Seminar Report On

SOFTWARE REUSE

Submitted by
Ferose Babu TA

In the partial fulfillment of requirements in degree of
Master of Technology (M-Tech) in Software Engineering

DEPARTMENT OF COMPUTER SCIENCE

COCHIN UNIVERSITY OF SCIENCE AND TECHNOLOGY

KOCHI-682022
2007
ABSTRACT
Software reusability is the likelihood a piece of source code that can be used again to add new functionalities with slight or no modification. Programmers have always reused sections of code, functions and procedures. Code reuse is the idea that a partial or complete computer program written at one time is being or should be used in another program written at a later time. The concept of software reuse emerged as a recognized area after proposed basing of industry on reusable components. To achieve full potential of reuse, we need to focus our attention on development for reuse.

Organizations implementing systematic software reuse programs must be able to measure their progress and identify the most effective reuse strategies. Metrics and models with various reuse readiness levels can be used to measure reuse and reusability. Current reuse techniques focus on the reuse of software artifacts on the basis of desired functionality. Non-functional properties of a software system are also crucial. Quality concerns, therefore, should also be the focus for software reuse.

This Mathematical Theory of Intelligence structures objects in such a way that they become maximally reusable, interoperable and archival. The theory claims that reusability of an object is maximized if the object itself is defined as having been produced by maximizing reuse of the operations that were used to produce it. Many metrics and models have been developed for software reuse and reusability.
Keywords

Develop for reuse, Libraries, Components, Framework, modularity, coupling, cohesion, information hiding, separation of concerns and RRL.

Contents

1. Introduction 3
2. What is reuse 6
3. Why reuse software 9
4. Types of Reuse 15
  4.1 Opportunistic
  4.2 Planned
5. Design Techniques 11
  5.1 Modular 11
  5.2 Coupling and Cohesion 13
  5.3 Information Hiding 14
  5.4 Separation of concerns 14
6. Develop for reuse 15
  6.1 Development guidelines
  6.2 Automation
7. Metrics and models 15
  7.1 Cost Benefit Analysis
  7.2 Maturity Assessment
  7.3 Amount of Reuse
  7.4 Reuse Failure Modes
  7.5 Reusability Assessment
  7.6 Reuse Library Metrics
8. Quality 17
9. Reuse Readiness Levels (RRL) 18
   9.1 Topic Areas
   9.2 Readiness Levels
   9.3 Topic Area Levels
10. Standards
11. Theory of Reusability
12. Conclusion 41
13. References 42
1. Introduction

Ad hoc code reuse has been practiced from the earliest days of programming. Programmers have always reused sections of code, templates, functions, and procedures. Software reuse as a recognized area of study in software engineering, however, dates only from 1968 when Douglas McIlroy of Bell Laboratories proposed basing the software industry on reusable components. Code reuse is the idea that a partial or complete computer program written at one time can be, should be, or is being used in another program written at a later time. The reuse of programming code is a common technique which attempts to save time and energy by reducing redundant work.

Programmers may decide to create internal abstractions so that certain parts of their program can be reused, or may create custom libraries for their own use. Some characteristics that make software more easily reusable are modularity, loose coupling, high cohesion, information hiding and separation of concerns. For newly written code to use a piece of existing code, some kind of interface, or means of communication, must be defined. These commonly include a "call" or use of a subroutine, object, class, or prototype. In organizations, such practices are formalized and standardized by software product line engineering.

Development for reuse is the process of producing potentially reusable components. The emphasis of the research should be on development for reuse rather than development with reuse. The process of developing potentially reusable components depends on defining their characteristics. Reuse guidelines can represent such characteristics clearly. Therefore, we need to formulate objective and automatable reuse guidelines.
Many researchers have worked to make reuse faster, easier, more systematic, and an integral part of the normal process of programming. Current reuse techniques focus on the reuse of software artifacts on the basis of desired functionality. Systems fail because of inadequate performance, security, reliability, usability, or precision, to name a few. Capabilities to measure the reusability of software artifacts can have benefits for developers and adopters of information systems, but reusability is generally omitted from most measurements of technology readiness.

Motivation

“If you want to live a happy life, tie it to a goal, not to people or things”

- Albert Einstein

Goals for reuse

• Productivity
• Quality
• Duplication
• Available reuse assets
• Time to Market
• Maintenance

Time to market is an important competitive advantage, but not at the cost of quality. Reuse is the most powerful model to incorporate successful innovations quickly and at lower costs.
2. What is Software Reuse

Software reuse is the process of creating software systems from existing software rather than building software systems from scratch. Something that was originally written for a different project will usually be recognized as reuse. Code reuse is the idea that a partial or complete computer program written at one time can be, should be, or is being used in another program written at a later time. The reuse of programming code is a common technique which attempts to save time and energy by reducing redundant work. Software assets, or components, include all software products, from requirements and proposals, to specifications and designs, high level designs, data formats, algorithms to user manuals and test suites. Anything that is produced from a software development effort can potentially be reused.

Software developed and used repeatedly by the same people on the same project, Product maintenance and new product versions, use of operating systems, database management systems, and other system tools doesn’t amount to reuse.

Software engineering has been more focused on original development but it is now recognised that to achieve better software, more quickly and at lower cost, we need to adopt a design process that is based on systematic software reuse. For systematic reuse to succeed organizations must recognize that good components, frameworks, and software architectures require time to design, implement, optimize, validate, apply, maintain, and enhance. Creating reusable software assets requires a mature organization whose developers and architects can distinguish key sources of variability and commonality in their application domain. Identifying and separating these concerns for complex networked applications requires an iterative development process since it's hard to design reusable...
Software reuse artifacts correctly the first time using a top-down “waterfall” software lifecycle model.

3. Why Software Reuse

A good software reuse process facilitates the increase of productivity, quality, and reliability, performance and the decrease of costs, effort, risk and implementation time. An initial investment is required to start a software reuse process, but that investment pays for itself in a few reuses. In short, the development of a reuse process and repository produces a base of knowledge that improves in quality after every reuse, minimizing the amount of development work required for future projects, and ultimately reducing the risk of new projects that are based on repository knowledge.

Benefits

- Reusing code saves programming time, which reduces costs. If one person or team has already solved a problem, and they share the solution, there's no need to solve the problem again (with some potential caveats - see Drawbacks).

- Sharing code can help prevent bugs by reducing the amount of total code that needs to be written to perform a set of tasks. Generally, the more code a system contains the more bugs it's likely to have. The shared code can also be tested separately from the applications which use it.

