

# A Survey on Reuse: Research Fields and Challenges

(Term paper for DIF8901)

Li Jingyue

Jingyue@idi.ntnu.no

## Abstract

Software reuse means reusing the inputs, the processes, and the outputs of previous software development efforts. Although there have been many studies in this area, there still some fields to be explored. Four fields, that is, domain engineering, product line, and component based software engineering and COTS are discussed in this paper. Challenges in each field are generalized and summarized in this paper in order to have a whole picture of the state of the art of software reuse,

**Keywords:** Reuse, domain engineering, product line, component based software engineering, COTS

## 1. Introduction

Software reuse means reusing the inputs, the processes, and the outputs of previous software development efforts. Idea was originally due to Doug McIlroy in his paper “Mass Produced Software Components” (McIlroy 68). It is later named as a potential “silver bullet” by Fred Brooks (Brooks 87). And many researchers showed interest in this issue in the '80s and '90s.

Jones identified four types of reusable artifacts (Jones 84)

1. Data reuse, involving a standardization of data formats
2. Architectures reuse, which consists of standardizing a set of design and programming conventions dealing with the logical organization of software
3. (detailed) design reuse: for some common business application
4. Program reuse, which deals with reusing executable code.

Software reuse includes two interrelated parts: *reuse as* and *reuse with*. *Reuse as* means creating the asset for future reuse. *Reuse with* indicates using the reusable artifacts in the new system. In this paper, I distinguish four research fields among which domain engineering and product line focus on reuse as and component based software engineering and COTS base software engineering give guideline for reuse with.

There are a lot of research challenges in the area of software reuse. In this paper, I discuss these

challenges in line with three perspectives: reuse in general, reuse as and reuse with. The rest of the paper organizes as follows: section 2 of this paper summarizes challenges, success and failure factors in reuse in general; section 3 generalizes the challenges in reuse as field; section 4 point out the challenges in CBSE and COTS; section 4 summarizes these challenges, and section 5 gives the conclusion.

## 2. Reuse in general

### 2.1 Introduction

The Motivations of reuse are primarily economic: The potential of saving cost, time and effort of redundant work, increase productivity; decreasing time to market and improving systems quality by reusing both the artifact and the underlying engineering experience.

The challenges facing reuse are structural, organizational, managerial, and technical (Mili 95). The first mistake is that organizations treat reuse as technology-acquisition problems instead of a technology-transition problem. Just buying technology usually does not lead to extensive reuse. I believe that most important obstacles to reuse are economic and cultural, not technological (Dard 94). We discuss results and challenges in different categories of issues in the following sections.

### 2.2 Issues and Challenges

#### 2.2.1 Economical issues

Mili et al. have summarized several cost models having been proposed in the past for estimating, predicting, and analyzing the costs of software reuse (Mili 00). Some features that distinguish between these different cost models include:

- Investment Cycle. Most decision that arises in the practice of software reuse can be modeled as return on investment decisions. Four distinct investment cycles can be identified: the *corporate* investment cycle, the *domain engineering* investment cycle, the *application engineering* investment cycle and the *component engineering* investment cycle.
- Economic Function: five different functions that an investment assessment model may needs to consider: Net Present Value, Payback Value, Average Return on Book Value,

### Internal Rate of Return and Profitability Index.

- **Cost Factors:** For a given investment cycle and economic function, the set of cost factors that are taken into account in a cost model is an important feature of the model: this feature specifies what aspects of the reuse decision we want to consider. Doing reuse analysis well actually requires a much more extensive activity-based reuse model that can deal with all 10 artifacts and their mutual interactions. The 10 artifacts are architecture, cost estimates, project plans, requirements, designs, source code, user documents, human interfaces, data, and test cases (Jones 94).
- **Reuse Organization:** the organization structure has some impact on how costs are determined, charged, and accounted for.
- **Scope:** some models consider a short-term decision whereas others consider a long term investment cycle.
- **Hypotheses:** Some cost models neglect integration costs; some cost models assume that software development costs are linear in the size of the product; some cost models fails to take into account the discount rate of resources; some cost models ignore quality gains and only focus on productivity gains; some cost models ignore the inflation of code size that stems from software reuse.
- **Viewpoint.** Many parties are involved in a software reuse initiative: these include the corporate executives, the producer staff, the consumer staff, the library managers, and component providers.

Considering all the differences mentioned above, an integration cost model for software reuse is proposed by Mili in figure 1:

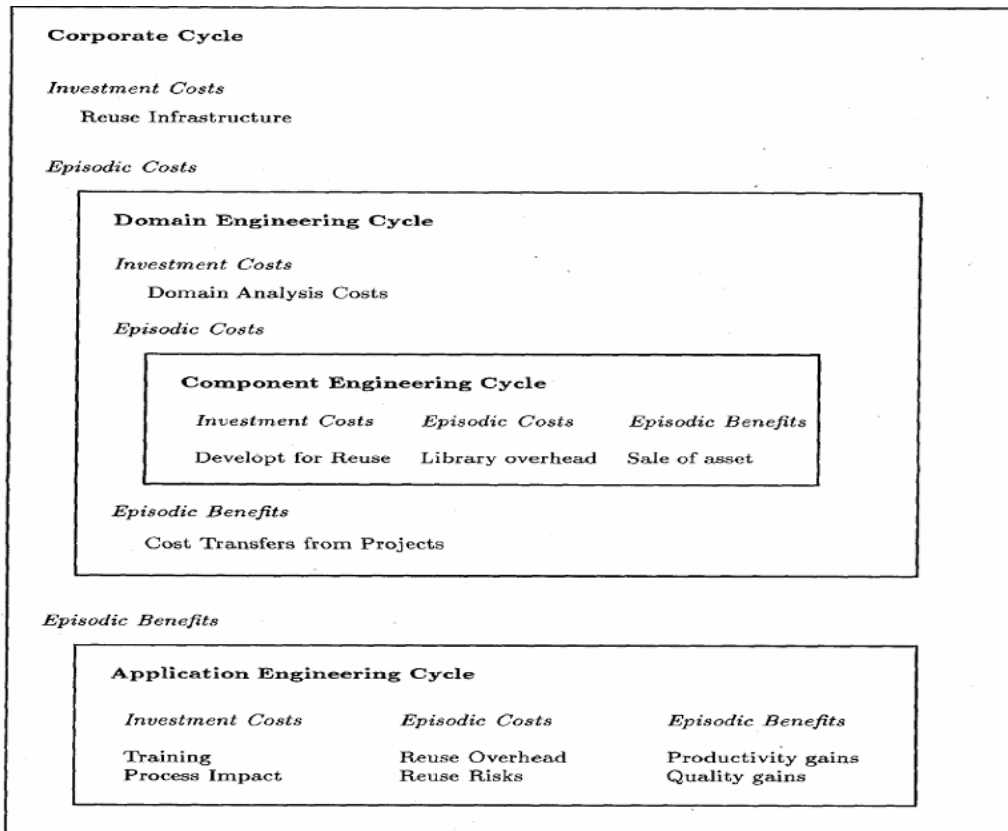


Figure 1: Software Reuse Cost Models

### 2.2.2 Organization and process issues

Morisio et al. get the conclusion from their research of several organization factors that affect the success and failure of reuse, such as: (Morisio 02)

1. Management commitment is the prerequisite of success reuse
2. Human factors must be addressed to sustain process change from bottom up. For example, four new roles will be introduced into the reuse process, such as lone producer, nested producer, pool producer and team producer (Fafchamps 94).
3. Introducing key reuse roles are not sufficient for successful reuse

Software process is assumed to be a key factor of software reuse. Although several reuse projects failed because no non-reuse-specific processes modified and no reuse-specific processes installed, software process maturity is useful but not sufficient factor in achieving success (Morisio 02).

### 2.2.3 Culture issues

Unlike other research field in computer science, cultural issues have either unique or misunderstood effects on software reuse (Dard 94):

- **Training**  
Training is often overlooked when reuse programs are planned because many people think reuse requires only minor variations on traditional software-development techniques. Misconceptions (Reuse – repository, Reuse-oo) cause a lot of project failures. (Morisio 02)
- **Incentives**  
Effective reuse requires not just passive acceptance of technology, but active engagement, in terms of the production and consumption of assets. Assets may turn out to be less robust than desirable if the producer simply follows standards rather than implementing the underlying principles of reusability
- **Measurement**  
Incorporating reuse measures into an organization’s measurement program can be a strong incentive to reuse within the organization.

#### **2.2.4 Technical Issues**

Although software component storage and retrieval is no longer an important issue in the practice of software reuse, the distinction between component retrieval and browsing is worth research (Mili 99).

- **Retrieval**  
A user submits a well-defined query to software library and expects the system to identify all the library assets that satisfy the query and exclude irrelevant assets. Retrieval is typically deployed in conjunction with top-down software design.
- **Browsing**  
The user has no predefined concepts of desirable functional properties but selects assets on the basis of general relevance guideline that might deal with application domain, computing platform, structural features, problem-solving knowledge, or other criteria. Browsing is typically deployed in conjunction with bottom-up software design.

## 3. Research field in reuse as

### 3.1 Domain Engineering

#### 3.1.1 Introduction

A domain can be defined by a set of problems or functions that applications in that domain can solve (Tracz 94). The term “domain” can be used in several associations (Schmid 00):

- Business area
- Collection of problems (problem domain)
- Collection of applications (solution domain)
- Area of knowledge with common terminology

To determine a domain, several approaches exist. The view can be focused on describing what is inside the domain; what is the boundary of the domain; or what is outside the domain. Domain can be divided into vertical and horizontal (Czarnecki 00). In *vertical domains*, software systems are classified according to the business area. Such systems are, for example, airline reservation systems, medical record systems, portfolio management systems, order processing systems, and inventory management systems. In *horizontal domains*, parts of a software system are classified according to their functionality. Examples are database systems, container libraries, workflow systems, GUI (graphical user interface) libraries, and numerical code libraries. When applying domain engineering to a vertical domain, the result could be reusable software that can be presented as a form of a reusable framework and components. In the case of horizontal systems, domain engineering may yield reusable components. The phases of domain engineering are domain analysis, domain design, and domain implementation. Domain analysis is first introduced by Neighbors to denote studying the problem domain of a family of application (Neighbors 80).

