

# On the Purpose of Software Reuse

– A Review of Software Reuse Techniques (DT8100 course essay)

**Hong Guo**

guohong@idi.ntnu.no

## **Abstract**

*As modern software grows more and more complex, various techniques are invented or introduced to this field to increase productivity and reliability of software development. Software reuse plays an important role with respect to this goal. This essay tries to summarize popular techniques in software reuse field. Basing on a set of terminologies, we examine the nature of the techniques by clarifying the form and kernel logic of the reused assets, then identifying involved actors and activities. We believe this helps on both understanding the techniques and using or developing the techniques.*

## **1. Background**

Software reuse has been practiced since programming began.[1] Software reuse's purpose is to improve software productivity and quality by utilizing existed and proven software pieces when develop the new software. It is of more and more interests because people always want to build systems that are bigger and more complex, but still reliable while costs even shorter time.

There have been various discussions in literature about software reuse from both the technical perspectives and the organizational perspectives, from mechanisms to methodologies, from programming language to tools. Those divergent perspectives at times blur software reuse outlines. What is the difference and what is the relationship between them? In this essay, we reviewed some popular techniques in this area, and tried to find out some common parts by observing in the same perspectives.

This essay will be organized as below. Section two introduces the terminologies and the theme with which we review the techniques. Section three reviews popularly used techniques in software reuse field. Section four concludes this essay.

## **2. Introduction**

### 2.1 Scope

We will emulate some popularly used concepts related to software reuse, which includes mechanisms, techniques, methodologies and etc. There are overlaps and collaborations between these concepts when they are used in both researching and industry. We will call them techniques for simple in this essay. We do not intend to make the enumeration exhausted, or intend to clarify their definitions. Instead, we aim to examine and understand their essence in a uniformed way. That is, keep what is expected to be reused in head, and try to identify who is involved and what kind of activities shall get benefits from the techniques. We believe this is helpful to not only understanding the technique, but also working to improve the technique and the application of the technique.

## 2.2 Terminologies

We will use the following unified terminologies in this essay:

### 2.2.1 Reuse, Reuse Asset, Reuse Activities, Reuse Technique

Reuse means software reuse, which means, utilizing useful part in existed software to construct new software.

Reuse Asset: The part extracted from existed software that will be used when constructs new software.

Reuse Activities involves all activities aiming to the development of the Reuse Assets, or to the usage of the Reuse Assets.

There is one or more Actors involved in one Reuse Activity.

Reuse Technique involves all related technique, which Reuse Activities can get benefits from, especially, includes the techniques to develop reuse assets, and the techniques to use reuse assets. One specific Reuse Technique can involve both kinds of techniques.

### 2.2.2 For Reuse and With Reuse:

'For reuse' and 'with reuse' are used in [2] to refer to different technical aspects of reuse. Here we change them a bit to indicate the category of reuse activities: 'For reuse' means the activity is about developing Reuse Assets, packaging the Reuse Assets, and everything that to ease the usage of Reuse Assets later. 'With reuse' means the activity is about using the Reuse Assets to develop new software.

### 2.2.3 Design Reuse and Code(implementation) Reuse:

This is to indicate the deployment format/form of Reuse Asset.

The reuse asset can be a piece of source code written in a specified programming language, or in binary organized with a platform specification. In such cases, the reuse asset is on hand to construct new software. However, there are other cases. Sometimes we may want to reuse some 'ideas'. We want to reuse these ideas in different context, in programs that are written in different languages, above different platforms. These 'ideas' are common in a higher abstraction level between the problem space to the solution space, thus we commonly express them with one modeling language or even less formally, with natural language.

Design Reuse has advantages over Code reuse. It can be applied in more contexts and so is more common. Also, it is applied earlier in the development process, and so can have a larger impact on a project.

But most design reuse is informal, and happens through using experienced developers. There is no standard design notation and there are no standard catalogs of designs to reuse. A single company can standardize, and some do, but this will not lead to industry-wide reuse.

In summary, design reuse is more abstract than code reuse, and thus can be used in wider contexts. But on the other hand, code reuse is more concrete, and makes it easier to construct new software in suitable contexts.

When code reuse is the case, the reuse assets are reused by black-box way (through well-defined interface, commonly in binary) or by white-box way (in source code).

### 2.2.4 Frame Reuse and Ware Reuse:

We use this to address the kernel logic of reuse asset.

It may be easier to understand if we use ‘framework’ and ‘component’. Different terms are used to avoid confusion because ‘framework’ and ‘component’ are all named as specific reuse techniques as I will review in next section.

It could be more accurate to call them ‘interface user’ and ‘interface implementer’. Here interface corresponds to the Reuse Asset. Interface user defines interfaces, and integrates interface implementer. But be aware that on the other hand, interface user itself may implements a higher level interface thus is an interface implementer to a higher level interface user.

