

*Reuse is like a savings account.
Before you collect any interest, you have to make a deposit,
and the more you put in, the greater the dividend.*
Attributed to Ted Biggerstaff,
1983 ITT Reuse workshop

*Reuse is something that is far easier to say than to do.
Doing it requires both good design and very good documentation.
Even when we see good design, which is still infrequently, we won't
see the components reused without good documentation.*
- D. L. Parnas, Software Aging,
16th International Conference Software Engineering, 1994

1	Software Reuse and Component Based Software Engineering	1
1.1	Software Reuse and its Relation to CBSE	2
1.1.1	Reusable Assets	5
1.2	Success Factors and Failure Factors in Reuse Management.....	7
1.2.1	Pitfalls and Obstacles to Reuse	7
1.2.2	Characteristics of Reusability.....	10
1.3	Economic models for Reuse.....	12
1.3.1	Calculating the break-even point (Gaffney and Durek model)	16
1.3.2	Estimating Development Effort using Reuse (Balda and Gustafson)	17
1.4	Recommended Further Reading	19
1.5	Assignments	20
1.6	Configuration Management.....	Error! Bookmark not defined.

1 Software Reuse and Component Based Software Engineering

An important motivation for many organisations for adopting CBSE as their software development paradigm is to reduce development cost. One of the main contributions that CBSE has to this objective is the reuse of software components in multiple systems. In this way a software component is developed only once, and can save out development effort multiple times. This approach towards saving effort does of course also work for other (intermediate) products that are made in the course of software development. An approach puts this idea central is called *reuse-based software engineering*. The following definition of reuse-based software engineering is given by Mili et. al. [MMYA02]:

Definition Software reuse is the process whereby an organisation defines a set of systematic operating procedures to specify, produce, classify, retrieve, and adapt software artefacts for the purpose of using them in its development activities.

This definition excludes ad-hoc reuse as a type of reuse because it is not governed by a well-defined process. The difficulty of ad-hoc reuse is that is not clear what can be expected of it in terms of possible contribution and costs.

Software reuse comes in two flavours: black box and white box reuse. Black-box reuse aims to integrate assets into a target system without modification of the original assets. In white-box reuse, assets may be modified before integration into target system. White box reuse is easier to realize, but yields a smaller benefit than black box reuse because more effort is required for understanding the asset and for redoing validation (e.g. testing) for the modified asset.

In this chapter, we look at CBSE from a reuse perspective. First, we discuss the similarities and differences between CBSE and reuse-based software engineering. Then, we discuss the factors that influence the success and failure of reuse-efforts. These factors are of economical and organisational nature, rather than of purely technical nature and will appeal more to readers with some exposure to industrial software engineering. At the start of a project, the project leader needs to decide whether to reuse software or not. Very often such decisions are based on intuitive arguments. In section 1.3 we present models for making quantitative assessments of the economic impact of reusing software. These models help in making more substantiated decisions about reusing of software.

1.1 Software Reuse and its Relation to CBSE

From a high-level business perspective, CBSE and Reuse-oriented software engineering ('reuse' for short) have the same goals: increasing productivity and quality. However, these paradigms differ with respect to the relative importance of these goals. Consequently the paradigms differ subtly in their approaches towards achieving these business goals. Because of the similarity of the business goals and commonalities of the approaches, there is often confusion about the precise meaning of the CBSE and reuse paradigms. In this section clarify the differences in emphasis between CBSE and reuse.

Reuse based approaches emphasize cost reduction as a means of increasing productivity. From an accounting perspective there are different ways of achieving this. One way is the amortization of the development and maintenance cost of assets over multiple projects. Another way is the avoidance of cost in later projects through the use of results of earlier projects. A less common objective of reuse that is not common to CBSE is that reuse initiatives aim at the dissemination of knowledge within an organisation.

As a result of the focus on reduction of cost, reuse approaches have the following characteristics:

- The subject of reuse is an '*asset*'. An asset can be any artefact that is used in the development and maintenance of a software system. Table 1 lists a set of assets that can be reused across software projects. Basically, any artefact that is recognized as having a particular value to other development efforts can be considered an asset.
- Reuse approaches have a strong emphasis on the organizational processes that need to be established for enabling reuse.

Possible Assets for Reuse		
Intermediate Artefact	Implemented Artefact	Project Management & Quality Assurance Artefacts
Requirements	(Sub)Systems	Process Models

Architectures	Frameworks, Components, Modules, Package	Planning Models
Designs	UML Models, Interfaces, Patterns	Cost Models
Algorithms	Libraries	Review and inspection forms (e.g. checklists)
Documentation (including templates)	Test Cases	Analysis models (e.g. performance, reliability)
	Classes, Procedures, Routines, Functions, Methods, Source code, Data	Design & Coding conventions

Table 1 A variety of assets can be reused across software development processes

CBSE approaches focus on improving productivity by reduction of development time and by improving flexibility of systems. This leads to different focuses in technical as well as organisational dimensions.

The most significant technical differences are the following: In the area of CBSE, techniques and mechanisms have been developed that enable the easy composition and upgrading of components in systems built from independently developed pieces of software. Reuse approaches are neutral toward the technical requirements on the system to be built. Both reuse and CBSE approaches depend on architectural conventions for compatibility of components.

The organizational scope of reuse differs from that of CBSE. In CBSE components may either be acquired from projects within the same organisation or components may be bought from some other company. In contrast, reuse approaches generally exclude the trading of components as this is not considered a means that can be exploited in order to achieve cost reduction. Thus the emphasis of reuse approaches is intra-organizational. A possible exception to this is the reuse of publicly available software such as open source software from the Internet.