- Separating code into common libraries lets programmers specialize in their particular strengths. A security library, for example, can be built by security
experts while a user interface which uses the library can let UI experts focus on their tasks.

- Repeatedly, separating code into specialized libraries lets each be tuned for performance, security, and special cases. For example, a Python application might delegate graphics functionality to a C library for performance.

- Delegation of tasks into shared modules allows offloading of some functionality onto separate systems. For example, a system specialized for fast read-only database queries can be used for reporting and accessed by multiple desktop applications.

- Proper and efficient reuse of code can help avoid code bloat. Bloated code contains unnecessary duplication and unused instructions. By efficiently sharing code across systems each individual component avoids duplicate or unneeded functionality.

**Drawbacks**

- Performance might become a factor.

- Depending on the platform and programming language, a library or framework might perform slower than desired. In some situations it might be beneficial to build a specialized one-time solution instead of using a common library.
Software reuse

- APIs accessed over a network will sometimes be slower than solving a problem within the local system.
- The system of modularity itself might create a bottleneck. For example, extra process initialization or shared library management can create overhead.
- Loss of control over 3rd party solutions might have negative repercussions. For example, there might be lack of support, desired feature enhancements might not get added, or security might not be fully tested. Outside the technical considerations, there might also be licensing and liability issues.
- When not well implemented or when taken too far, code reuse can eventually cause code bloat. Ironically, adding modularity can eventually lead to lingering APIs and libraries which go unused. In very large systems it's not uncommon to lose track of how every component is used. Over time a component may become useless, but linger in the system. This, however, is not so much an inherent drawback of code reuse as it's a problem of implementation.
3. Types

**Opportunistic reuse** - While getting ready to begin a project, the team realizes that there are existing components that they can reuse. Opportunistic reuse can be categorized as

*Internal reuse* - A team reuses its own components. This may be a business decision, since the team may want to control a component critical to the project.

*External reuse* - A team may choose to license a third-party component. Licensing a third-party component typically costs the team 1 to 20 percent of what it would cost to develop internally.[1] The team must also consider the time it takes to find, learn and integrate the component.

**Planned reuse** - A team strategically designs components so that they'll be reusable in future projects.

4. Design Techniques

Some characteristics that make software more easily reusable are

- Modularity
- Loose coupling
- High cohesion
- Information hiding
- Separation of concerns
Modular programming

A design technique that increases the extent to which software is composed of separate parts, called modules. Conceptually, modules represent a separation of concerns, and improve maintainability by enforcing logical boundaries between components.[1] Modules are typically incorporated into the program through interfaces. A module interface expresses the elements that are provided and required by the module. The elements defined in the interface are detectable by other modules. The implementation contains the working code that corresponds to the elements declared in the interface.

With Modular Programming, concerns are separated such that no (or few) modules depend upon other modules of the system. To have as few dependencies as possible is the goal. When creating a modular system, instead of creating a monolithic application (where the smallest component is the whole application), several smaller modules are built (and usually compiled) separately that, when composed together, will construct the executable application program. A just in time compiler may perform some of this construction "on-the-fly" at run time.

Loose coupling

Coupling refers to the degree of direct knowledge that one class has of another. This is not meant to be interpreted as encapsulation vs. non-encapsulation. It is not a reference to one class's knowledge of another class's attributes or implementation, but rather knowledge of that other class itself. Strong coupling occurs when a dependent class contains a pointer directly to a concrete class which provides the required behavior. Loose coupling occurs when the dependent class contains a pointer only to an interface, which can then be
implemented by one or many concrete classes. Loose coupling provides extensibility to designs. A new concrete class can easily be added later that implements that same interface without ever having to modify and recompile the dependent class. Strong coupling does not allow this.

**High cohesion**

Cohesion is a measure of how strongly-related is the functionality expressed by the source code of a software module. cohesion is a measure of how strongly-related or focused the responsibilities of a single module are.

**Information hiding**

Information hiding is the principle of segregation of design decisions in a computer program that are most likely to change, thus protecting other parts of the program from extensive modification if the design decision is changed. The protection involves providing a stable interface which protects the remainder of the program from the implementation (the details that are most likely to change).

A common use of information hiding is to hide the physical storage layout for data so that if it is changed, the change is restricted to a small subset of the total program.
**Separation of concerns**

SoC is the process of separating a computer program into distinct features that overlap in functionality as little as possible. A concern is any piece of interest or focus in a program. Typically, concerns are synonymous with features or behaviors. Progress towards SoC is traditionally achieved through modularity of programming and encapsulation (or "transparency" of operation), with the help of information hiding. Layered designs in information systems are also often based on separation of concerns (e.g., presentation layer, business logic layer, data access layer, database layer).

6. Development for reuse

Development for reuse is a process of producing potentially reusable components. The emphasis here is on development for reuse rather than development with reuse, which is a process of normal systems development. The process of developing potentially reusable components depends solely on defining their characteristics such as language features and domain abstractions. Reuse guidelines can represent such characteristics clearly. Therefore, we need to formulate objective and automatable reuse guidelines.

The important factors for formulating objectives and guidelines are

1. Assessing the reusability of software components against objective reuse guidelines.

2. Providing reuse advice and analysis.
3. Improving components for reuse which is the process of modifying and adding reusability attributes.

Language-oriented reuse guidelines

Most existing programming languages including object-oriented languages provide features that support reuse. However, simply writing code in those languages doesn't promote reusability. Components must be designed for reusability using those features. Such features must be listed as a set of design techniques for reusability before design takes place.

Domain-oriented reuse guidelines

Guidelines those are relevant to a specific application domain.

6.1. Development guidelines
The objective of the design process is to produce components which are potentially reusable. These components form building blocks for future development and are applicable for various situations and perhaps across application domains.

In development with reuse, reuse is desirable but there need be no resources expended in creating new reusable components. Development for reuse implies expending resources specifically to increase the reusability of components. In many cases, this process might follow development with reuse where components generated during normal system development are made more reusable by generalisation and improvement. Figure 1 shows number of stages to be followed which start from identifying an application domain, identify & classify reusable abstractions, domain-oriented reuse, language-oriented reuse, design components, assessment for reuse, improvement for reuse, and deliver potentially reusable components. The idea is to identify a number of frequently reusable domain-specific abstractions and then to apply domain-specific and language-specific criteria that are defined by the reuse guidelines.