‘Frame reuse’ will be used to indicate that the interfaces, the parts using the interfaces are what we want to reuse. While ‘Ware reuse’ indicates that the interfaces, the parts implementing the interfaces are what we want to reuse. Thus both ‘Frame’ and ‘Ware’ are relative to the interfaces.

### 2.3 Review Theme

We will then review popular reuse techniques by considering the following questions:

- What is the Reuse Assets when the technique is applied?
- What is the deployment format of the Reuse Assets (Design or Code), and what is the valuable kernel logic of the Reuse Assets (Frame or Ware).
- What are the main Reuse Activities involved in two reuse categories (For Reuse and With Reuse) individually.
- What kinds of actors may be involved in?

## 3. Techniques Review

### 3.1 Programming Language Mechanisms

The evolution of programming languages is tightly coupled with reuse in two important ways. First, programming languages have evolved to allow developers to use ever larger grained programming constructs, from ones and zeroes to assembly statements, subroutines, modules, classes, frameworks, etc. Second, programming languages have evolved to be closer to human language, more domain focused, and therefore easier to use.[1]

Programming language mechanisms are the basis of all other Reuse techniques, because when a new software is constructed, it is always implemented in one or more specific programming languages.

As indicates in [3], Encapsulation of procedures, macros and libraries has been exploited for many years to enhance the reusability of software. Object-oriented techniques achieve further reusability through the encapsulation of programs and data.

Here we will take Object-oriented programming languages for example, and review related mechanisms which Reuse could get benefits from.

With different benefits that can be gained in different mechanisms, the Reuse Assets are always in source code format or in binary (Code Reuse).

	Frame Reuse	Ware Reuse
Design Reuse		
Code Reuse	Functions, procedures, class,	

## Figure 3-1

### 3.1.1 Instantiation and Prototyping

Instantiation is perhaps the most basic object-oriented reusability mechanism.

Objects may either be statically or dynamically instantiated.

An alternative approach to instantiation is to use prototypical objects rather than object classes as the “template” from which new instances are forged. This approach is useful to avoid a proliferation of object classes in systems where objects evolve rapidly and display more differences than similarities.[3]

### 3.1.2 Inheritance

Class inheritance is to provide a simple and powerful mechanism for defining new classes that inherit properties from existing classes.[3]

By utilizing various forms of inheritance, we could reuse part or all of the class members, at build time or run time.

With single inheritance, a subclass may inherit instance variables and methods of a single parent class, possibly adding some methods and instance variables of its own.

A natural extension to simple inheritance is multiple-inheritance, which means, inheritance of a subclass from multiple parent classes.

An interesting variation on class inheritance is what we call partial inheritance. In this case we inherit some properties and suppress others.

Class inheritance is essentially a static form of inheritance: new classes inherit properties when they are defined rather than at run-time. We will use dynamic inheritance to refer mechanisms that permit objects to alter their behavior in the course of normal interactions between objects.

We can distinguish two fundamentally different forms of dynamic inheritance, which we will call part inheritance and scope inheritance. The key difference is that the former occurs when an object explicitly changes its behavior by accepting new parts from other objects, whereas the latter occurs indirectly through changes in the environment.

Both forms of dynamic inheritance are possible within systems based on prototypical objects. An object may have instance variables and methods, but it may also delegate certain messages to an acquaintance, called a prototypical object. Part inheritance occurs when an object replaces an acquaintance to which it delegates messages. Scope inheritance occurs when a prototype changes its behavior, implicitly affecting all the objects that delegate to it.

Dynamic sub-classing mechanism could also support these two forms of dynamic inheritance.

Below diagram approximately reveals the relationship of different inheritance mechanisms.

### 3.1.3 Object Composition

Object composition is an alternative to class inheritance.

Composition and aggregation are two kinds of whole-part associations. Here, new functionality is obtained by assembling or composing objects to get more complex functionality. The composed objects are required to have well-defined interfaces. This style of reuse is called black-box reuse. In composition the whole strongly owns its parts, implying that the lifetime of the ‘part’ is controlled by the ‘whole’. This control may be direct or transitive. In aggregation the coupling is looser. [4]

Comparing inheritance with composition we find that inheritance is only useful in limited contexts,

whilst, composition is useful in almost every context.

### 3.1.4 Polymorphism and Overloading

A polymorphic function is one that can be applied uniformly to a variety of objects. Class inheritance is closely related to polymorphism. But it is also possible to have support for polymorphism without class inheritance.

Polymorphism may or may not impose a run-time overhead depending on whether dynamic binding is permitted by the programming language.

Polymorphism enhances software reusability by making it possible to implement generic software that will work not only for a range of existing objects, but also for objects to be added later. [3]

### 3.1.5 Generic Class

While class inheritance achieves software reusability by factoring out common properties of classes in parent classes, generic object classes do so by partially describing a class and parameterizing the unknowns.

