	0%
	Fixed_Clause	100%
	No_Pragma_Pack	0%
	No_Machine_Code_Stmt	100%
	No_Impl_Dep_Pragmas	0%
	No_Impl_Dep_Attrs	0%

4.0 Qualitative Methods

Because finding and agreeing upon a purely objective reusability metric often proves difficult, many organizations provide subjective guidance on identifying or building reusable software [12], [16], [17], [20], [29], [38]. These guidelines help ameliorate the problem of not knowing *exactly* how to define reusability with an intuitive description of what a reusable component ought to look like. The guidelines range in content from

general discussions about designing for reuse to very detailed rules and specific design points. Usually module-oriented, they cover points such as formatting and style requirements. Although guidelines primarily belong to the general areas of *designing for reuse* or *building for reuse*, organizations may develop reusability metrics based on how well a component meets the published standard.

Rather than expand on the many available guidelines, this section presents some general "reusability" attributes and two examples showing how these attributes translate to reusability principles. Finally, this section ends with an example of a component-oriented approach.

4.1 General reusability attributes

Most sets of reusability guidelines have a lot in common: The value they add to their organizations comes from code specific rules and from level of detail. In general, the guidelines reflect the same software characteristics as those promoted by good software engineering principles [36]. This emphasizes the fact that reuse requires a focus on the basic problem of good software design and development. Table 3 gives a high-level summary of these software engineering concepts as seen by the Software Technology for Adaptable, Reliable Systems (STARS) Program [41].

Table 3. General attributes of reusable software

Attribute	**Description**
Ease of understanding	The component has thorough documentation, including self-documenting code and in-line comments.
Functional completeness	The component has all the required operations for the current requirement and any reasonable future requirements.
Reliability	The component consistently performs the advertised function without error and passes repeated tests across various hardware and operating systems.
Good error and exception handling	The component isolates, documents, and handles errors consistently. It also provides a variety of options for error response.
Information hiding	The component hides implementation details from the user, for example, internal variables and their representation. It also clearly defines the interfaces to other operations and data.
High cohesion and low coupling	The component does a specific, isolated function with minimal external dependencies.
Portability	The component does not depend on unique hardware or operating system services.

The application of these abstract concepts will in large part determine their success. In situations where one team provides shared software to the rest of a project, the products this team builds must have characteristics that both make them general enough for multiple uses and have enough supporting information to make them easily useful. If the team does not meet the needs of their customers, the customers will not use the software. The team must make the software general by carefully examining each customer requirement and abstracting the necessary detail.

4.2 The 3 C Model

The "3 C Model" of reusable software components comes from its three constituent design points: [44]:

- *Concept.* What abstraction the component represents.
- *Content.* How the component implements the abstraction.
- *Context.* The environment in which the component operates.

The Concept of a component relates to its specification or interface; it gives a black-box perspective of the component's function. The Content relates to the actual

algorithm, or code, that implements the function abstracted by the Concept. The Context refers to those parts of the environment that affect the use of the component; in other words, the Context defines the component's dependencies when used in another application.

The 3C Model seeks to isolate Concept, Content, and Context-specific dependencies from each other during the design and implementation of a module. By building a clear boundary between each, a reuser can modify one without affecting the others. For example, several module implementations may exist to serve the same interface, thereby allowing the reuser to select the best implementation to meet specific constraints or performance criteria. Successful implementation of the 3C Model would allow developers to treat modules like "software integrated circuits" by plugging them into sockets in an application framework.

4.3 University of Maryland

Two example sets of coding guidelines to increase the reusability of Ada modules resulted from a set of reuse related projects at the University of Maryland [2], [3]. These guidelines fall into two categories, those based on data binding and those based on transformation. Data binding measures the strength of data coupling between modules. Transformation formally quantifies the number and types of modifications required to adapt a module into something reusable. By writing modules to reduce data binding dependencies and the number of expected changes required, a programmer can develop more reusable modules. The following guidelines reflect software engineering principles applied to Ada:

Reuse guidelines based on data bindings

- Avoid multiple level nesting in any of the language constructs.
- Minimize use of the "use" clause.
- The interfaces of the subprograms should use the appropriate abstraction for the parameters passed in and out.
- Components should not interact with their outer environment.
- Appropriate use of packaging could greatly accommodate reusability.