Example To illustrate the differences between CBSE and reuse let's consider an example. Consider the pipe-and-filter mechanism of the UNIX operating system as a component system. In this context, filters can be considered as components and pipes as mechanisms for composition. This mechanism has a simple and standardized interaction style: a filter obtains input by reading from a stream of characters from *stdin* and produces output by writing a stream of characters to *stdout*. Composition of components is easily done at a UNIX command-line by listing filters in sequence, separated by a pipe-connector, denoted '|'. The following example, defines a pipe-and-filter system that composes three independent programs: *more* is a source that generates a stream of characters of the files that match a template, *grep* filters lines that match a keyword (in this example the word 'component') and *wc* counts words. Together this system counts the number of times the word component occurs in all text files in directory.

```
more *.txt | grep component | wc -l
```

Production of a filter-component consists of implementing a program that conforms to the rules of the pipe-and-filter model. So in this case, the production and composition of

components is easy. However, in order for a filter-component to be *reused* in another project, possibly in another part of an organisation, other conditions need to be met, such as the following:

- Development alignment:
 - o The component must be documented, tested according to target project standards
 - o A component is retrieved from a certain repository. After possible modifications, the asset is returned to the repository. Ideally this process is managed under some configuration management discipline.
- Architectural alignments
 - o Technically, the other system needs to be based on the pipe-and-filter architecture and be based on a compatible operating system.
 - o For the design, the functionality of the component needs to fit the functionality required in the target system

Reuse approaches attempt to achieve alignment of the above issues through establishing organizational standards and working procedures.

The following table summarizes the differences between CBSE and Reuse-based software engineering.

	CBSE	Reuse
<i>Basic concept</i>	Component designed for Composability	Asset designed for Generality
<i>Purpose</i>	Reduction of development time through technical facilities that enable the easy assembly and upgrading of systems out of independently developed pieces of software.	Reduction of cost through reuse of previously developed assets in the development of new systems
<i>Emphasis</i>	Technology for enabling the ease of assembly, upgrade and extension	Organizational processes for sharing of assets
<i>Typical organisational scope</i>	Inter-organization, Intra-organisational or both	Intra-organization

Tabel 2 Diffences in emphasis of CBSE and Reuse

From now on, will use the term ‘assets’ to denote a unit of reuse and ‘component’ to denoted a unit of composition.

Strictly speaking, it is possible to develop systems in a component-based style without doing reuse, and conversely, it is possible to reuse assets without developing in a component-based manner. Of course, this explanation serves to highlight the differences. Clearly, CBSE and reuse-based SE are highly compatible: commonly a component is considered an asset and as such an unit of reuse. Summarizing, CBSE enables reuse, but reuse does not require the use of components.

1.1.1 Reusable Assets

To give an idea of the broader scope of reuse-based software development compared to component-based development, we list a set of artefacts that can be considered reusable assets.

- Requirements
 - Typically requirements are the result of a labour intensive process involving analysts and domain experts. They codify important domain knowledge. Organisations that make multiple systems in the same domain can benefit from reusing requirements. In the transition from requirements to architecture, analysis models and feature models may be reused.
- Architectures
 - Architectures description the principal solutions for organising a type of system. Ideally, architecture descriptions include important design decisions and their rationale. Typically, architectures are the result of discussions of experienced architects that know very well how to capture the stable properties of a system. As such architectures codify very valuable knowledge about building systems in a specific domain, and may very well be among the most commonly reused assets in an organisation. The presence of design rationale is important for knowing whether some design decisions need to be reconsidered when an architecture is used in a different context.
- Design
 - A design is a moderately detailed specification of a subsystem or part of a subsystem. Design assets can be generic or specific to a domain or application. The most widespread generic design-level assets are probably *design patterns*. A design pattern is a solution to a commonly recurring problem. Currently patterns are mainly documented in books (e.g. [Bu+1996]) and internet repositories [WWW-Pat]. More and more support for the (re)use of patterns is being integrated in software development tools.
A typical example of a domain specific design level asset is a UML model. Such a UML model may be reused as the basis for implementations on different platforms or for subsequent designs.
- Implementation
 - Program code is the most common type of implementation artefact. However, also different types of data may be valuable assets.
 - Program code is machine processable text, typically in some programming language such as C or Java. Reusable program codes may take on many shapes and sizes, for example:
 - Executable, source, macro's, scripts for building/compiling, configuration files, template's, libraries, ...Source code may be packaged in different sizes, ranging from methods, to classes, to packages.
Program code embodies knowledge of encoding solution in implementation language. Reuse of program code is attractive because it immediately provides tangible results. Care should be taken that code not only has functional properties, but also non-functional properties, like performance. When the functionality of an implementation asset fits some context, it is not

automatically the case that also its non-functional properties match the new context. However, because non-functional properties are not directly visible from the code, assessing whether these aspects fits require additional effort.

- Data: Programs do not only exist of program code, but also may contain valuable data. In a route-navigation system for example, the key data is the set of roads and their GPS positions. A hospital information system maintains a list of possible medical diagnoses. A computer game may contain graphic designs (fonts, characters, scenery) that can be used in other games.

The use of commodity software such as operating systems or database system is typically not considered reuse. As a rule of thumb, if a component is not considered as part of the design of a system, it is not considered as being reuse. In CBSE, commodity software is sometimes considered as a component. This style of CBSE is called COTS-based development (Common Of The Shelf).

Code scavenging

When a programmer needs to implement certain functionality, he/she searches through existing programs for source code that is comparable to the one that is desired. When such code is found, it is adapted to its new context. Clearly this is a type of reuse as not all code is generated from scratch. While in structured approaches for reuse assets are defined a priori, code scavenging identifies assets for reuse on a *when needed*-basis. This way of working is not well studied empirically and little is know about its effectiveness. Nevertheless, with the increase of available source codes on the internet, the practice of code scavenging is widely observed in industry. In [Pou97] Poulin claims that such opportunistic reuse achieves only marginal savings as the modified software still needs to undergo testing, documentation and maintenance.

- Quality Assurance / Validation

Assets that may be reused across projects for quality assurance are:

- Review and inspection forms (e.g. checklists)
- Testing: Reuse test scenario's e.g. after a modification/extension of the system. Test scenario's capture knowledge about typical faults.
- Documentation templates

Look at CBSE from a reuse perspective, we can wonder what are the assets that are consider in CBSE approaches? Clearly, CBSE typically focuses on creation and use of implemented components. To enable this, a (product line) architecture needs to be defined. A key aspects of architectures are their interface definitions as these define the subsystems/components. Summarizing, the key assets in CBSE are: components, (product line) architectures and interfaces.