Domain analysis is associated with reuse, its purpose is to capture information involved with the domain to be reused in developing further applications of the same domain (Prieto 90). The output of domain analysis is domain model, the ingredients of domain model include domain scoping (domain definition, context analysis), commonality analysis, domain dictionary (domain lexicon), notations (concept modeling, concept representation), requirement engineering (feature modeling).

Domain design means designing the core architecture for a family of applications. The core architecture should also provide variability between applications. According to the feature models and commonality documents, it should also be selected which components or items (such as requirement) are provided in the core architecture and which items are implemented as variations in individual applications.

Domain implementation covers the implementation of the architecture, components, and tools

designed in the previous phase. This comprises, for example, writing documentation and implementing domain-specific languages and generators.

The purpose of domain engineering is to produce reusable assets that are implemented in this phase. Thus, the result of whole domain engineering phase comprises a series of assets which can be divided into two categories:

- Assets that abstracts a function, such as components,
- Assets that abstracts a structure, such as feature models, analysis and design models, architectures, patterns, frameworks, domain-specific language, production plans, and generators.

### **3.1.2 Issues and Challenges**

There are four kinds of outstanding issues that we feel require more research attention (Mili 99).

- How assets are represented.  
Assets that embody a function should be represented by a function that abstracts its most relevant functional properties, whereas an asset that embodies a structure should be represented in a way that highlights its relevant structural properties.
- How assets are matched.  
Matching an asset that embodies a structure is different with matching an asset that embodies a function.
- How assets are developed  
Developing for black-box reuse is primarily a specification issue and is determined by the generality of the asset's specification and the genericity of its implementation. Design for white-box reuse is primarily determined by such design issues as modularity, simplicity, and structuredness. We would expect design life cycle for reuse to depend a great deal on which reuse policy the asset is designed for.
- How assets are reused  
By definition, assets that embody a function might only be used for black-box reuse, where the user has no cognizance of the asset's internal structure. Assets that embody a structure may or may not have a functional associated with them; typically, they are geared toward white-box reuse.

## 3.2 Software Product Line

### 3.2.1 Introduction

In traditional software engineering, evolution of software occurs only in the maintenance phase in software system life cycle. A system is developed according to the initial customer requirements, and has normally limited possibilities to incorporate new requirements. As the system cannot be adapted to newly emerging customer needs, it is replaced with a subsequent version. In this scheme, software lifecycle phases and responsibilities are rather straightforward ones. There is a group taking care of development of a new system and another group is maintaining the old systems and giving customer support. Moreover, there are normally just few system generations varying from each other that have to be taken care of.

In the product-line approach, delivered software system are organized around commonalities that are shared by a family of products. The commonalities are implemented as software core assets. A common architecture is designed to support a certain amount of variability in the product level. Those functionalities neither supported by nor conflicting with the common architecture can be implemented in the product level. If functionalities created in product level are noticed to be usable also in other products, they can be generalized. They become new core assets. Thus, a product line is a continuously evolving organism.

### 3.2.2 Issues and Challenges

The general concerns and issues relating to product-line are:

- Organization and processes

“Organizational structure chosen for product line production must assign decision making responsibilities to make sure that evolution and core assets are managed to bring long-term benefit to the entire organization” (Clements 01). What is notably different in product-line organization from other organization is the product-line vision: the investment made now on reusable quality cores assets will be paid back in the future. For the smooth evolution, it is essential to define responsibilities between development of core assets and products, which can be seen in the product-line organization and in its roles. The essential difference from traditional software reuse is that all artifacts can be core assets, not just software components although they often are of main importance. Another noticeable feature for product-line organization is the existence product-line *champion* or product-line *sponsor* (or dedicated sponsor group). The role of champion is to maintain the overall organization knowledge of the product-line approach and keep the vision clear. No short goal decision should be made that would disturb the long term goals of the product-line.

Bosch presents four organization models (Bosch 00). The main principle in moving from an organization model to another is the size of the organization. The rule of thumb figures are offered for a number of personnel applicable with each of the models. Presented models are (recommended personnel size in parentheses):

1. Development department (up to around 30 software-related staff members).  
Products and product line assets are developed under a one unit and same staff participates in both core asset and product development projects.
2. Business unit (up to 100 software engineers in general case).  
Each business unit concentrates on developing a certain product or product group and units share a common asset base. Common projects might be initiated to revise the asset base or to create new assets. The maintenance of a certain set of assets is given to a certain business unit based on fact how much each unit is using the set and how probable is that the unit is the one that will further develop the assets.
3. Domain engineering unit (over 100 software engineers).  
Develops and maintains reusable assets that are used by product engineering units
4. Hierarchical domain engineering units (hundreds of software engineers/over 30 engineers in one domain engineering unit).  
In this model, domain engineering units are specialized on assets for certain product lines although there is a domain engineering unit that maintains common platform for all.

The main reason for moving from one model to another seems to be the avoidance of a situation when n-to-n communication will cause too much overhead. The purpose of re-organization is to break n-to-n communication net into n-to-one communication and force communication flow through certain communication points.

Clements and Northrop propose that reusable assets may include in addition to normal software components (Clements 01), for example:

1. Training specific to product line
2. Business case for the use of a product line for a set of products
3. Set of identified risks for building products for product line

Each core asset should have a process associated with it and the process specifies how asset will be used in the development of actual products.