Depending on the nature of the parameters, it may or may not be possible to compile generic classes before the parameters are bound. [3]

### 3.1.6 Virtual Class Attributes

In most object-oriented languages, the attributes of an object may be references to objects and (virtual) procedures. In Simula and BETA it is also possible to have class attributes.

In BETA a class may also have virtual class attributes. This makes it possible to defer part of the specification of a class attribute to a subclass. In this sense virtual classes are analogous to virtual procedures. Virtual classes are mainly interesting within strongly typed languages where they provide a mechanism for defining general parameterized classes such as set, vector and list. In this sense they provide an alternative to generics. [5]

The main Actors and Activities involved are identified in below table. Be noticed that utilizing programming language reuse mechanism itself does not directly produce reuse asset. Software coded with such programming language produces reusable codes. Thus there are two kinds of activities involved in Develop For Reuse category.

	Actors	Activities
Develop For Reuse	Language Designer, Tools Developer,	Define and support mechanisms for corresponding contexts
	Software Developer	Select suitable programming language, Design classes in an logically correct and extendable way
Develop With Reuse	Software Developer	Understand previous design, Identify reusable assets, Complement with new software part

**Figure 3-2**

## 3.2 Design patterns

Design patterns provide a means for capturing knowledge about problems and successful solutions in software

development, making it easier to reuse successful designs.[4]

Design patterns help you choose design alternatives that make a system reusable and avoid alternatives that compromise reusability. The required functionality of the software system is realized as patterns of interactions between objects. [6]

The use of patterns is essentially a form of reuse of well-established good ideas.[4] Thus, design patterns are expressed in natural language, graphical models, or some other informal way (Design Reuse).

Design patterns provide us with a software design vocabulary, which could help us to identify recurring problems. Further, design patterns help us to resolve them constructively based on proven solutions that provided. The design vocabulary also supports us in understanding the architecture of a given software system. This somehow, helps indirectly in potential reuses.

	Frame Reuse	Ware Reuse
Design Reuse	Design Patterns	
Code Reuse		

**Figure 3-3**

	Actors	Activities and Challenges
Develop For Reuse	Pattern Designer	Identify recurring problems in a specific context, Recommend solutions with benefits and cost analysis, Formalize the problem description and solutions.
Develop With Reuse	Software Developer	Identify patterns in problem domain, Evaluate and select recommended solution. Implement the solution.

**Figure 3-4**

### 3.3 Software architecture

There may be various definition for this: various structural models called architecture styles, that were commonly used in software and then examined quality attributes related to each style, or at a lower level of abstraction than style, [7]identified architectural patterns that commonly occur in various design problem domains [8].

Here we will use the definition in [9] under a wide context: providing a unifying or coherent formal structure. The author indicates some of the benefits we expect to gain from the emergence of software architecture as a major discipline are:

- architecture as the framework for satisfying requirements;
- architecture as the technical basis for design and as the managerial basis for cost estimation and process management;
- architecture as an effective basis for reuse; and
- architecture as the basis for dependency and consistency analysis.

The primary focus in architecture is the identification of important properties and relationships – constraints on the kinds of components that are necessary for the architecture, design, and implementation of a system. Given

this as the basis for use and reuse, the architect, designer, or implementer may be able to select the appropriate architectural element, design element, or implemented code to satisfy the specified constraints at the appropriate level.[9]

Architecture itself does not produce code commonly, however, produces design that designers and programmers need to respect. This design is also the basis on which other reusable assets could be used in this software.

	Frame Reuse	Ware Reuse
Design Reuse	Architecture design	Components dependency and etc
Code Reuse		

**Figure 3-5**

	Actor	Activities
Develop For Reuse	Architect	Construct an application framework Identify all important properties, relationships, constraints, dependencies and etc for both the framework and the components.
Develop With Reuse	Software Developer	Identify reusable assets satisfy the specified constraints. Construct software.

**Figure 3-6**

### 3.4 Components

As underlying technologies and business/organizational context within which applications are developed, deployed, and maintained matures, component based software engineering (CBSE) grows rapidly in nowadays. Components are complex design-level entities, that is, both abstractions and implementations.

Components are inseparable from architecture. Components should implement two interface types: a functional one that reflects the component's role in the system, and an extra-functional one that reflects the component model imposed by some underlying component framework.[8] Consider three different views of architecture:

- Runtime. This includes frameworks and models that provide runtime services for component-based systems.
- Design-time. This includes the application specific view of components, such as functional interfaces and component dependencies.
- Compose-time. This includes all the elements needed to assemble a system from components, including generators and other build-time services; a component framework may provide some of these services.[8]

There are actually two classes of concerns based on whether components are:

- Seen as off-the-shelf building blocks used to design and implement a component-based system. [8]  
This is what we mainly considered here also.