1.2 Success Factors and Failure Factors in Reuse Management

Reuse does not happen spontaneously. There are quite a lot of things that are likely to go wrong when they are not addressed explicitly. Fortunately, lessons about this have been accumulated over the years. The first set of lessons addresses conditions that need to be satisfied to ensure that there is a potential for reuse. These conditions help to determine whether reuse is possible in a given context.

- This software must belong to a focussed and narrow domain that has considerable commonality. The commonality in the domain should be identified through domain analysis and captured by a (product line) architecture.
- A large volume of software must be developed.
- The reuse program must be planned over a long period of time in order to amortize cost of new procedures and to populate the reuse repository

The next set of recommendation is aimed at getting reuse started and started in the right manner:

- Introduce reuse in an incremental manner rather than as big-bang. Select a few pilot projects that look promising for starting reuse.
- Establish development processes that support reuse. For instance new processes may be needed for repository management, quality assurance of assets, configuration management.
- Organize training for staff to make them familiar with the available assets and to explain how to do reuse.
- Create incentives for individual engineers for doing reuse

Once an organisation is doing reuse, the following advices help to continue doing reuse in a fruitful manner:

- Apply strict quality criteria for accepting assets in the reuse library. If customers are not satisfied with the quality of assets offered for reuse, they will quickly lose interest and prefer to develop assets themselves.
- Collect reuse metrics in order to manage reuse. A reuse program should be driven by business objectives. To find out whether the reuse program is continuing to make a contribution to these objectives, it is necessary to keep track of progress. To this end, different metrics about the reuse process should be collected. Reporting the progress of a reuse initiative to its participant helps give a sense of accomplishment and can be a good motivator to a team. Poulin [Pou97] and Lim [ref] provide detailed suggestions on which metrics to consider in a reuse program.
- Keep track of technological developments to avoid obsolescence. Setting up a reuse program easily takes one to three years. In the area of software engineering, quite a few things may have changed. It is important that the architecture and assets in the reuse program are evolved in accordance with relevant technological developments.

1.2.1 Pitfalls and Obstacles to Reuse

Despite the benefits promised, there are a number of obstacles that stand in the way of the adoption of reuse based software engineering. These obstacles are of the following types: managerial & organizational, economical, psychological, legal, and also technological. These categories are not strictly disjoint, but we follow this structure for ease of explanation. This section summarizes and supplements findings collected by Sametinger [Sam97].

Managerial and Organizational Obstacles

- Lack of management support
Management is not willing to invest in the introduction of reuse. This may be due to lack of money for the up-front cost or lack of human resources (no people are available for implementing the reuse program/managing reuse-based projects). Or lack of belief in the value that reuse adds to the business.
- Lack of organisational incentives

Producing software for reuse requires more effort than producing software for single use, hence is more expensive. Project managers are often reluctant to invest in a comprehensive reuse program because the benefits of making assets reusable often accrue to projects outside their realm of responsibility. Unless a business unit is compensated for the cost of providing reusable software, it will avoid making it. Either a special group needs to be assigned with managing a reuse-program, or an incentive scheme needs to be set up that rewards creating reusable software. For example individuals or groups could be rewarded if their software is reused by others.

- Organizational Inertia: it is difficult to change the way an organization works. The wider the scope of the reuse program, the more effort it will take to get all involved to support the reuse program. Overcoming this inertia requires coming up with a good business case for reuse. Methods for quantifying benefits of reuse are discussed in section 1.3.

Economical Obstacles

- Investment hurdle: introducing a reuse program first incurs cost (e.g. for the organizing the new development processes and its technical infrastructure, training, etc). The potential benefits are obtained only after some period of time passes. To compensate the extra efforts of the reuse groups, appropriate taxation or charge-back schemes need to be invented. No schemes that are generally applicable exist. Such schemes need to be tailored to the specific of an organisation. Different accounting schemes are needed for the start-up phases and for the phase of institutionalized reuse.

Psychological Obstacles

- A psychological affect that influences the adoption of new technology is the “Not Invented Here”-syndrome. This syndrome is fuelled by several factors:
 - Fear of the unknown: Adopting new methods and techniques introduces risks in the development. An organization needs to build up confidence in its estimates about effort and time needed for developing software in a reuse-based process. Essentially this is an issue of trust: If we don’t know how it was constructed, how can we gain confidence in the quality of the asset? Especially talented programmers are inclined to think that they can do a better job.
 - A decree to use assets developed elsewhere is by some engineers interpreted as a sign that the organisation does *not* want to use the software (s)he develops. The engineer then interprets this as a lack of confidence in his/her skills or even as a treat to their jobs.

- Engineers feel hindered in their creativity and independence if they are urged to reuse software they did not develop themselves. The possibility of doing creative work is an important factor that contributes to the work-satisfaction of employees.
- Lack of incentives
If contributing to the collection of reusable assets is considered to be part of every software engineer's task, then an incentive structure needs to be used that rewards the effort of individuals. Some benefits, like reduction of development effort reflect more on the project manager than on the individual engineer. An experience report [PD91] suggests that financial incentives for individual engineers may overcome this hurdle and also remedy the not invented here syndrome.

Political obstacles

- Groups or individuals that are in charge managing what is accepted as reusable asset obtain power over key architectural decisions. This power may be used to the advantage some groups and disadvantage of others.

Legal obstacles

- Liability: often components are sold without liability for the seller. This means that the buyer/consumer of the component takes on the risk of cost incurred from failure of the component.

Technical Obstacles

- Given a problem, it is difficult to find suitable assets that can potentially be used to solve this problem. In essence a problem description needs to be matched with a solution description. Generally, the problem has many different degrees of freedom that can be exploited in a solution. No methods have crystallized that can bridge this gap. Existing approaches are based on classifications, keywords, formal specifications. Related to this is the lack of methods for cataloguing and matching assets. In matching assets, perfect matches hardly ever occur and it is difficult to determine when a match is good enough. This also involves balancing the degree of mismatch indifferent characteristics of a component. For instance, it may be the case that a component may match well with requirement A, but not match well with requirement B. How does this compare with a component that matches these requirements in a converse manner?

Technical problems that are specific to CBSE will be important topics in the remainder of this book.