**Identify domain**

Domain analysis has been identified as essential for effective reuse. The first step is to identify a specific application domain and define its boundary.

**Identify and classify frequently reusable abstractions**
To identify potentially reusable components, the reuse assessor must know what the important domain abstractions are and how frequently these abstractions are used in systems developed for that domain. There is not much point in devoting a lot of effort in producing a reusable domain abstraction if that abstraction is rarely used. Domain classification helps to identify effective reusable abstractions. This stage involves interviewing domain experts, surveying domain literature and studying existing systems.

**Identify design/programming language constructs that support reuse**

Selecting an appropriate language is an important part of development for reuse. We should be able to express our reuse guidelines effectively using language mechanisms.

**Study and formulate language reuse guidelines**

This emphasises the effective use of language features for reuse. This process includes studies of existing techniques and appropriate modifications to them.

**Study and formulate domain reuse guidelines**
Software reuse

This emphasises the reusable domain abstractions that are identified in the application domain. Guidelines should not just be general advice but should be specific and verifiable for creating potentially reusable components. Design/redesign components based on these guidelines.

**Reuse assessment**

Reuse assessment is a process of assessing components based on the number of guidelines satisfied against the total number of guidelines that are applicable, and then produce an assessment report. This is where we need to automate this process. The outcome of this process is to make sure that the components designed for reuse satisfy some of the key characteristics.

**Reuse improvement**

Reuse improvement is a process of modifying and improving these components for reuse by adding attributes of an abstraction for reuse. This process is based on the assessment report produced during the previous step. The reuse improver must know what attributes of an abstraction must be generalised to make it reusable. Again, an automatic reuse improvement is essential. Finally, produce potentially reusable components.
Where possible, these two processes of assessing and improving components for reuse.

6.4. Automation

The guidelines discussed can be partially or completely automated as shown in Figure 2. This system takes a component, checks through various reuse guidelines that are applicable, provides reuse advice and analysis to the reuser, and generates that component which is improved for reuse. Components are modelled using component templates and reuse guidelines are checked objectively against that template. Some of these domain reuse guidelines have been represented and analysed using component templates. For most of these guidelines, automation depends on some user interactions and domain knowledge.

One of the major objectives of this system is to demonstrate, how well-defined reuse guidelines can be used to automate the process of reuse assessment by providing support for language analysis and domain analysis. The system interacts with the engineer to discover information that can't be determined automatically. The conclusion of this first pass is an estimate of how many guidelines are applicable to the component and how many of these have been breached. The
Software reuse

The report generator produces a report with all the information that has been extracted about that component and changes that have been made for reuse.

The second pass involves applying domain knowledge to the system. The component templates have been modelled representing static and dynamic structures. Their reusability is assessed by comparing the component with that template. The support provided by the system ensures that the reuse engineer carries out a systematic analysis of the component according to the suggested guidelines. He or she need not be a domain expert. Again, an analysis is produced which allows the engineer to assess how much work is required to improve system reusability.

Guidelines for automation are represented in two distinct ways:

• Wherever possible, a rule-based representation is used so that it is clear when a guideline should be applied. We have found that rule-based representations are mostly applicable for language-oriented guidelines.

• For domain-oriented guidelines, we are mostly concerned with checking that a component fits a model of a reusable domain abstraction.
7. Metrics and Models

Software reuse 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. Organizations implementing systematic software reuse programs must be able to measure their progress and identify the most effective reuse strategies. A metric is a quantitative indicator of an attribute of a thing. A model specifies relationships among metrics.

Figure 3 Categorization of reuse metrics and models
7.1. Cost Benefit Analysis

As organizations contemplate systematic software reuse, the first question of concern will be 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. 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 in the following.

Cost/Productivity Models

Two cost and productivity models for software reuse are in practice [4]. 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 ($R \leq 1$). (Note that $R$ is a type of reuse level; this topic is discussed in detail in Section 4.) $b$ 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:

$$[(\text{relative cost of all new code}) \times (\text{proportion of new code})] + [(\text{relative cost of reused software}) \times (\text{proportion of reused software})].$$
The equation for this is:

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

and the corresponding relative productivity is,

\[ P = \frac{1}{C} \]
\[ = \frac{1}{((b-1)R+1).} \]

b must be < 1 for reuse to be cost effective. The size of b depends on the life cycle 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 because 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.

**Quality of Investment**

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.

Quality of investment (Q) is the ratio of reuse benefits (B) to reuse investments (R)

\[ Q = \frac{B}{R}. \]

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:

- Increase the level of reuse
- Reduce the average cost of reuse
- Reduce the investment needed to achieve a given reuse benefit

Business Reuse Metrics

A set of metrics is used 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. Here, cost is broken down into development costs and maintenance costs. The metrics are derived from a set of data elements defined in Table 1. Given the preceding data, the following metrics are defined:

<table>
<thead>
<tr>
<th>Sl No</th>
<th>Data Element</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Shipped source instructions</td>
<td>SSI</td>
</tr>
<tr>
<td>2</td>
<td>Changed source instructions</td>
<td>CSI</td>
</tr>
<tr>
<td>3</td>
<td>Resourced source instructions</td>
<td>RSI</td>
</tr>
<tr>
<td>4</td>
<td>Source instructions reused by others</td>
<td>SIRBO</td>
</tr>
</tbody>
</table>
Reuse percent reflects how much of the product can be attributed to reuse. This distinguishes the reuse percent of a product, the reuse percent of a product release, and the reuse percent for the entire organization.

Product Reuse percent = \( \frac{RSI}{RSI + SSI} \times 100 \) percent.

Reuse cost avoidance measures reduced total product costs as a result of reuse. Various estimates 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. This study also acknowledges savings that are realized in the maintenance phase as 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 = \( RSI \times 0.8 \times (\text{new code cost}) \)
Service cost avoidance = \( RSI \times (\text{error rate}) \times (\text{new code cost}) \)
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.

Reuse value added = \( \frac{SSI + RSI + SIRBO}{SSI} \).

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) * code written for reuse
by others * 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.

7.2. 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. 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.

7.3. 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

Amount of life cycle object reused / total size of life cycle object
A common form of this metric is based on lines of code as follows

\[
\text{Lines of reused code in system or module} / \text{total lines of code in system or module}
\]

*Amount of Reuse* metric by defining reuse level metrics that include factors such as abstraction level of the life cycle objects and formal definitions of internal and external reuse. Work is also underway to define metrics specifically for object oriented systems.