- Economic

Many companies do not use a product line engineering approach when developing their product lines. More often than not, they either start from a single system, branching off new variants as the need arises and end up with completely independent code bases, or they start with the different variants as independent projects from the beginning.

In theory, the optimal product line development adoption scheme is to set up a completely new product line by developing a reuse infrastructure for the whole range of products right from the start. We often call this *big bang* approach. Another frequently used approach is *incremental approach*. With this approach, you develop assets to support the next few upcoming products, deliberately excluding highly uncertain potential products. Usually, constraints on available resources force this incremental approach, but it is generally useful even with unlimited resources because of the intrinsic uncertainty of future products and their requirement. Fig 2a shows the ideal big bang pattern, and figure 2b compares it with the corresponding patterns for the incremental approach (Schmid 02).

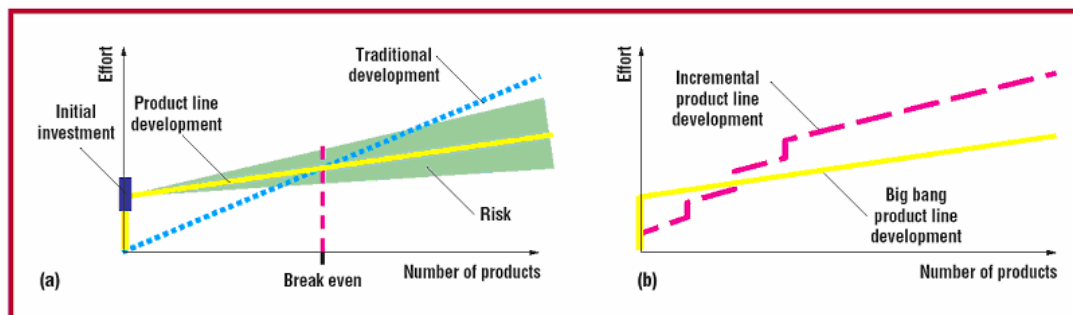


Figure 2. Product line investment curves: (a) the big bang approach, including risks; (b) the big bang versus incremental approach

Regardless of which approach you use, it is best to first distinguish several basic situations from which product line adoption can start. You can then link each situation to corresponding strategies (or adoption schemes) and connect a different pattern of investment and resulting benefits to each one.

1. Independent.  
The company starts a new product line without and predecessor products
2. Project-integrating  
Existing systems are already under development to address a new market. As part of product line development, the software engineers integrate the system so that they can derive them from the same reuse infrastructure.
3. Reengineering-driven  
Legacy systems already exist, but the engineering can not use them for product line development. Rather, they need to perform a nontrivial reengineering effort.
4. Leveraged  
The company sets up a new product line (to address a new market) based on a product line that is already in place.

Another issue that determines the economic part of product-line is evolution pattern. The main factor determining how a product line evolves is how much deviation the organization allows before reunifying the infrastructure. There are three situations:

1. Infrastructure-based evolution.  
New product requirements that might be reusable immediately lead to a generalization of the product line infrastructure. Thus, the organization can avoid the problem of multiple implementations of the same requirement.
2. Branch-and-unite  
The organization creates a new version branch for a new variant and then reunifies this branch with the original infrastructure after releasing the product.
3. Bulk  
At certain intervals, the organization reintegrates the product line infrastructure.

Table 1 summarizes the product line adoption and evolution patterns mention above.

<b>Table 1</b>			
<b>Product line adoption and evolution patterns</b>			
	<b>Situation type</b>	<b>Product line planning look-ahead</b>	<b>Approach</b>
<b>Adoption</b>	Independent	Broad portfolio of future systems	Big bang
	Project-integrating	Medium-size portfolio of future products	Incremental, by functional area or component
	Reengineering-driven	Broad portfolio of future products and legacy products	Incremental, by functional area or component, or big bang, by packaging existing legacy as a whole
	Leveraged	Broad portfolio of future products	Big bang
<b>Evolution</b>	Infrastructure-based	A small number of products	Incremental, by product
	Branch-and-unite	Single product	Incremental, by product
	Bulk	A small number of products (perhaps a market segment)	Incremental, by product group

Table 1. Product line adoption and evolution patterns

How an organization performs product line adoption and evolution strongly influences its product line's overall economic results. However, even if it selects a specific adoption approach, it still must decide which products to consider; when developing or extending the product line infrastructure, which technical areas to integrate next into its product line infrastructure, and which requirements reusable assets will directly support. Based on the types of decisions that must be made, three levels of decision making or scoping will be distinguished.

1. Product portfolio scoping  
Which products shall be part of the product line
2. Domain-based scoping  
Which technical areas (domains) provide good opportunitites for product line reuse
3. Reuse infrastruactrue scoping  
Which functionalities should the reuse infrastructure support.

Scoping techniques and their relation to product line adoption can be found in Table 2.