Reuse guidelines based on dependencies

- Avoid direct access into record components except in the same declarative region as the record type declaration.
- Minimize non-local access to array components.
- Keep direct access to data structures local to their declarations.
- Avoid the use of literal values except as constant value assignments.
- Avoid mingling resources with application specific contexts.
- Keep interfaces abstract.

Providing and enforcing a *component's* compliance to completeness or quality standards also provides a way to enhance reusability. For a developer to efficiently use a software module, the developer must have access to other information such as design documents, integration instructions, test cases, and legal information [39]. Table 4 contains a listing of the kinds of supporting information IBM provides to potential reusers. A component receives one of three quality ratings based, in part, on the completeness and quality of this information supporting the reusable module.

Table 4. Information helpful when reusing software

Attribute	**Description**
Abstract	Provides a clear, concise description of the component.
Change history	Describes changes to the code, who made them, the date of the changes, and why.
Dependencies	Describes prerequisite software and other software the component uses.
Design	Describes the internal design of the code and major design decisions.
Interfaces	Describes all inputs, outputs, operations, exceptions, and any other side effects visible to the reuser.
Legal	Provides a summary of legal information and restrictions, such as license and copyright information.
Performance	Describes the time requirements, space requirements, and any performance considerations of the algorithm.
Restrictions	Lists any situations that limit the usability of the component, such as nonstandard compiler options and known side effects.
Sample	Provides a usage scenario showing how the component applies to a specific problem.
Test	Contains information about the test history, procedures, results, and test cases.
Usage	Provides helpful information on how to integrate the component.

5.0 A Common Model for Reusability

Table 5 summarizes the reusability metrics discussed in this paper. Most empirical methods take a module-oriented approach since modules provide many easily measurable attributes. Of these attributes, the following appear common to most of the empirical methods:

- Reusable modules have low module complexity.
- Reusable modules have good documentation (a high number of non-blank comment lines).
- Reusable modules have few external dependencies (low fan-in and fan-out, few input and output parameters).
- Reusable modules have proven reliability (thorough testing and low error rates).

The qualitative methods vary in their application of software engineering principles, code-specific implementation issues, and level of detail. When they assign a reusability value to a module, they typically base the value on a subjective assessment of how well a module complies with a set of guidelines. Component-oriented approaches not only specify code standards but also list required supporting information that a developer must have to effectively reuse a component.

Table 5. Reusability metric summary

Method	**E**	**Q**	**M**	**C**
Prieto-Diaz and Freeman	√		√	
Selby	√		√	
Chen and Lee	√		√	
Caldiera and Basili	√		√	
REBOOT	√		√	
Hislop	√		√	
Boetticher and Eichmann	√		√	
Torres and Samadzadeh	√		√	
Mayobre	√		√	
NATO (2 approaches)	√	√	√	
Army (2 approaches)	√	√	√	
STARS		√	√	
U. Maryland		√	√	
IBM		√		√

Note:

E = Empirical
Q = Qualitative
M = Module oriented
C = Component oriented

6.0 Domain Considerations

Tracz observed that for programmers to reuse software, they must first find it *useful* [43]. In some sense, researchers have fully explored most traditional methods of measuring reusability: complexity, module size, interface characteristics, etc. However, although many recognize the importance of the problem *domain* to reuse, few have linked this affect on the ultimate "usefulness" of a component.

The usefulness of a component depends as much on the framework in which it fits as it does on internal characteristics of the component. In other words, the real benefits of reuse occur following:

1. an analysis of the problem domain
2. capturing the domain architecture
3. building or including components into the architecture.