The basic dependent variable in software reuse improvement efforts is the level of reuse. 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. 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 that are not from an external repository.
- \( M = \) the number of items not from an external repository that are used more than once.
These counts are of unique items, not tokens. Given these quantities, the following reuse level metrics are defined:

External Reuse Level = \( \frac{E}{L} \)

Internal Reuse Level = \( \frac{M}{L} \)

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), level of abstraction, and amount of usage.

The variables and reuse level metrics are:

\[ ITL \text{ = internal threshold level, the maximum number of times an internal item can be used before it is reused.} \]

\[ ETL \text{ = external threshold level, the maximum number of times an external item can be used before it is reused.} \]

\[ IU \text{ = number of internal lower level items that are used more than ITL.} \]

\[ EU \text{ = number of external lower level items that are used more than ETL.} \]

\[ T \text{ = total number of lower level items in the higher level item, both internal and external.} \]

Internal Reuse Level = \( \frac{IU}{T} \)

External Reuse Level = \( \frac{EU}{T} \)
Software reuse

<table>
<thead>
<tr>
<th>Component</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal Reuse</td>
<td>IUF/TF</td>
</tr>
<tr>
<td>External Reuse</td>
<td>EUF/TF</td>
</tr>
<tr>
<td>Total Reuse</td>
<td>(IUF + EUF)/TF</td>
</tr>
</tbody>
</table>

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 noncommentary 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:

- Internal reuse level;
- External reuse level;
- Total reuse level;
Internal reuse frequency;
External reuse frequency;
Total reuse frequency;
Complexity (size) weighting for internal functions.

7.4. Reuse Failure Modes

Implementing systematic reuse is difficult, involving both technical and nontechnical factors. Failure modes analysis provides an approach to measuring and improving a reuse process based on a model of the ways a reuse process can fail. The reuse failure modes model reported by Frakes and Fox [1996] 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 Does Not Exist
- Part Is Not Available
- Part Is Not Found
- Part Is Not Understood
- Part Is Not Valid
Part Can Not 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.

7.5. 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 reengineering 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 reengineering. 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.

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
7.6 Reuse Library Metrics

As can be seen in Figure 4, 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 reengineering, designed and built from scratch, or 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 1990]. The evaluation criteria for indexing schemes of reuse libraries are: costs, searching effectiveness, support for understanding, and efficiency.
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, and 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 the total of items retrieved. Another effectiveness measure is overlap, which reports the percentage of relevant documents retrieved jointly by two methods. Frakes and Pole [1994], 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 [1994] used a 7-point ordinal scale (7 5 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 nonfunctional 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 can be 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 [1987] 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 are a good indication that the module is of higher 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 percent and branch coverage of at least 80 percent.

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* assigns users to the following categories:

- Government/contractor, commercial, academia, NonUS;
- Type of library function performed searches, browses, extracts;
8. Quality in Software Reuse

Software reuse has been a goal for Software Engineering practice, as a means to reduced development cost and improved quality. Current reuse techniques focus on the reuse of software artifacts on the basis of desired functionality. However, non-functional properties of a software system are also crucial. Systems fail because of inadequate performance, security, reliability, usability, or precision, to name a few. Quality concerns, therefore, should also be front and centre in methods for software reuse.

One must look for, and possibly adopt, a reusable component that meets stringent requirements in precision, performance and reliability. Despite this practical need, few methods for reuse have focused on non-functional requirements. The typical object of software reuse as surveyed is an artifact, initially executable code, and more recently large-scale components, software architectures, designs, frameworks, and software product lines. All of these are predominantly reused on the basis of functionality. One will not find precision, performance or reliability as components ready-made for reuse. Improved software productivity and reduced
Software reuse

development costs result from building with reuse; building for reuse actually has an overhead cost.

Why is it hard to incorporate quality requirements into reuse methods? One important reason for this is that software artifacts include both functional and quality fragments. Some of the quality fragments are hard to recognize since they are mingled with the functional fragments in order to be executable.

One approach is to focus on qualities as reusable assets. Would it be possible to separate knowledge about how to achieve a quality, such as performance from a specific function, say interpolation? Would it be possible and reasonable to look for knowledge on performance, instead of looking for different implementations of interpolation? Would it be possible to retrieve useful knowledge relative to a concern that is applicable in several domains?

Unfortunately, the cross-cutting nature of quality attributes in software makes them hard to classify [2, 5]. In function-oriented classification, quality information is not apparent in the reusable artifact. This difficulty increases when there are multiple quality concerns being dealt within one artifact. To overcome these difficulties, we should combine techniques from research in non-functional requirements [6, 7], goal-oriented requirements engineering [8], aspect oriented programming [9], software reuse [3] and quality management [10]. The results of aspect-oriented programming with goal-oriented requirements engineering combination proves to be effective because it unites a goal refinement and classification strategy with a packing strategy provided by aspect-oriented programming, making use of well-defined relations among functional and quality fragments. It is possible to store qualities, retrieve it for reuse, specialize it for different contexts and integrate it with functional descriptions.
9. Reuse Readiness Levels (RRL)

Recognizing the need to measure the maturity of software for reuse, the NASA Earth Science Data Systems (ESDS) Software Reuse Working Group (WG) proposes a set of Reuse Readiness Levels (RRLs). The maturity of a particular technology can be measured in various ways, one common method being with Technology Readiness Levels (TRLs) or other similar measurements. However, the ability or readiness of a particular technology to be reused is generally not considered, or plays only a small role if it is considered.

Recognizing that the existing TRLs are a useful measure and have been successfully applied within their domain, the WG decided to use NASA”s TRLs [11] as a guide while developing RRLs. In particular, the WG agreed to have RRLs ranging from 1 to 9 (inclusive), to align with the familiar TRL scale.

9.1 Topic Areas

Through extensive discussions, the WG identified the nine topic areas that were deemed important for measuring the reuse maturity of software. Alphabetically, they are:

Documentation

*Information that describes the software asset and how to use it.*

Documentation consists of installation and developer guides, development methodologies and documentation of the support available, API specifications,
commented code and build instructions, technical support instructions and support forums, technical manuals, libraries and tutorials, and reuse and deployment case studies. This documentation may be in various stages of development and accessibility, and may not have a clear audience defined.

Documentation enables potential adopters to determine whether the software addresses the need and informs adopters how to utilize the software and reduce the risks and costs of reuse. Documentation includes descriptions of interfaces and capabilities, information about the execution environment, and instructions for the consumer on the purpose of the asset and on ways in can be reused. Documentation also describes plans for subsequent releases and future development.