<b>Table 2</b>			
<b>Scoping techniques and their relation to product line adoption</b>			
<b>Mode of product line extension</b>	<b>Portfolio definition</b>	<b>Domain-potential analysis</b>	<b>Reuse infrastructure scoping</b>
Partial big bang and evolution by product group	Very important	Recommended, but mainly for risk analysis	Recommended to support architecture definition
By (single) product	Not necessary	Only needed if the extension requires restructuring	Only needed if the extension requires restructuring
By component or functional area	Should be performed	Key for identifying the next component for product line extension	Should be applied to support architecture definition

Table 2. Scoping techniques and their relation to product line adoption

- Variation management

The products belonging to the same product family have much in common. However, there are also variations among the products of the same family. Variation is provided according to different users of different design and implementation requirements. It is important to take variation into account in the early phases in designing product-line architecture and handling variation in the architecture level instead of code level (Cheong 98).

Variability can be divided into three categories called axes of variability (Meekel 98).

1. Feature variability  
Feature variability means variation in the definition and implementation of a specific feature or additional features.
2. Hardware platform variability  
It means variation in the type of microcontroller, memory, and devices that need to be supported.
3. Performance and attributes variability  
It means variation in the required performances such as number of back-to-back messages to be received, and in the attributes such as failure handling and concurrency support.

Variability can occur at different levels in the design (Svahnberg 00):

1. Product line level  
Variability at product line level defines how different products in the product line vary.
2. Product level  
Variability at product level defines the architecture and choice of components for a particular product.
3. Component level  
Variability at component level defines the component implementation to be selected into the

product. At this level, the set of framework implementation is selected. It is taken into account how to enable addition and use of several component implementations, and how to design the component interface to adapt to the addition of more concrete implementations.

4. Sub-component level

Variability at this level defines the features to form a component for a particular product.

5. Code level

There are different mechanisms to enable variation (Bosch 00) (Svahnberg 00):

1. Inheritance

Inheritance can be used if the component is implemented as a class in an object-oriented language. Through inheritance and late binding, each component can be specialized from its super-class for each specific context. Inheritance and extensions can be used at all levels of variability. However, they are especially important at class level (code level). They provide a way to divide the source code into several files.

2. Extensions

Extension means such kind of variation that the user selects one of different behavioral variants. There is typically the stable functionality, and each variable function is modeled as an independent entity. The user can select an existing variant entity or introduce a new one.

3. Configuration

Configuration allows variation where all variants are present at all variation points. The user may select appropriate files and set parameters to connect modules and components to each other. Configuration is mainly used in product-line level, product level, and component level. At the product-line level, it can be applied to select the components and the product-specific code. At the product level, the selected components are connected together. At the component level, the actual concrete implementations are selected to include into the product. Configuration may also be used at subcomponent level, if components have been designed as a collection of disjoint sub-components. In this case, configuration management can be used to select the specific parts of the components.

4. Parameterization, templates, and macros

These variability techniques are used when parameters or macro expressions can be introduced and later instantiated with the actual parameter or by expanding the macro. In template instantiation, components are configured with application-specific types. This variation can be applied in list or queue implementations for different element types.

5. Generation

A generator requires its input to be a specification written in some domain specific or component-specific language. The generator then translates this specification into a source-code-level component which can be attached to the product or application. For

example, graphical user interfaces can be generated from graphical or textual specifications. Generation is best suited for product level. At that level, code needs to be instrumented with product-specific code and to connect the components to each other.

#### 6. Compiler directives

Compiler directives (like `ifdef` in C++) can be used at compile-time to select between different implementations in the code. Different variation mechanisms can be applied at different variation levels.

The same three levels (product-line level, product level, and component level) may also apply compiler directives and parameterization. At the product-line level, compiler directives can be used to remove unnecessary product-specific code. At the product level, both of the techniques can be used to connect components to each other. However, they allow only a static way of connecting components. Compiler directives and parameterization at sub-component level are not recommended because they may lead to dead code and the complexity of the code. Instead of using inheritance, templates can usually be chosen. However, at sub-component level, templates are not always suitable. Usually more than one extension is allowed to be present in a system, and this is not technically possible when using templates.

#### ● Testing

Testing concerning product line covers the core asset software, the product-specific software and interactions between them. Concerning testing in product line, the testing software should be structured such that it supports reuse. Due to the demand of traceability, the test software should reflect the product-line architecture. For example, when two parts of the product-line architecture have a particular relationship, the test code for those parts should also be related. This reduces maintenance costs of the test software because it is easier to identify points requiring modification.

The product-line architecture can provide support for testing. This includes special test interface allowing a self-test functionality. The basic support for self-testing can be defined in the product-line architecture and applied by specific products according to their individual features.

#### 1. Testing in core asset development

There are three categories of reusable testing assets (Northrop 01):

- a) Documents such as test plan and test reports
- b) Test data sets
- c) Test software

The test documents can be organized hierarchically according to the relationships among the product in the product line. A skeletal test plan supports reuse by reflecting to the commonalities among the products. This also enables common features to be tested in the same way for every product. In product-line architecture, different products need different components. Thus, component can be variants for each other. The test plan for individual

component is divided into functional and structural test suites. Functional tests can be used for all variant components. Structural test must be modified for each different variation.

## 2. Testing in product development

In product-line architecture, products are generated from core assets. This makes integration testing an important activity. This kind of testing focuses on the interactions between components.