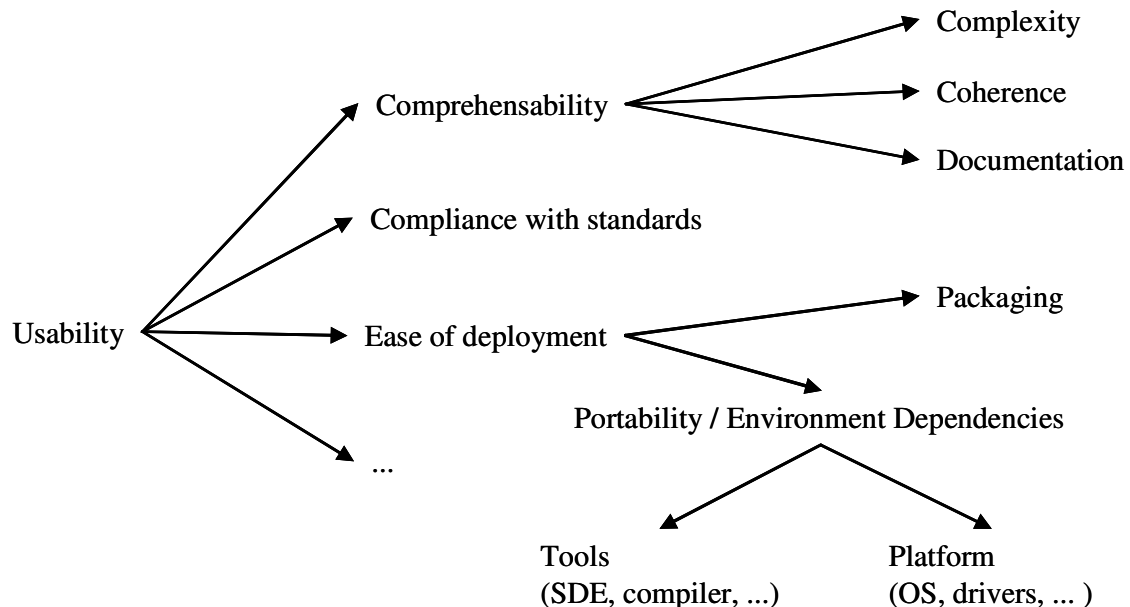
1.2.2 Characteristics of Reusability

The reusability of assets is different in different contexts. However, there are some characteristics that generally contribute to the reusability of assets. Although many of these characteristics apply to assets in general, we focus in this section on components as assets.

At a high level, we distinguish two aspects of reusability (following Mili et.at [MMYA02]).

$$\text{Reusability} = \text{Usability} + \text{Usefulness}$$

Usability is the degree to which an asset is ‘easy’ to use in the sense of the amount of effort that is needed to use an asset. Usability as such is independent of functionality of the component. Subcharacteristics of usability are shown in Figure 1 **Error! Reference source not found.** Comprehensibility is the effort needed to understand what a component does. Comprehensibility depends on the size and complexity of the component. Clearly, a component can also be faster understood if good documentation is supplied together with the component. A component that captures a single abstraction is easy to understand than a component that captures multiple abstractions or, when badly designed, mixes abstractions. Ease of deployment impacts the ease by which a component can be installed in an environment. The packaging of a component can be of large help in installing it - for instance by including all relevant files, configuration files and build procedures. In windows components may be packaged in an installer program. This program takes care of storing files in the appropriate locations and setting configuration parameters in a proper way. Conformance to standards may save effort in verification of a component. Portability is also a desirable characteristic as this means that little effort is needed for using a component on



different platforms.

Figure 1 Characteristics of Usability

Usefulness is the ‘frequency’ of suitability for use. How well does it fit in a context? So, usefulness depends on the functionality, the generality and quality of a component. The generality of a component can be high if it can be configured in many ways. This however has to be balanced with complexity of usability. There are many aspects of the quality of a component. Many, but not all, of the characteristics that are defined by the standard for software quality can also be applied to individual components, such as efficiency, reliability. Usefulness is also influenced by economic considerations like cost of purchasing and maintenance. **Error! Reference source not found.** show characteristics of usefulness.

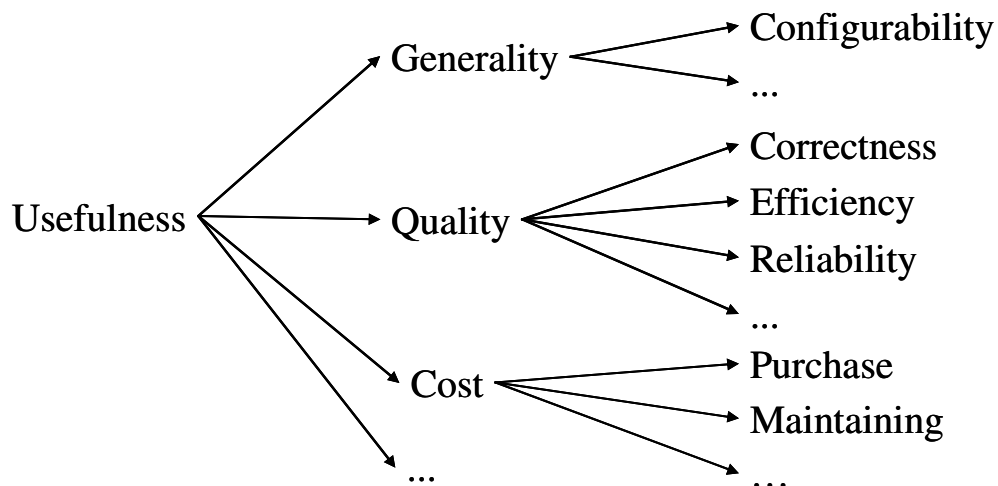


Figure 2 Characteristics of Usefulness

It is not the case that a more general component is more usable. Actually, more general components probably require more tailoring by setting parameters. The effort required for tailoring components impedes their use. The corollary is that components should strike a balance between being specific and being general.

The lists of characteristics of usability and usefulness is meant to aid the understanding of these concepts, but is not an exhaustive list.