**Extensibility**

*The ability of the asset to be grown beyond its current context.*

The implementation takes into consideration future growth and ease of extending function. A measure of the ability to extend a system and the level of effort required to implement the extension. Extensions, or expandability, can apply to re-engineering or during runtime.

Extensibility is an important dimension to be able to incorporate an asset and add to or modify its functionality.

**Intellectual Property Issues**

*The legal rights for obtaining, using, modifying and distributing the asset.*

A formal and documented explanation of the involved parties and roles, with binding statements describing any licensing mechanisms, ownership rights,
restrictions, and user/consumer responsibilities related to the distribution and reuse of assets. The legal rights are established in accordance with the policies and laws of the organization that originally produced the software. Potential adopters need to understand the intellectual property issues to know whether they have the authority to reuse the software.

Modularity

*The degree of segregation and containment of an asset or components of an asset.*

Modularity is a software design technique that increases the extent to which software is composed from separate components, called modules. Conceptually, modules represent a separation of and encapsulation of concern, purpose, and function, and they improve maintainability and reusability. Modular assets generally are easier to synthesize and extend.

Packaging

*The methodology and technology for assembling and encapsulating the components of a software asset.*

Packaging pertains to the technologies, standards, and procedures related to gathering, organizing, assembling, and compressing the parts of a software system and distributing it as a collection. Packaging is important to ensure completeness, to allow distribution, and to simplify the installation of the asset.
Software reuse

Portability

The independence of an asset from platform-specific technologies.
Portability refers to two components: software consisting of source code that can be compiled for various computing platforms; software executables that can be executed on various platforms.
The ability to be installed or executed on various platforms maximizes reuse potential and increases the flexibility and (re-)usability of the asset and its applications.

Standards Compliance

The adherence of an asset to accepted technology definitions.
Concerning commonly accepted criteria, models, patterns and/or specifications have been followed in the creation of a reusable asset; and at what level the asset complies with the standard.
By complying with accepted standards, the asset has increased potential for adoption.

Support

The amount and type of assistance available to users of the asset.
Technical support exists, in the form of various communication methods with the asset’s developers, documentation/knowledge bases, user communities, support level agreements, and online forums. A release strategy and plan for patches and versions has been created.
Support provisions expertise to assist in maintenance, evolution, extension and issue resolution.

**Verification and testing**

*The degree to which the functionality and applicability of the asset has been demonstrated.*

This can be realized through the provision of test material, requirements compliance, proper function, and usability (robustness). Tests documented, results analyzed and published, and fixes and enhancements applied.

Sufficient verification and testing increases the accuracy and confidence and reduces potential risks and costs of reuse.

### 9.2 Readiness Levels

Following the format of TRLs, there are nine Reuse Readiness Levels (RRLs) ranging from 1 (least mature) to 9 (most mature).

**RRL 1 – Limited reusability; the software is not recommended for reuse.**

Little is provided beyond limited source code or pre-compiled, executable binaries. There is no support, contact information for developers or rights for reuse specified, the software is not extensible, and there is inadequate or no documentation.
RRL 2 – Initial reusability; software reuse is not practical.

Some source code, documentation, and contact information are provided, but these are still very limited. Initial testing has been done, but reuse rights are still unclear. Reuse would be challenging and cost-prohibitive.

RRL 3 – Basic reusability; the software might be reusable by skilled users at substantial effort, cost, and risk.

Software has some modularity and standards compliance, some support is provided by developers, and detailed installation instructions are available, but rights are unspecified. An expert may be able to reuse the software, but general users would not.

RRL 4 – Reuse is possible; the software might be reused by most users with some effort, cost, and risk.

Software and documentation are complete and understandable. Software has been demonstrated in a lab on one or more specific platforms, infrequent patches are available, and intellectual property issues would need to be negotiated. Reuse is possible, but may be difficult.

RRL 5 – Reuse is practical; the software could be reused by most users with reasonable cost and risk.
Software is moderately portable, modular, extendable, and configurable, has low-fidelity standards compliance, a user manual, and has been tested in a lab. A user community exists, but may be a small community of experts. Developers may be contacted to request limited rights for reuse.

**RRL 6 – Software is reusable; the software can be reused by most users although there may be some cost and risk.**

Software has been designed for extensibility, modularity, and portability, but software and documentation may still have limited applicability. Tutorials are available, and the software has been demonstrated in a relevant context. Developers may be contacted to obtain formal statements on restricted rights or to negotiate additional rights.

**RRL 7 – Software is highly reusable; the software can be reused by most users with minimum cost and risk.**

Software is highly portable and modular, has high-fidelity standards compliance, provides auto-build installation, and has been tested in a relevant context. Support is developer-organized, and an interface guide is available. Software and documentation are applicable for most systems. Brief statements are available describing limited rights for reuse and developers may be contacted to negotiate additional rights.
RRL 8 – Demonstrated local reusability; the software has been reused by multiple users.

Software has been shown to be extensible, and has been qualified through test and demonstration. An extension guide and organization-provided support are available. Brief statements are available describing unrestricted rights for reuse and developers may be contacted to obtain formal rights statements.

RRL 9 – Demonstrated extensive reusability; the software is being reused by many classes of users over a wide range of systems.

Software is fully portable and modular, with all appropriate documentation and standards compliance, encapsulated packaging, a GUI installer, and a large support community that provides patches. Software has been tested and validated through successful use of application output. Multiple statements describing unrestricted rights for reuse and the recommended citation are embedded into the product.

9.3 Topic Area Levels

The detailed topic area levels described here are labeled as “Level”, but their numbering corresponds to the overall RRLs. This resulted in some topic areas with less than nine levels missing some level numbers, as the levels present were spread out over the entire nine-level range of the overall RRLs.
**Documentation**

Level 1 – Little or no internal or external documentation available. Source code is available, with little or no useful internal or external documentation.

Level 2 – Partially to fully commented source code available. Source code is available and fully commented, but no other documentation is provided. It may be challenging for a good programmer to determine how to reuse the software.

Level 3 – Basic external documentation for sophisticated users available. For example, a README file, a “man” page, or command line usage examples. This type of documentation would be sufficient for a sophisticated user to figure out how to use the software, but probably not a general user.

Level 4 – Reference manual available. Reference manual provides complete documentation on use of the software, but may not be easily approached or accessed by general users. Some documentation relevant to customization is available.