The specific issues concerns and issues relating to product-line evolution are:

- Recording of assumptions and design decisions

Software design process is based on multiple assumptions about the surrounding world in order to get software to fit into some mould that is formed based on these assumptions and constrains. As time goes by, some constrains will vanish, and a part of assumption will become invalid. That implies an inevitable change that has to happen in software. In that phase, it is invaluable to have knowledge about original assumptions, constraints and design decision. Impacts of change are impossible to estimate without knowing both the set of original assumptions and constrains and the current situation. In the product-line point of view, design change will be easier if one can concentrate on the assumption that is most probable to change.

Solutions for recording of design decisions are architecture description language (ADL) and design decision tree (DDT) (Karhinen 98).

- Configuration management

The approach to following configuration management issues are version management, temporary variation, permanent variation, build support and distributed development(Ommering 01). One typical approach relating product line configuration management is KobrA approach (Atkinson 02).

- Visualize and trace evolution

Product-line engineering requires more than tracking of software artifacts and their version histories. One must be able to trace evolution of non-software artifacts, different variants of software artifacts and the owner of the core assets. Configuration management tools can be used to do some work, but still there should be visual formats to help the trace effort and offer medium for discussion. An important issue is dependency tracking between product-line artifacts.

## 4. Research field in reuse with

## 4.1 Component-based software Engineering

### 4.1.1 Introduction

CBSE is enabled by the maturity of development environments and programming languages (such as Visual Basic, C++, and JavaBeans) that support both the development of the components themselves and the development of applications using components. The following features and challenges characterize CBSE in contrast with traditional (custom) software development:

1. The CBSE process is an interface-centered process in which component interface play the major role in plugging components together.
2. The development process is a composition mechanism in which one assembles an application from components.
3. An application development from components relies heavily on the separation of concerns and functionalities of the individual components. This is required to decrease the interdependency between components and hence improve system maintainability and reliability.
4. When an application uses COTS, it is much more dependent on the vendor, his support, and his upgrade schedule.

### 4.1.2 Issues and Challenges

Crnkovic et al. lists out eight general challenges in the component based software domain (Crnkovic 02).

- Component models

Even though existing development models demonstrate powerful technologies, they have many ambiguous characteristics, they are incomplete, and they are difficult to use. The relation between system architecture and component models is not precisely defined.

- Component based software life cycle

The life cycle of component based software is becoming more complex because many phases are separated in unsynchronized activities. For example, the development of component may be completely independent of the development of systems using those components. The process of *engineering requirements* is much more complex because the possible candidate components usually lack one of more features that the system requires. In addition, even if some components are individually well suited to the system, it is not obvious that they function optimally in combination with others in the system. These constraints may require another approach in requirements engineering – an analysis of the feasibility of requirements

in relation to the components available and the consequent modification of requirements.

Moreover, many questions remain in the last phase of component-based software life cycles. Because component-based system includes components with independent life cycle, the problem of system evolution becomes significantly more complex. There are technical issues (can a system be updated technically by replacing the components?), administrative and organizational issues (Which components can be updated, which component should be or must be updated ?), legal issues ( who is responsible for a system failure, the producer of the system or the producer of the component ?)

- Composition predictability

Even if we assume that we can specify all the relevant attributes of components, we do not necessarily know how these attributes will determine the corresponding attributes of systems of which they may become part. The idea approach- to derive system attributes from component attributes – is still a subject of research. The question remains: Is such derivation at all possible? Should we not concentrate on the determination of the attributes of component composites?

- Trusted components and component certification

One way of classifying components is to certify them. In spite of the component belief that certification means absolute trustworthiness, it in fact merely provides the results of tests performed and a description of the environment in which the tests were performed. Although certification is a standard procedure in many domains, it has not yet been established in software in general especially not for software components.

- Component configurations

As soon as we begin to work with complex structures, problems involving structural configurations appear. For example, two compositions may include the same component. Will such a component be treated as two different entities or will the system accept the component as a single instance, common to both compositions? What happens if different versions of a component are incorporated in two compositions? Which version will be selected? What happens if the different versions are not compatible? The problem of the dynamic updating of components is already known, but their solutions are still the subject of research.

- Tools support

The purpose of software engineering is to provide practical solutions to practical problems. Component evaluation tools, component repositories and tools for managing the repositories, component test tools, and component based design tools, run-time analysis tools, and component configuration tools and so on.

- Dependable system and CBSE

The use of CBD in safety-critical domains, real-time systems and different process – control systems, in which reliability requirements are particularly rigorous, is particularly challenging. A major problem with CBD is the limited possibility of ensuring the quality and other nonfunctional attributes of the components and thus our inability to guarantee specific system attributes.

## 4.2 COTS-based system development

### 4.2.1 Issues and Challenges

- Should protective wrappers be used on COTS products to minimize the probability of adverse interaction?
- What risks are alleviated by using COTS products (such as cost and reliability), and what risks are amplified (such as future compatibility and reliability evaluation?)
- What is the difference between using COTS products in real-time and embedded systems from their use in other types of systems?
- What complexity is added to the system through the use of multiple COTS products, and does this complexity increase the risk of errors as the system evolves.

## 5. Discussion

From the issues we mentioned above, a lot of challenges in software reuse general are divided into different research fields. We made a summary of these issues and challenges in Table 3.