1.2.2.1 Object-orientation and reuse

Object-orientation is regularly advocated as a technique for promoting reuse. However, empirical research does not support this claim. Rotherberger et. al. [RDKN03] performed a survey to investigate factors that contribute to success of reuse of code. The success of reuse was assessed in terms of reuse benefit (development and maintenance effort, cost, time-to-market), strategic impact (capitalize reuse by reaching new markets) and software quality (reduced number of errors). The conclusion of this research is that object technology by itself is insignificant in explaining the success of a reuse program. This should not be interpreted that object orientation does not help in realizing reuse, only that it is not critical for the success of a reuse effort. Also, object orientation brings along other benefits that contribute to an increase in productivity, only not necessarily through reuse.

Additional empirical research on the contribution of object-oriented programming to productivity is summarized by Endres [ER03]:

- Type checking leads to better productivity, reduced number of defects and shorter defect lifetimes.
- Object orientation enhances modularity which in turn reduced errors and enhances maintainability.

1.3 Economic models for Reuse

There are many informal arguments that make software reuse an appealing and economically viable idea. But reuse is not for free. Reuse of software incurs costs that would not have to be made if software was developed from scratch and not to be used again. In this section we will study some models and theories that have been developed to assess economics of software reuse. The main source for this section is the book *Measuring Software Reuse* by Poulin [Pou97]. In this section we will stick to terminology of software reuse, hence talk about assets rather than components. The majority of work on economics of reuse is on the reuse of source code.

From an economic perspective ‘black box reuse’ is considered as the only viable way of software reuse. In black box reuse, software assets are not modified internally; the only tailoring takes place via (configuration) parameters. Furthermore, Poulin distinguishes between

- *Internal reuse*: software developed and used repeatedly by the same people on the same application.
- *External reuse*: the use of software obtained from another organization or software application.

Internal reuse is considered ‘good engineering practice’. It is external reuse that we are interested in.

It is widely accepted that developing a reusable asset costs more than developing an equivalent custom developed asset. However, if this cost is compensated by a reduced cost of using an asset, then the overall balance may be positive. In order to be able to compute this balance, we should add up all the costs and subtract all revenues. The following list contains costs and benefits for both for the producers and consumers of assets.

Producer cost

Several factors contribute to the higher cost of developing reusable software:

- Assets must satisfy more (general) requirements; hence effort is needed for generalizing the functionality & interface. One way of increasing the generality of an asset is through adding extra customization / variability options.
- More effort is made to produce high quality documentation
- More thorough testing is performed to increase the quality of the asset

Producer benefits

- Revenue from selling assets
- Revenue from fees or royalties

Consumer cost

Additional activities that consumers need to perform in reuse-based development contribute to the cost-side of reuse:

- performing domain analysis
- performing Cost-Benefit analysis

- procurement procedure
 - the procurement procedure consists of
 - finding potentially suitable assets and the companies that sell them
 - assessing the quality of the assets
 - negotiating commercial terms for use of the assets (such as buying price, but also consider licensing and reselling fees)
- integrating / configuring parts
 - an important factor in the use of an asset is the effort needed to understand how a component works. This is also a key difference with traditional in-house development of software. Using the traditional approach the user of a component often has intimate knowledge of how a component works because it was designed by the same person or team.
 - wrapping assets
- use & maintenance contracts
- modification
 - testing modified parts!

Consumer benefits

Reduced cost of

- design,
- document,
- implement,
- design tests, unit test, document tests, implement & execute tests
- maintenance
- tools
- equipment

Potential additional benefits are

- improved sales due to increased quality or shorter time-to-market,
- delivering product early.

This is becoming quite a list, but is likely still not yet complete. There are some problems with using this list as the basis of an economic assessment of software reuse projects. Collecting data for measuring these factors is itself not economically feasible. Furthermore, there are so many confounding factors that it is doubtful that data for these factors can be determined with some reasonable accuracy. Thirdly, not all of the factors in the above list are relevant to a specific project. Therefore we need another way of assessing the economics of reuse. Let's look at what are some of the best practices for this problem.

The area of software engineering that deals with how reuse can best be measured is called 'reuse metrics'. The goals of reuse metrics are

- To objectively assess the amount of reuse
- To estimate the benefits of reuse
- To provide feedback on the amount of reuse to developers and managers.

One way to measure the benefits of reuse is to develop the same project once with and once without reuse. If the same crew is used, then there is a learning effect that influences the project's performance. If not the same crew is used, then the expertise of the different individuals on the team affects the project's performance. A way out of this dilemma is by

collecting statistics over a large number of projects as this is believed to even out effects of peculiarities of specific projects.

Unfortunately, there is no commonly accepted metric for assessing success of reuse. This makes it difficult to compare the empirical data that is published. An obvious metric to consider for measuring reuse is the percentage of software of a new system that is realized by reusing existing assets. However, this metric gives no indication of the contribution of reuse to the achievement of business goals (such as reduction of effort or cost).

~~$$\text{Reuse \%} = \frac{\text{Reused Software}}{\text{Total Software}} \times 100\%$$~~

A better metric that relates reuse to productivity is the notion of *Reuse Leverage for Productivity* (RL). The relative productivity of an organizational unit (project, department or company) that does not practice reuse is set at 1. If the productivity of the organization increases after the introduction of reuse processes, then the reuse leverage is greater than 1.

$$\text{Reuse Leverage for Productivity} = \frac{\text{Productivity with Reuse}}{\text{Productivity without Reuse}} \times 100\%$$

This notion of reuse leverage can only be measured after the introduction of reuse. Of course, it is desirable to be able to predict the effect of reuse. One way around this is to first try reuse on in one or a small set of pilot projects. Other ways are to work with estimates or with industry benchmarks.

From here on we assume that it is possible to obtain some high level data about reuse. Based on such statistics the following metrics can be determined:

Definition *Relative cost of reuse (RCR).*

Assume that the cost to develop a new module equals one unit of effort. The portion of this effort that it takes to reuse an equivalent module without modifications is called *relative cost of reuse*.