Although low coupling, high cohesiveness, modularity, etc. give general indications as to the ease of use of a component, they cannot, by themselves, provide a generic measure of reusability. Reusability comes from adhering to a domain architecture and to the above software engineering principles. With the exception of [26], the above methods do not include any domain characteristics. The input parameters come from observable or readily obtainable data about the software component. If the reusability of a component depends on *context* then reusability metrics must also include domain and environment characteristics. Research issues must include ways to quantify a domain's size, stability, and maturity [21].

6.1 Applying Metric Theory

The fact remains that most published work in software metrics, including reusability metrics, does not follow the important rules of measurement theory. We can informally say that any number we assign to a component's reusability must preserve the intuitive understanding and observations we make about that component. This understanding leads us to put a value on the metric. More formally, if we observe component *A* as less reusable than component *B*, then our reusability measurement, *M*, must preserve $M(A) < M(B)$.

The first problem in metric theory comes from our trying to put an empirical value on a poorly understood attribute. To make matters worse, relations we define generally fall apart when we change contexts. We often

find that a component we feel has very high reusability attributes in domain *X* and environment *S* has none of these characteristics in domain *Y* and environment *T*. Software engineers working in these two situations would never reach consensus on the reusability for this individual component. This makes the search for a general reusability metric seem futile for the same reasons Fenton discusses regarding the search for a general software complexity metric [14].

Metric theory tells us that we can look for metrics that assign an empirical value to specific *attributes* or *views* of reusability, but not an overall reusability rating. Furthermore, if research truly reveals that *context* affects our view of a component's reusability as much as the component's internal attributes, then we may some day find a metric *M* that maps the tuple *(internal attributes, domain attributes, environment attributes)* to *M*.

7.0 Conclusion

Although reusability guidelines and module-oriented metrics provide an intuitive feel for the general reusability of a component, we need to work on proving that they actually reflect a component's reuse potential. Until researchers can agree on this issue, we will not develop a uniform metric. Existing techniques show a wide range of ways to address this problem, ranging from empirical to qualitative methods. So far, the results mostly indicate a need to first define a suitable scope for future research. This scope must include the affects of *domain* and *environment* when we talk about measuring the "reusability" of individual components.

8.0 Acknowledgements

This paper expands an earlier work with current research and input from recent discussions on reuse@wunet.wustl.edu. I would also like to thank Will Tracz and Marilyn Gaska of Loral Federal Systems–Owego for their insights during the preparation of this paper. *Author's address:* poulinj@lfs.loral.com or MD 0124, Loral Federal Systems–Owego, New York, 13827.

9.0 Cited References

[1] Anthes, Gary H., "Software Reuse Plans Bring Paybacks," *Computerworld*, Vol. 27, No. 49, pp.73,76.

[2] Bailey, John W. and Victor Basili, "Software Reclamation: Improving Post-Development Reusability," *8th Annual National Conference on Ada Technology*, 1990.

[3] Basili, Victor R., H. Dieter Rombach, John Bailey, and Alex Delis, "Ada Reusability Analysis and Measurement,," *Empirical Foundations of Information and Software Science V*, Atlanta, GA, 19-21 October 1988, pp. 355-368.

[4] Boetticher, G., K. Srinivas, and D. Eichmann, "A Neural Net-based Approach to Software Metrics," *Proceedings of the 5th International Conference on Software Engineering and Knowledge Engineering*, San Francisco, CA, 14-18 June 1993, p. 271-4.

[5] Boetticher, Gary and David Eichmann, "A Neural Network Paradigm for Characterizing Reusable Software," *Proceedings of the 1st Australian Conference on Software Metrics*, 18-19 November 1993.

[6] Caldiera, Gianluigi and Victor R. Basili, "Identifying and Qualifying Reusable Software Components,," *IEEE Software*, Vol. 24, No. 2, February 1991, pp. 61-70.