Level 5 – User manual available. User manual allows a “normal” or general user to understand how to use and possibly customize aspects of the software.

Level 6 – Tutorials available. Step-by-step walkthroughs of how the software is customized and used in various scenarios, demos, etc. This makes it very easy to understand/teach the software and use it in a new project.

Level 7 – Interface guide available. Documentation describes how to customize and interface the software with other software, programmatic interfaces, APIs, etc., so that it can more easily be embedded in a larger system.
Level 8 – Extension guide and/or design/developers guide available.

An extension guide provides information on how to customize and add to the software, add plug-ins and the like, use internal programming “languages”, etc. A design/developers guide provides a description of internals, design documentation, internal documentation, etc. that is sufficient for someone “skilled in the art” to contribute to the development of the software or take over maintenance of the software.

Level 9 – Documentation on design, customization, testing, use, and reuse is available.

All stages of the software engineering lifecycle are fully documented. This includes design and review artifacts, testing artifacts, customization, and regression tests. The documentation provided is easy to read/access and is appropriate for different categories of users.

Extensibility

Level 1 – No ability to extend or modify program behavior.

Source code is not available; execution parameters cannot be changed, and/or it is not possible to extend the functionality of the software, even for application contexts similar to the original application domain.

Level 2 – Very difficult to extend the software system, even for application contexts similar to the original application domain.

The software was not designed with extensibility in mind. While some level of documentation and/or source code is available, it is extremely difficult to extend the software. For cases where source code is available, the logical flow of code may be hard to follow, with few (if any) comments, and little to no cohesion.
Level 3 – Extending the software is difficult, even for application contexts similar to the original application domain. Minimal consideration to extensibility is included in the design, through use of methods such as object-oriented design or other tools which provide logical cohesion. Where source code is available, the software has some structure, but may have a high number of independent logical paths, minimal comments and documentation, and/or a low degree of cohesion.

Level 4 – Some extensibility is possible through configuration changes and/or moderate software modification. Consideration to extensibility to some range of application contexts is included in the design though means such as (a) use of configuration files, (b) isolation of configuration parameters and constants in clearly identified sections of source code (distinct from logic and display code), (c) some documentation of the effects of changes to these parameters and the allowed values for these parameters, and/or (d) effective use of programming practices designed to enable reuse, such as object oriented design.

Level 5 – Consideration for future extensibility designed into the system for a moderate range of application contexts; extensibility approach defined and at least partially documented. The procedures for extending the software are defined, whether by source code modification (e.g., object-oriented design) or through the provision of some type of extension functionality (e.g., callback hooks or scripting capabilities). Where source code modification is part of the extension plan, the software is well-structured, has a moderate to high level of cohesion, and has configuration elements clearly separated from logic and display elements. Internal and external documentation are sufficient to allow an experienced programmer to understand program flow and logic with moderate effort.
Software reuse

Level 6 – Designed to allow extensibility across a moderate to broad range of application contexts, provides many points of extensibility, and a thorough and detailed extensibility plan exists. The extensibility capability for the software is well defined, sufficient to enable an experienced developer generally familiar with the project to extend the software. That documentation should include clear information about the range of application contexts to which the software can be extended as well as potential limitations on expansion.

Level 7 – Demonstrated to be extensible by an external development team in a similar context. The software has been extended and applied to a similar application context to the original. This extension may have been done by an external team using extension documentation, by may have involved substantial assistance from the original development team members.

Level 8 – Demonstrated extensibility on an external program, clear approach for modifying and extending features across a broad range of application domains. The software has been extended by at least one group of users outside the original development group using existing documentation and with no assistance from the original development team.

Level 9 – Demonstrated extensibility in multiple scenarios, provides specific documentation and features to build extensions which are used across a range of domains by multiple user groups. The software is regularly extended externally by users across multiple applications using available documentation. There may be a library available of user-generated content for extensions.
Software reuse

Intellectual Property Issues

Level 1 – Developers have been identified, but no rights have been determined. Product developers have been identified and their responsibilities have been determined, but they have not considered or determined the rights for the product.

Level 2 – Developers are discussing rights that comply with their organizational policies.

Level 3 – Rights agreements have been proposed to developers. Each developer has received a draft intellectual property rights agreement that would result from cooperative activities with other developers. Rights are not specified.

Level 4 – Developers have negotiated on rights agreements. Developers have reviewed proposals from each of the other developers and have proposed an agreement that addresses any potential conflicts in the proposed intellectual property rights and responsibilities for development. A limited rights statement has been drafted and developers may be contacted to negotiate rights for reuse.

Level 5 – Agreement on ownership, limited reuse rights, and recommended citation. Developers have agreed on proposed ownership, limited intellectual property rights for reuse, and responsibilities. Order of developers' names, recommended citation, and agreements have been formalized. Developers may be contacted to obtain formal statements on restricted rights for reuse.

Level 6 – Developer list, recommended citation, and rights statements have been drafted.
Agreements on development responsibilities, the list of developers, a recommended citation, and intellectual property rights statements, offering limited rights for reuse have been drafted and are included in package. Developers may be contacted to obtain formal statements on restricted rights or to negotiate additional rights.

Level 7 – Developer list and limited rights statement included in product prototype.

A list of developers, recommended citation, and intellectual property rights statements, including copyright or ownership statements, are embedded in the source code of the product, in the documentation, and in the expression of the software upon execution. These include any legal language that has been approved by all parties or their representatives, machine-readable code expressing intellectual property, and concise statements in language that can be understood by laypersons, such as a pre-written, recognizable license. Brief statements are available describing limited rights, restrictions, and conditions for reuse. Developers may be contacted to negotiate additional rights.

Level 8 – Recommended citation and intellectual property rights statement included in product.

All parties have reviewed the list of developers, recommended citation, and intellectual property rights statements, including limited rights for reuse, in the product to ensure that all interests are represented and that the statements conform to their institutional policies and agreements. Brief statements are available describing unrestricted rights and any conditions for reuse. Developers may be contacted to obtain formal rights statements.

Level 9 – Statements describing unrestricted rights, recommended citation, and developers embedded into product.
Multiple statements are embedded into the product describing unrestricted rights and any conditions for reuse, including commercial reuse, and the recommended citation. The list of developers is embedded in the source code of the product, in the documentation, and in the expression of the software upon execution. The intellectual property rights statements are expressed in legal language, machine-readable code, and in concise statements in language that can be understood by laypersons, such as a pre-written, recognizable license.