Based on experience reports, the average relative cost of reuse is estimated to be 20% (with variations reported between 3% and 40%). Additionally, some preliminary numbers are reported about relative cost of reuse with modifications. There are few empirical studies that report on this numbers and it is difficult to assess whether it is fair to compare the numbers as there are many factors that may differ. However, the following assumptions are commonly agreed upon for measuring the RCR:

- The programmer(s) understand(s) both the source and the target system
- The asset does not contain very complicated abstractions
- The asset consists of relatively small amounts of code
- The asset came with good documentation

<i>Percentage of component code that is modified</i>	<i>Relative cost of reuse</i>
0	0.20

<25%	0.40
>25%	0.90

The relative production of a reusable asset is defined by the relative cost of writing for reuse.

- **Definition** *Relative cost of writing for reuse (RCWR).*
Assume that the cost to develop a new module for one-time use equals one unit of effort. The portion of this effort that it takes to produce an equivalent reusable module is called *relative cost of writing for reuse*.

The most commonly reported values for the relative cost of writing for reuse are in the range of 1.5 - 2.2.

In developing for reuse, extra effort is spent on several tasks. The relative extra effort for different tasks Margono and Rhoads [MR91] are depicted in Figure 3. This data is based on a project that developed an Air Traffic Control. The system had to meet high targets for (real-time) response time, reliability and availability. The implemented system consisted of about 2 million lines of Ada code. Because the data is from a single project with fairly tough requirements, care must be taken to generalize these results.

In this figure the extra effort for reuse is set at 100%. It shows that of this 100% about 15% is spent on the architecting activities, about 60% on design activities and about 25% on testing. No extra effort was found for implementation activities. These numbers suggest that especially the design phase requires additional effort. This is attributed to the extra effort required for generalization of the functionality of the component. The same project reports that cost of integrating activities amounted to 30% of the total development cost. The Margono and Rhoads conclude that for a fair assessment of the success of reuse, also the quality of the resulting system needs to be taken into account.

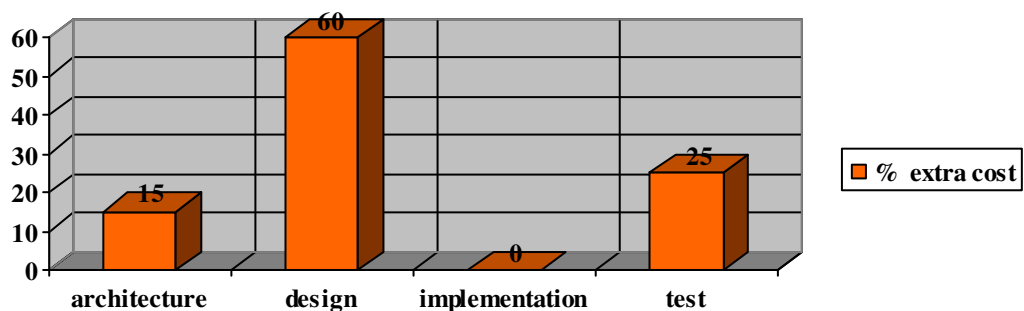


Figure 3 Estimated Distribution of Extra Reuse Effort over Development Tasks

The RCR and RCWR combine all factors that contribute to the benefits, respectively cost of reuse mentioned earlier in a single number. Although it is difficult to get exact data for the RCR and RCWR for a specific organisation, industry benchmarks can be used as a starting point. These benchmarks give an average and a range in which these values vary. If an organisation believes there they are doing a bit better or worse than average, they can make reasonable estimates for their situation based on these benchmarks. Consequently, using RCWR and RCR only an *approximate* analysis can be made about the economic effects of reuse.

In addition to affecting the effort, the reuse of software may also affect the schedule of development. The effect of the schedule is generally not proportional to the effect on the effort, as effort is not distributed uniformly over the tasks of a development process.

Another factor that affects the economics of reuse is the scale of a reuse program. The benefit of reuse increases as the scope of the reuse program increases coverage in terms of number of projects. This can be achieved by scaling the program across a wider scope of the organisation. **Error! Reference source not found.** The benefits of reusing scale with a factor of about 1.5 as a result of scoping reuse across multiple product lines in comparison to reuse within a single product.

1.3.1 Calculating the break-even point (Gaffney and Durek model)

Gaffney and Durek propose a model for making a cost-benefit analysis of reuse [GD89]. Their model assumes that the cost of the reuse program, including the additional cost to build reusable components is amortized across all future projects that will use the component. Their model further assumes a centrally maintained repository that must recover its cost by charging equal sums of money to the first n projects that use a component from the repository.

Gaffney and Durek define the cost of software development with reuse relative to the cost of software development with all new code. The relative cost factor of developing new code is denoted CF_{new} . The relative cost factor for reuse of code is denoted CF_{reuse} . The model further assumes some ratio between newly developed, R_{new} , code and reused code R_{reuse} , such that $R_{new} + R_{reuse} = 1$.

$$C = CF_{reuse} * R_{reuse} + CF_{new} * R_{new}$$

The cost factor for reused code is composed out of two parts:

- First, there is the relative cost of integrating reused code into a system. This is represented by a weight factor b . For reuse to be cost effective b must be < 1 . An optimistic estimate for the value of b is 0.08. This is reached if the requirements, design, and code that belong to the code are reused as well, such that only the testing phase must be done. Otherwise, if only the implementation is reused and requirements and design activities have to be performed for the reused implementation, then b is estimated to be 0.85.
- Secondly, each project has to bear a portion of the cost that is made for making reusable the software that it uses. The relative cost factor for this is denoted by E . This factor is assumed to be ≥ 1 . This cost is divided over the number of projects that reuse this software, denoted n .

Substituting $R_{new} = 1 - R_{reuse}$ and $CF_{new} = 1$, the equation for relative cost C becomes

$$\begin{aligned} & (b + E/n) * R_{reuse} + (1 - R_{reuse}) * 1 \\ = & (b + E/n - 1) * R_{reuse} + 1 \end{aligned}$$

where

b is the relative cost of integrating reused code ($0 < b < 1$)

R_{reuse} is the portion of reused code in the project ($0 \leq R_{reuse} \leq 1$)

E is the relative cost of creating a reusable component ($E \geq 1$)

n is the number of uses over which the reused code is to be amortized ($n > 0$)

If there is no reuse ($R_{reuse} = 0$), then the relative cost equals 1 ($C=1$). The model also implies that for a reusable component to pay off, the component must be reused at least two times.