[7] Canfora, G. Cimitile, A. Munro, M. Tortorella, M., "Experiments in Identifying Reusable Abstract Data Types in Program Code," *Proceedings IEEE Second Workshop on Program Comprehension*, Capri, Italy, 8-9 July 1993, pp. 36-45.

[8] Chen, Deng-Jyi and P.J. Lee, "On the Study of Software Reuse Using Reusable C++ Components," *Journal of Systems Software*, Vol. 20, No. 1, Jan 1993, pp. 19-36.

[9] Chidamber, Shyam R. and Chris F. Kemerer, "Towards a Metrics Suite for Object Oriented Design,," *Proc. OOPSLA 1991*, ACM Press, Oct 1991, pp. 197-211.

[10] Conte, S.D., H.E. Dunsmore, and V.Y. Shen. Software Engineering Metrics and Models. Benjamin Cummings, NY, 1986.

[11] Cruickshank, Robert D. and John E. Gaffney, Jr., "The Economics of Software Reuse," *Software Productivity Consortium*, SPC-92119- CMC, Version 01.00.00, September 1991.

[12] Edwards, Stephan, "An Approach for Constructing Reusable Software components in Ada," *Strategic Defense Organization Pub # Ada233 662*, Washington, D.C., September 1990.

[13] Fenton, Norman E., Software Metrics: A Rigourous Approach. *Chapman & Hall*, London, UK, 1991.

[14] Fenton, Norman E., "Software Measurement: A Necessary Scientific Basis," *IEEE Transactions on Software Engineering*, Vol. SE-20, No. 3, March 1994, pp. 199-206.

[15] Halstead, Maurice H. Elements of Software Science. Elsevier North-Holland, New York, 1977.

[16] Hooper, James W. and Chester, Rowena O., "Software Reuse Guidelines," U.S. Army Institute for Research in Management Information, Communication, and Computer Sciences, ASQB-GI-90-015, April 1990.

[17] Hooper, James W. and Chester, Rowena O.. Software Reuse Guidelines and Methods. Plenum Press, 1991.

[18] Griss, Martin and Will Tracz, eds. "WISR'92: 5th Annual Workshop on Software Reuse Working Group Reports," *ACM SIGSOFT Software Engineering Notes*, Vol. 18, No. 2, April 1993, pp. 74-85.

[19] Hislop, Gregory W., "Using Existing Software in a Software Reuse Initiative," *The Sixth Annual Workshop on Software Reuse (WISR'93)*, 2-4 November 1993, Owego, New York.

[20] Hollingsworth, Joe. Software Component Design-for-Reuse: A Language Independent Discipline Applied to Ada. *Ph.D. Thesis*, Dept. of Computer and Information Science, The Ohio State University, Columbus, OH, 1992.

[21] Isoda, Sadahiro, "Experience Report on Software Reuse Project: Its Structure, Activities, and Statistical Results," *Proceedings of the International Conference on Software Engineering*, Melbourne, Australia, 11-15 May 1992, pp. 320-326.

[22] Karlsson, Even-Andre, Guttorm Sindre, and Tor Stalhane, "Techniques for Making More Reusable Components," *REBOOT Technical Report #41*, 7 June 1992.

[23] Kitchenham, Barbara and Kari Kansala, "Inter-item Correlations among Function Points," *Proceedings of the IEEE Computer Society International Software Metrics Symposium*, Baltimore, MD, 21-22 May 1993, pp. 11-15.

[24] Linn, Marcia C. and Michael J. Clancy, "The Case for Case Studies of Programming Problems," *Communications of the ACM*, Vol. 35, No. 3, March 1992, p. 121.

[25] Matsumoto, Yoshihiro, "Some Experience in Promoting Reusable Software Presentation in Higher Abstraction Levels," *IEEE Transactions on Software Engineering*, Vol. 10, No. 5, September 1984, pp. 502-513.

[26] Mayobre, Guillermo, "Using Code Reusability Analysis to Identify Reusable Components from the Software Related to an Application Domain," *Fourth Annual Workshop on Software Reuse (WISR'91)*, Reston, VA, 18-22 November 1991.