**Modularity**

Level 1 – Not designed with modularity.
Research or prototype-grade code written with no designs for organizing code in terms of functionality for modularity or reuse.

Level 3 – Modularity at major system or subsystem level only.
No clear distinctions between generic and solution-specific functionality; few internal functions accessible by external programs (i.e., closed architecture), limited distinction between visible functions; code is organized into a primary system that provides general functionality and one or two subsystems that each provide multiple, unrelated, functions; code within each module contains many independent logical paths.

Level 5 – Partial segregation of generic and specific functionality.
Top to bottom structuring into individual components that provide functions or services to outside entities (i.e., open architecture); internal functions or services documented, but not consistently; modules have been created for generic functions, but modules have not been created for all of the specified functions; code within each module contains many independent logical paths.

Level 7 – Clear delineations of specific and reusable components.
Organization of all components into libraries or service registries; consistent documentation of all libraries as APIs or standard web service interfaces; modules have been created for all specified functions and organized into libraries with consistent features within interfaces; code within each module contains many independent logical paths.

Level 9 – All functions and data encapsulated into objects or accessible through web service interfaces.

All functions and data encapsulated into objects or accessible through web service interfaces; consistent error handling; use of generic extensions to program languages for stronger type checking and compilation-time error checking; services available externally, e.g., in “third-party” service workflows; code within each module contains few independent logical paths.

Packaging

Level 1 – Software or executable available only, no packaging.

Inadequate or no documentation and no auto-build/install facility is available.

Level 3 – Detailed installation instructions available.

System includes auto-build feature, but is built for a particular operating system.

Level 5 – Software is easily configurable for different contexts.

For example, locations of resources (files, directories, URLs) are configurable. All configuration-specific information is centralized.

Level 7 – OS-detect and auto-build for supported platforms available.

Operating system detection configuration files are available. Packaging includes auto-build for supported OS platforms and suite of regression tests for platform-specific testing.

Level 9 – Installation user interface provided.
A user interface guides the installer through all steps needed to build, configure, and install the software.

**Portability**

Level 1 – The software is not portable.
No source code or instructions for customization are provided. Executable binaries are provided and there are known severe limitations for running it on the hardware or operating system. There is only minimal information on installation or use. There is no information on porting to another platform or application.

Level 2 – Some parts of the software may be portable.
Some source code is provided with some internal and external documentation. Binaries are provided and there is some documentation on how to install the software. There is no useful information on porting. Porting is prohibitively expensive, but some portions (e.g. modules, functions) of the code may be portable.

Level 3 – The software is only portable with significant costs.
The complete source code is available, without external dependencies that are portable, but the software cannot be ported without significant changes to the software or the target context. Documentation on porting the code to another platform or application is missing or deficient. Porting would not be practical or cost effective.

Level 4 – The software may be portable at a reasonable cost.
The cost benefits of using the software slightly outweigh the cost of developing new software. Documentation is barely sufficient, but may contain some useful information on porting to another platform or application. Porting will nonetheless
require significant effort. Only at this level is it generally worth considering porting the software.

Level 5 – The software is moderately portable.
The software can be ported with only relatively small changes necessary to the context or the software itself. Documentation on porting exists and is complete, but requires considerable effort and expertise. Some rudimentary understanding of the underlying software or the target system may be necessary.

Level 6 – The software is portable.
The software can be ported to most major systems without modification. The documentation, however, addresses porting to a large number of systems that are identified. Any modifications needed to port the software to these systems are well described in the documentation and would be relatively easy to implement.

Level 7 – The software is highly portable.
The software can be ported to all but the most obscure or obsolete systems without modification. The documentation is complete and thorough. No changes to the software are necessary and the effort to port the software is minimal.

Level 9 – The software is completely portable.
The software can be ported to all systems since it runs on an application layer rather than on the underlying operating system layer. Such software is written in languages Java, C#, etc. In theory at least, the software will run on any system in which the appropriate application layer has been installed.

Standards Compliance

Level 1 – No standards compliance.
Neither the software nor the software development process adheres to any identified standards other than those inherent in the software languages employed.
Level 2 – No standards compliance beyond best practices.
The software and software development process adhere, at least in part, to some common best practices, but do not identify or claim compliance with any recognized standard.
Level 3 – Some compliance with local standards and best practices.
The software and software development process comply with standards and best practices defined locally by the development organization.
Level 4 – Standards compliance, but incomplete and untested.
The software and software development process attempt to comply with recognized standards, but without verification. Standards compliance is thus untested and may not be complete.
Level 5 – Standards compliance with some testing.
The software and software development process comply with recognized standards, but verification of compliance is incomplete. Standards compliance may not be followed by all components.
Level 6 – Verified standards compliance with proprietary standards.
The software and software development process comply with specific and proprietary standards (such as Windows GUI) and compliance with those standards has been verified through testing.
Level 7 – Verified standards compliance with open standards.
The software and software development process comply with specific open standards and compliance with those standards has been verified through testing.
Level 8 – Verified standards compliance with recognized standards.
The software and software development process comply with internationally recognized standards such as W3C, XML, XHTML, WAI, IP for Web; or ANSI/ISO (C/C++), JCP (Java), for software; and CMMI, IEEE Software
Software reuse

Engineering Standards for development process. Standards compliance has been verified through testing, but not by an independent testing organization.

Level 9 – Independently verified standards compliance with recognized standards.

The software and software development process comply with internationally recognized standards. Independent and documented standards compliance verification is included with the software. The development organization maintains standards compliance in its development process through regular testing and certification from an independent group.

Support

Level 1 – No support available.

The original developer of the code is not known, not locatable, or is refusing support.

Level 2 – Minimal support available.

There is known contact information available for the original developer(s) and they are willing to provide minimal, occasional support.

Level 3 – Some support available.

Contact information is available and there is a willingness to provide some support infrequently, without guarantees. This may include things such as providing makefiles or different flavors of the code for different contexts.

Level 4 – Moderate systematic support is available.

Latest updates/patches are usually made available. Support is available, but may be intermittent.

Level 5 – Support provided by an informal user community.

There is an informal user community that provides answers, for example, via a Web site FAQ.
Level 6 – Formal support available.
Support is centralized in a web site containing relevant resources, answers to FAQ, and other useful information.

Level 7 – Organized/defined support by developer available.
There is organized and defined support by the developer with email/telephone help desk and links to case studies and other relevant information. No continuity of support implied.