The graph in Figure 4 shows the relative cost of a project when we use typical values of RCR = 0.2 and RCWR = 1.5 for b and E respectively. The break-even point is reached when $C=1$, hence $n = E / (1-B)$. When taking $b=RCR$ and $E=RCWR$, this gives $n = RCWR / (1-RCR)$.

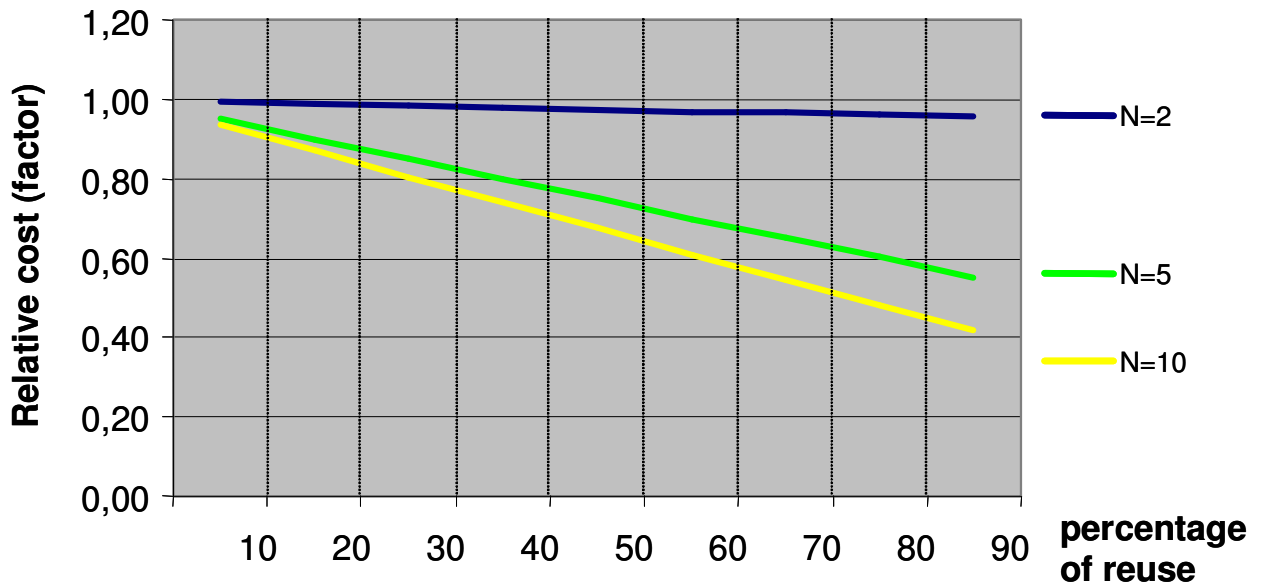


Figure 4 Relative cost C as a function of percentage of reuse in a project

1.3.2 Estimating Development Effort using Reuse (Balda and Gustafson)

How do you plan a project assuming you will be reusing software? Reuse-based software development requires dedicated models for estimating effort needed for executing a project.

A seminal work in estimating effort and cost of software projects is *Software Engineering Economics* by Barry Boehm [BB81]. In this book Boehm proposed the COCOMO model for measuring and predicting software cost of a project. This model is still widely used and numerous improvements to it have been suggested [BB00]. The COCOMO model has been tailored by Balda and Gustafson to take reuse-based development into account [BG89]. We next first have a look at the basic COCOMO model and then look at its specialization for reuse-based development.

The basic formulation of the COCOMO model, which provides a coarse estimation, is as follows:

$$E = \alpha * S^\beta$$

where

E = Effort in labour-months

S = (estimated) KLOC

α, β = complexity coefficients

Coefficient α is typically in the order of 2.4 ... 3.6 and exponent β is typically in the order of 1.05 ... 1.20 where higher values are associated with more complex systems.

Balda and Gustafson adapted the simple COCOMO model by distinguishing newly developed code that is specific to the project, newly developed code that is made for reuse, code that is reused and code that is modified. The model uses the variables S_1 through S_4 to represent these types of code. The adapted model is as follows:

$$E = \alpha_1 * S_1^\beta + \alpha_2 * S_2^\beta + \alpha_3 * S_3^\beta + \alpha_4 * S_4^\beta$$

where

- E = Effort in labour-months
- S_1 = KLOC for unique code developed
- S_2 = KLOC for code developed for reuse
- S_3 = KLOC from reused code
- S_4 = KLOC from modified code
- α_i, β = complexity coefficients

In the early COCOMO models, the schedule of a project is derived from the effort by assuming a Rayleigh distribution (see sidebar and Figure 5) of labour across the duration of a project. In 2000 the COCOMO model was updated for new developments in software engineering [BB00]. It now also supports other model for manpower build-up. Estimating effort of component reuse as part software product lines is discussed in [BB04].

The Raleigh model was originally proposed based on analysis of manpower data of hardware development projects. The COCOMO model predicts the area under the curve. The Raleigh curve then predicts how effort is spread out over time. Some critiques on the use of the Raleigh model are that it does not take into account the early phases of development (inception and requirements). Also, it does not capture steep increase in manpower as is regularly seen in software development project.

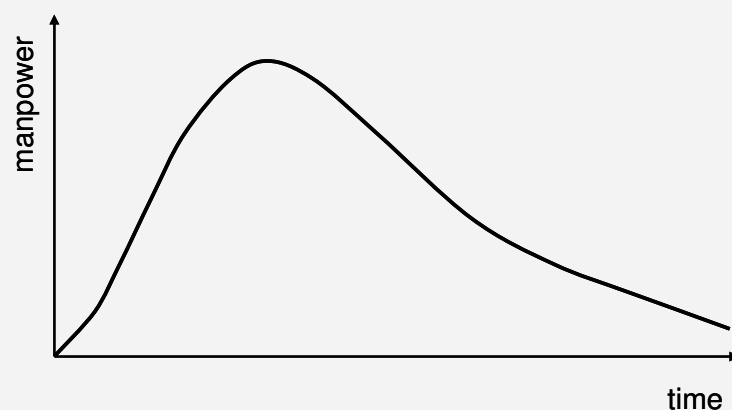


Figure 5 Raleigh Curve

Concluding remarks

CBSE can be considered an enabler for reuse. CBSE focuses on architectures, interfaces and components as reusable assets. Reuse based development has a wider scope than only the reuse of components.