Reuse in General	Reuse as (domain engineering/product line)	Reuse with (CBSE/COTS)
------------------	--	---------------------------

<p><b>Organization and process issues</b></p> <ul style="list-style-type: none"> <li>● Management commitment is the prerequisite of success reuse</li> <li>● Human factors must be addressed to sustain process change from bottom up</li> <li>● Introducing key reuse roles are not sufficient for successful reuse</li> <li>● Software process maturity is useful but not sufficient factor in achieving success</li> <li>● But several reuse projects are failure because no non-reuse-specific processes modified and no reuse-specific processes installed. (Morisio 02)</li> </ul>	<ul style="list-style-type: none"> <li>● Decision making responsibility must be assigned.</li> <li>● Champion and sponsor roles assigned. (Clements 01).</li> <li>● Four organization models proposed. (Bosch 00)</li> <li>● Different core asset should have a process associated with it (Clements 01).</li> </ul>	<ul style="list-style-type: none"> <li>● Component based software life cycle is needed (Crnkovic 02).</li> <li>● Methods of Investigating and improving a COTS-Based Software Development Process are proposed (Morisio 00).</li> </ul>
<p><b>Culture issues:</b></p> <ul style="list-style-type: none"> <li>● The most important obstacles to reuse are economic and cultural, not technological.” A reuse program is a technology transition problem and should be market-driven. Cultural factors hinder technology transition and should be addressed by training, incentives, measurement.(Dard 94))</li> <li>● Programming language, experience, rewards, repository, and reuse measurement are not decisive factors, while reuse education is. (Frakes 96)</li> <li>● Misconceptions (Reuse – repository, Reuse-OO) cause a lot of project failure. (Morisio 02).</li> </ul>	<p>Not Mentioned</p>	<p>Not Mentioned</p>
<p><b>Economics issues:</b></p>		

<p>A lot of cost models have been proposed. An integrated Cost Model for software Reuse integrated them together. (Mili 00)</p>	<p>Product line adoption and evolution strategies will influence the investment of reuse. (Schmid 02)</p>	<p>Only COCOMO 2.0 is indicated as cost model for CBSE, but time-to-market and reuse organization is not considered. (Boehm 95)</p>
<p>Technology issues:</p> <ul style="list-style-type: none"> <li>● Component description</li> <li>● Distinct between component retrieval and browsing (Mili 99)</li> </ul>	<p>Domain Engineering:</p> <ul style="list-style-type: none"> <li>● Assets representation; assets matching; assets development; assets reuse (Mili 99).</li> </ul> <p>Product Line:</p> <ul style="list-style-type: none"> <li>● Variation management (Cheong 98)</li> <li>● Testing (Northrop 01).</li> <li>● Recording of assumptions and design decisions (Karhinen 98).</li> <li>● Configuration management (Ommering 01).</li> <li>● Visualize and trace evolution</li> </ul>	<p>CBSE:</p> <ul style="list-style-type: none"> <li>● Component models</li> <li>● Composition predictability</li> <li>● Trusted components and component certification</li> <li>● Component configurations</li> <li>● Dependable system and CBSE (Crnkovic 02).</li> </ul> <p>COTS:</p> <ul style="list-style-type: none"> <li>● Should protective wrappers be used on COTS products to minimize the probability of adverse interaction?</li> <li>● What risks are alleviated by using COTS products (such as cost and reliability), and what risks are amplified (such as future compatibility and reliability evaluation?)</li> <li>● What is the difference between using COTS products in real-time and embedded systems from their use in other types of systems?</li> <li>● What complexity is added to the system through the use of multiple COTS products, and does this complexity increase the risk of errors as the system evolves. (Mili 99)</li> </ul>

Table 3 Research challenges summary

## 6. Conclusion

The context of our future research is SPIKE (Software Process Improvement based on Knowledge Engineering) project, which includes many Norwegian software companies. As the companies involved in the SPIKE project are doing more or less research and practice on reuse, we summarize the challenges in software reuse to guide our research and give suggestions for their future work.

## References

- (Atkinson 02) Colin Atkinson, Joachim Bayer, Christian Bunse, Erik Kamsties, Oliver Laitenberger, Roland Laqua, Dirk Muthig, Barbara Paech, Jürgen Wüst, Jörg Zettel, "Component-based Product Line Engineering with UML", Addison-Wesley, 2002.
- (Boehm 95) B. Boehm, B. Clark, E. Horowitz, C. Westland, R. Madachy, and R. Selby. "Cost models for future software lifecycle processes: COCOMO 2.0", *Annals of Software Engineering*, Sep. 1995, pp57-94
- (Bosch 00) Jan Bosch, "Design and Use of Software Architectures: Adopting and evolving a product-line approach." Addison-Wesley, 2000.
- (Brooks 87) Frederick P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering", *Computer Magazine*; April 1987
- (Cheong 98) Yu Chye Cheong, Akkihebbal L. Ananda, and Stan Jarzabek, "Handling variant requirements in software architectures for product families" In Frank van der Linden, editor, *Development and Evolution of Software Architectures for Product Families*, Second International ESPRIT ARES Workshop, volume 1429 of *Lecture Notes in Computer Science*, pages 188–196. Springer, 1998.
- (Clements 01) Paul Clements, Linda Northrop, "Software Product Lines: Practices and Patterns." Addison-Wesley, 2001.
- (Crnkovic 02) Ivica Crnkovic, M. L. "Building reliable component-based software systems." Artech House 2002
- (Czarnecki 00) Krzysztof Czarnecki and Ulrich W. Eisenecker. "Generative Programming: Methods, Tools, and Applications." Addison-Wesley, 2000.
- (Dard 94) Dave Dard, Ed Comer, "Why do so many reuse programs fails", *IEEE Software*, Sep 1994, pp114-115
- (Fafchamps 94) Danielle Fafchamps, "Organizational Factors and Reuse", *IEEE Software*, Sep 1994, pp 31-41.
- (Frakes 96) W.B. Frakes and C.J. Fox, "Quality Improvement Using a Software Reuse Failures Model", *IEEE Transaction on Software Engineering*, Vol. 23, No. 4, pp 274-279, Apr. 1996
- (Glass 98) Robert L. Glass, "Reuse: What's Wrong With This Picture?", *IEEE Software*, Mar/Apr. 1998
- (Jones 84) T. Capers Jones: "Reusability in programming: A survey of the state of the art," *IEEE Trans. Software Engineering*, vol. 10, no. 5, pp 488-494, Sept. 1984
- (Jones 94) Capers Jones, "Economics of software reuse", *IEEE Computer*, July 1994
- (Karhinen 98) Anssi Karhinen, Juha Kuusela, "Structuring Design Decisions for Evolution",