[27] McCabe, T.J., "A Complexity Measure," *IEEE Transactions on Software Engineering*, SE-2, 1976, pp. 308-320.

[28] Musser, David R. and Alexander A. Stepanov, The Ada Generic Library. Springer-Verlag, New York, 1989.

[29] NATO, "Standard for the Development of a Reusable Software Components," *NATO Communications and Information Systems Agency*, 18 August 1991.

[30] NATO, "Standard for Management of a Reusable Software Component Library," *NATO Communications and Information Systems Agency*, 18 August 1991.

[31] Pennell, James P., "An Assessment of Software Portability and Reusability for the WAM Program," *Institute for Defense Analysis*, Alexandria, VA, October 1990.

[32] Piper, Joanne C. and Wanda L. Barner, "The RAPID Center Reusable Components (RSCs) Certification Process," *U.S. Army Information Systems Software Development Center - Washington*, Ft. Belvoir, VA.

[33] Poulin, Jeffrey S. and Joseph M. Caruso, "A Reuse Measurement and Return on Investment Model," *Proceedings of the Second International Workshop on Software Reusability*, Lucca, Italy, 24-26 March 1993, pp. 152-166.

[34] Poulin, Jeffrey S., "Issues in the Development and Application of Reuse Metrics in a Corporate Environment," *Fifth International Conference on Software Engineering and Knowledge Engineering*, San Francisco, CA, 16-18 June 1993, pp. 258-262.

[35] Poulin, Jeffrey S., Debera Hancock and Joseph M. Caruso, “The Business Case for Software Reuse,” *IBM Systems Journal*, Vol. 32, No. 4., 1993, pp. 567-594.

[36] Pressman, R.S. Software Engineering: A Practitioner's Approach. McGraw-Hill, 1992.

[37] Prieto-Diaz, Ruben and Peter Freeman, “Classifying Software for Reusability,” *IEEE Software*, Vol. 4, No. 1, January 1987, pp. 6-16.

[38] RAPID, “RAPID Center Standards for Reusable Software,” *U.S. Army Information Systems Engineering Command*, 3451-4-012/ 6.4, October 1990.

[39] RIG Technical Committee on Asset Exchange Interfaces, “A Basic Interoperability Data Model for Reuse Libraries (BIDM),” *Reuse Interoperability Group (RIG) Proposed Standard RPS-0001*, 1 April 1993.

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

[41] STARS, “Repository Guidelines for the Software Technology for Adaptable, Reliable Systems (STARS) Program,” CDRL Sequence Number 0460, 15 March 1989.

[42] Torres, William R. and Mansur H. Samadzadeh, “Software Reuse and Information Theory Based Metrics,” *Proc. 1991 Symposium on Applied Computing*, Kansas City, MO, 3-5 April 1991, pp. 437-46.

[43] Tracz, Will, “Software Reuse Maxims,” *ACM Software Engineering Notes*, Vol. 13, No. 4, October 1988, pp. 28-31.

[44] Tracz, Will, “A Conceptual Model for Megaprogramming,” *SIGSOFT Software Engineering Notes*, Vol. 16, No. 3 July 1991, pp. 36-45.

[45] Woodfield, Scott N., David W. Embley, and Del T. Scott, “Can Programmers Reuse Software,” *IEEE Software*, Vol. 4, No. 7, July 1987, pp. 168-175.

[46] Zhuo, Fang, Bruce Lowther, Paul Oman, and Jack Hagemeister, “Constructing and Testing Software Maintainability Assessment Models,” *Proceedings of the IEEE Computer Society International Software Metrics Symposium*, Baltimore, MD, 21-22 May 1993, pp. 61-70.

[47] Zuse, H., “Criteria for Program Comprehension Derived from Software Complexity Metrics,” *Proceedings IEEE Second Workshop on Program Comprehension*, Capri, Italy, 8-9 July 1993, pp. 8-16.