Many studies confirm that software reuse may reduce effort and cost for development and maintenance. The results of these studies are difficult to compare as they assess the success of reuse in different manners (either by counting different units or by considering different criteria for success). Another complicating factor is that the introduction of reuse or CBSE is often combined with the introduction of other new tools and techniques. As a result the contribution of reuse is difficult to isolate. In assessing the success of reuse, keep track of the business goals that it was aimed to improve – reuse is not a goal on itself. However, there are many pitfalls in doing reuse and we have discussed a number of these.

A number of models for assessing the economics of reuse exist. Listing all factors that may contribute to costs and benefits becomes unwieldy. Instead, models that use fewer parameters are more practical. Cost models work best if you tune the parameters to a specific organisation.

What you should know

- Reuse is not ‘for free’; investment comes up front.
- Reuse does not happen spontaneously; a lot of things need to be organised in order to avoid pitfalls and make reuse work.
- Reuse is generally economically feasible if assets are used as ‘black box’
- You should know which factors influence the usability and usefulness of an asset
- How **not** to measure reuse

What you should be able to do

- You should be able to apply the models for the assessing economic perspectives of reuse.
- You should be able to apply the criteria for usability and usefulness in the design of assets.

1.4 Recommended Further Reading

- Software reuse anti-patterns, John Long, SIGSOFT Software Engineering Notes, Volume 26, Issue 4, pp. 68 – 76, July 2001.
- Confessions of a used program salesman: institutionalizing software reuse, Will Tracz, Addison-Wesley, 1995.
- Managing Software Productivity and Reuse, Barry W. Boehm, IEEE Computer, p. 111-113, September 1999.
- Managing Software Reuse, W. C. Lim, Prentice Hall, 1998. Jacobson, I, Griss, M. & Jonsson, P., *Software Reuse: Architecture, Process and Organization for Business Success*. Addison Wesley, 1997.
- Frakes, W. and Terry, C. 1996. Software reuse: metrics and models, *ACM Comput. Surv.* 28, 2 (Jun. 1996), p. 415-435.

1.5 Assignments

1. Use the model by Gaffney and Durek to derive an equation for the number of times of reuse of an asset. n , in order to reuse to be profitable.
2. Group assignment: this assignment can be executed in groups of two or more students. The assignment is aimed at experiencing the subtle things that where hiccups that hinder the smooth integration of independently developed components.

Execute the following steps:

- a. Design an architecture for a system. The size of the architecture depends on the size of the number of people that participate in the group. The architecture should be large enough such that each individual has responsibility for development of at least one component.
- b. Allocate to each person in the group at least one component.
- c. Agree on a deadline when everyone in the group has to have completed the implementation of his component.
- d. As a group, document the assumptions and design decisions that you think are necessary for the independent development of the components. Describe these in a document – the *base-line document*.
- e. Recommended candidate techniques for documenting the architecture are:
- f. UML – use a multiple-views description of components that describes at least the structural and dynamic view of each component.
- g. Architectural description languages
- h. Next, everyone in the group goes of to develop his own component(s).
- i. Each time a person runs into an issue that he needs to consult at least one other person in the group for in order to prevent mismatch, then this issue should be added to a list of missed alignment issues.
- j. After the deadline passes, copies of all sources of components need to be collected at a central point. These copies are not to be changed such that they can be used for reference later in the experiment.
- k. Now, an integration phase starts. The components that have been developed by individual team-members have to be assembled into a working system. During this process some issues may pop up that need to be resolved in order to get the system as a whole to work. Each team should collect and classify the issues they had to resolve. The following is an example classification of issues. You may use a more detailed or extended classification. For comparison it is desirable that all groups use the same classification.

A. Interface mismatch

- i. Syntactic (different names, order of arguments, argument-types)
- ii. Semantic
- iii. the servers (callee) expects input to satisfy a precondition that is not met by the client (caller).
- iv. the client (caller) expects outputs to satisfy a postcondition that is not delivered by the server (caller).

B. Protocol mismatch

- i. There is a mismatch between the order in which messages are expected/exchanged
 - C. User Interface mismatch
 - i. There is a mismatch in the interaction between components and the user interface
 - ii. Different components use different style of user-interfacing
 - D. Development mismatch
 - i. Different components require different (versions of) compilers
 - ii. Different components require different configurations of the compiler
 - E. Execution platform mismatch
 - i. Different components require different versions of an operating system / network
 - ii. Different components require different configurations of the operating system / network
- 1. Reflect on how the mismatches were introduced/how they have gone unnoticed. What type of measures could have been effective for preventing the (persistence of the) mismatch?
- 3. Propose an experiment that can be executed in an industrial setting from which you gain insight in the economic aspects of reuse, or more specifically component-based software development.
- 4. Propose a method by which you can estimate the effort needed for integrating a component.

Sidebar Empirical Software Engineering

Experimentation in software engineering is necessary but difficult. Common wisdom, intuition, speculation, and proofs of concept are not reliable sources of credible knowledge. V.R. Basili (1999).

The study into the economic aspects of reuse is a particular area of research within the domain of empirical software engineering. Empirical investigations rely on three methods: experiments, case studies and surveys. Empirical research is based on the scientific paradigm of observation, reflection/theory-building and experimentation.

One of the difficulties in performing software engineering experiments is that it is difficult to control the parameters. Experiments at universities are not representative of the conditions in industry, yet industry is continually changing and conditions change from one experiment to the next.

There is however an increasing recognition for the need to perform empirical studies into software engineering to increase our understanding of the factors that do and do not contribute to the improvement of software engineering in practice. The *Handbook on Software and Systems Engineering – Empirical Observations, Laws and Theories* by Endres and Rombach provide a comprehensive collection of results in this area. The methods and techniques required for performing experiments in software engineering in a scientifically sound manner are described in *Experimentation in Software Engineering: An Introduction* by Wohlin et. al. (Kluwer Academic publishers, 2000).