Proceedings of Second International ESPRIT ARES Workshop, Las Palmas de Gran Canaria, Spain, February 1998. Springer-Verlag.

(Lawrence 96) Shari Lawrence Pfleeger, “Measuring Reuse: A Cautionary Tale”, IEEE Software, July 1996, pp 118-127

(McIlroy 68) M.D. McIlroy, “Mass-Produced Software Components”, Bell Telephone Laboratories, Inc., New Jersey, USA, NATO SCIENCE COMMITTEE, Garmisch, Germany, 7 to 11 October, 1968

(Meekel 98) Jacques Meekel, Thomas B. Horton, and Charlie Mellone, “Architecting for domain variability”, In Frank van der Linden, editor, Development and Evolution of Software Architectures for Product Families, Second International ESPRIT ARES Workshop, volume 1429 of Lecture Notes in Computer Science, pages 205–213. Springer, 1998.

(Mili 95) Hafedh Mili, Fatma Mili, and Ali Mili: “Reuse Software: Issues and Research Directions”, IEEE Trans. Software Engineering, Vol. 21, No. 6, June 1995

(Mili 99) Ali Mili, Sherif Yacoub, Edward Addy, Hafedh Mili, “Toward an Engineering Discipline of Software Reuse”, IEEE Software, Sep/Oct, 1999

(Mili 00) Ali Mili, S. Fowler Chmiel, R. Gottumukkala, L. Zhang, “An Integrated Cost Model for Software Reuse”, Proceeding of International Conference of Software Engineering, Limerick Ireland, 2000.

(Morisio 00) M. Morisio, C.B. Seaman, A. T. Parra, V.R. Basili, S.E. Kraft, S.E. Condon, “Investigating and improving a COTS-Based Software Development Process”, Proc. 22<sup>nd</sup> International Conference on Software Engineering (ICSE 2000), p 31-40

(Morisio 02) Maurizio Morisio, Michel Ezran, Colin Tully, “Success and Failure Factors in Software Reuse”, IEEE Transaction on Software Engineering, Vol. 28, No.4, April 2002. pp 340-357

(Moore 01) Melody M. Moore, “Software Reuse: Silver Bullet?”, IEEE Software, Sep/Oct 2001

(Neighbors 80) James M. Neighbors, “Software Construction Using Components.” PhD thesis, Department of Information and Computer Science, University of California, Irvine, 1980.

(Northrop 01) Linda Northrop, “A Framework for Software Product Line Practice — Version 3.0”, Software Engineering Institute, Carnegie-Mellon University, January 2001. Available from <http://www.sei.cmu.edu/plp/framework.html>

(Ommering 01) Rob Van Ommering, “Configuration Management in Component Based Product Populations”, In Tenth International Workshop on Software Configuration Management, 2001. Available from <http://www.ics.uci.edu/~andre/scm10/papers/ommering.pdf>

(Prieto 90) Rubén Prieto-Díaz, “Domain analysis: an introduction.” Software Engineering Notes, 15(2):47–54, 1990.

(Reifer 97) Donald J. Reifer, “Practical Software Reuse”, John Wiley & Sons, 1997

(Schmid 00) Klaus Schmid, “Scoping software product lines”, In Patrick Donohoe, editor, Software Product Lines, Experience and Research Directions, pages 513–532. Kluwer Academic Publisher, 2000.

(Tracz 94) Will Tracz. “Domain-specific software architecture (DSSA) frequently asked questions (FAQ).” Software Engineering Notes, 19(2):52–56, 1994.

(Schmid 02) Klaus Schmid, Martin Verlage, “The Economic Impact of Product Line Adoption and Evolution”, IEEE Software, July/August 2002, pages 50-57

(Svahnberg 00) Mikael Svahnberg and Jan Bosch, “Issues concerning variability in software product lines”, In Frank van der Linden, editor, Software Architectures for Product Families, International

Workshop IW-SAPF-3, volume 1951 of Lecture Notes in Computer Science, pages 146–157 , Springer, 2000.