Level 8 – Support available by the organization that developed the asset.
The support is by an organization and is well defined with frequent updates, releases, etc., and help desk. Continuity of support is implied. Support may be free or fee-based and may be offered by a third party.

Level 9 – Large user community with well-defined support available.
This may include resources such as a help desk, a Web site for the latest information, an active discussion group willing to answer questions, frequent patches and updates as well as consolidation of changes by the community. One example would be the Linux operating system.

**Verification and Testing**

Level 1 – No testing performed.
Ideas have been translated into software development. Examples might include studies of development languages, prototype, or diagram of interface. Requirements have not been verified, and there is no formal test mechanism in place.

Level 2 – Software application formulated and unit testing performed.
Software application compiles, and executes with known inputs. For example, a prototype application where there is no testing or validation to support the
software, but only testing to demonstrate a prototype. Requirements may not be finalized yet, or overall testability of the software determined.

Level 3 – Testing includes testing for error conditions and handling of unknown input.

Software applications have been „white box“ tested. This includes both known and unexpected inputs to the application. This level of testing has been incorporated into the build and/or deployment mechanism.

Level 4 – Software application demonstrated in a laboratory context.

Following successful testing of inputs and outputs, the testing has integrated an application to establish that the “pieces” will work together to achieve concept-enabling levels. This validation has been devised to support the concept that was formulated earlier, and is consistent with the requirements of potential system applications. The validation is relatively “low-fidelity” compared to the eventual system – it could be composed of ad hoc discrete components in a laboratory; for example, an application tested with simulated inputs.

Level 5 – Software application tested and validated in a laboratory context.

The fidelity of the software application testing has not been demonstrated. The software application must be integrated with reasonably realistic supporting elements so that the total application (component level, sub-system level, or system level) can be tested in a “simulated” or somewhat relevant context. At this level, issues such as scalability, load testing, and security are addressed when applicable.

Level 6 – Software application demonstrated in a relevant context.

The fidelity of the software application testing has not been demonstrated. The software application must be integrated with existing elements and interfaces so that the total application (component level, sub-system level, or system level) can be tested and validated in a relevant context. At this level, issues such as number
of users and operational scenarios, as well as load testing and security are addressed if applicable.

Level 7 – Software application tested and validated in a relevant context. The software application testing meets the requirements of the application that apply to the software when it is to be delivered or installed. The software application has been tested in the lab so that the application can be validated as if the software were delivered for use in another context. At this level, all issues have been resolved regarding security and operational scenarios.

Level 8 – Software application “qualified” through test and demonstration (meets requirements) and successfully delivered. The software has passed testing and meets all requirements of the software, with the additional testing of the software delivery and installation for various applications.

Level 9 – Actual software application tested and validated through successful use of application output. Demonstrable that for any application of the software, testing shows the software meets all defined requirements.
10. Standards


Abstract: A common framework for extending the software life cycle processes of 12207.0-1996 to include the systematic practice of software reuse is provided. This standard specifies the processes, activities, and tasks to be applied during each phase of the software life cycle to enable a software product to be constructed from reusable assets. It also specifies the processes, activities, and tasks to enable the identification, construction, maintenance, and management of assets supplied.

**IEEE/EIA 12207.0**, "Standard for Information Technology – Software Life Cycle Processes", is a standard that establishes a common framework for software life cycle process.

This standard officially replaced MIL-STD-498 for the development of DoD software systems in May 1998.

This standard defines a comprehensive set of processes that cover the entire life-cycle of a software system—from the time a concept is made to the retirement of the software.
11. Myths

Myth #1: Software Reuse is a Technical Problem
Myth #2: Special Tools are Needed for Software Reuse
Myth #3: Reusing Code Results in Huge Increases in Productivity
Myth #4: Artificial Intelligence Will Solve the Reuse Problem
Myth #5: The Japanese Have Solved the Reuse Problem
Myth #6: Ada has Solved the Reuse Problem
Myth #7: Designing Software from Reusable Parts is like Designing Hardware using Integrated Circuits
Myth #8: Reused Software is the Same as Reusable software
Myth #9: Software Reuse Will Just Happen [13]

11. Theory

This mathematical language is based on what this theory regards as the two fundamental principles of intelligence:

Maximization of Transfer

Any agent is regarded as displaying intelligence and insight when it is able to transfer actions used in previous situations to new situations.
In fact, the agent must maximize the transfer of parts of generative sequences onto other parts of generative sequences.
Maximization of Recoverability

Any intelligent agent must be able to infer the causes of its own current state, in order to identify why it failed or succeeded, and thereby edit its behavior. Notice that this is part of a still larger problem, which the theory calls the problem of recoverability: Given the present state of an object, recover the sequence of operations which generated that current state.

Mathematical Theory of Transfer

To describe this Mathematical Theory of Intelligence, we will first describe the Mathematical Theory of Transfer. This begins by giving a statement of the Principle of the Maximization of Transfer, which the reader will recall, is basic to the above definition of intelligence.

MAXIMIZATION OF TRANSFER

In the generative sequence defining an object, make one part of the generative sequence a transfer of another part of the generative sequence, whenever possible. Therefore the maximization of transfer gives a maximization of reuse of the generative operations that define the object.

MODULARIZATION

The maximization of transfer of the operations that generate the object has the effect of modularizing the object into a hierarchy of generative components that are maximally reused within the generation of the object.
INSIDE-TO-OUTSIDE MAXIMIZATION OF REUSE

Representation of the object as generated by operations that were maximally reused within the generation of the object, makes the object itself maximally reusable elsewhere. This is because reuse within the object achieves most of the reuse that is needed when the entire object has to be reused.

New Foundations to Geometry give a mathematical theory of transfer and a mathematical theory of recoverability. Furthermore, the foundations combine these two mathematical theories, and this leads to a Mathematical Theory of Intelligence. This Mathematical Theory of Intelligence structures objects in such a way that they become maximally reusable, interoperable, and archival.

The mathematical theory of intelligence that reusability of an object is maximized if the object itself is defined as having been produced by maximizing reuse of the operations that were used to produce it. This is because reuse within the object achieves most of the reuse that is needed when the entire object has to be reused. Therefore the object must be represented generatively; and the generative operations used to represent it must be maximally reused in that representation.

12. Conclusion

Using a mixture of technical practices, collaboration, and pragmatism it is possible to slowly grow your reusable asset base. This article presented tips that I have used repeatedly as part of my everyday work in order to increase the odds of success. I am interested in hearing more about your experience with what has and hasn't worked when trying to foster effective software reuse.
13. References