	Frame Reuse	Ware Reuse
Design Reuse	Architecture constraints	

Code Reuse		Components Implementation
------------	--	---------------------------

**Figure 3-7**

	Actor	Activities
Develop for reuse	Component Provider	Provide component implementation, Formalize specification with regarding to using the component.
Develop with reuse	Component User	Evaluate component and framework, Construct software with reusable component

**Figure 3-8**

- Used as a design philosophy independent from any concern for reusing existing components, or this approach stabilizes system design at the interface level, concentrates attention on collaborations among interfaces as the basis for understanding a system architecture, and enables reuse and replacement of implementations that conform to the interface specifications. Actually determine the develop with reuse environment, then develop components for reuse. [8]

	Frame Reuse	Ware Reuse
Design Reuse		
Code Reuse	Component Interfaces	

**Figure 3-9**

	Actor	Activities
Develop For Reuse	System Architect	Do architecture design and decide components' interfaces aiming to reuse components in a big extent.
Develop With Reuse	Programmer	Implement components according to the interface. Program the software.

**Figure 3-10**

### 3.5 Frameworks

A more recent and promising reuse model aiming at reusing large components and high-level designs is that of frameworks. Frameworks are a kind of domain-specific architecture (accept that a framework is ultimately an object-oriented design, while a domain-specific architecture might not be.).

Based on OOT, they are defined as 'semi-completed applications that can be specialized to produce custom applications'' Usually, a framework is made of a hierarchy of several related classes. Or, "a framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact." [4]

The framework designer should have a deep knowledge of the application domain and has to foresee future

requirements and domain evolutions.

In [10], the author explained common confusion about whether frameworks are large-scale patterns, or whether they are just another kind of component. The author proposes that,

- Framework provides the standard interfaces that enable existing components to be reused. Framework also provides some templates and reusable codes to make it easier to develop new components.
- Frameworks are a form of design reuse, similar to other techniques for reusing high-level design, such as templates or schemas.
- Frameworks are firmly in the middle of reuse techniques. They are more abstract and flexible than components, but more concrete and easier to reuse than a pure design (but less flexible and less likely to be applicable).
- Although they can be thought of as a more concrete form of a pattern, frameworks are more like techniques that reuse both design and code, such as application generators and templates. Patterns are illustrated by programs, but a framework is a program.

From above, we could identify framework actually consists of both design and implementation, on the other hand, includes both the component interaction, and part of the component implementation. Thus we can locate the reuse assets involved in a framework in our 2 dimension space like below:

	Frame Reuse	Ware Reuse
Design Reuse	Component Interaction Specification	Component Implementation Specification
Code Reuse	Frameworks Body	Components templates, reference implementation and etc

**Figure 3-11**

	Actor	Challenges
Develop for reuse	Framework Developer	Ensure the framework easy to learn, Provides standard way for existed components to be used, Make it easier to generate application in an efficient and elegant way for aimed domain, Make it easier to develop new components, by providing the specifications for new components and a template for implementing them.
Develop with reuse	Software Developer	Choose suitable framework, Develop new component utilizing given template according to spec, Customize framework itself according to problem context

**Figure 3-12**

### 3.6 Reuse/Software Library

A very common example of code reuse is the technique of using a software library. Many common operations, such as converting information among different well-known formats, accessing external storage, interfacing with external programs, or manipulating information (numbers, words, names, locations, dates, etc.) in common ways, are needed by many different programs. Authors of new programs can use the code in a software library to perform these tasks instead of writing their own.

A reuse library consists of a repository for storing reusable assets, a search interface that allows users to search for assets in the repository, a representation method for the assets, and facilities for change management and quality assessment.[1]

	Frame Reuse	Ware Reuse
Design Reuse		
Code Reuse	Reuse library	

**Figure 3-13**

	Actor	Activities
Develop For Reuse	Library Provider	Ensure reused components must be liable, available, findable, and understandable
Develop With Reuse	Software Developer	Find and evaluate usable components in an efficient and reliable way in the library Construct new software using assets from the library

**Figure 3-14**

### 3.7 Generative methods

An important approach to reuse and one tightly coupled to the domain engineering process is generative reuse. Generative reuse is done by encoding domain knowledge and relevant system building knowledge into a domain specific application generator. New systems in the domain are created by writing specifications for them in a domain specific specification language. The generator then translates the specification into code for the new system in a target language. The generation process can be completely automated, or may require manual intervention. [1]

An important part of making domain engineering repeatable is a clear mapping between the outputs of domain analysis and the inputs required to build application generators. Better integration of these two phases of domain engineering will mean much improved environments for domain engineering[1]

A similar concept with generative process may be in modeling field. Designer constructs a model according to modeling language (meta model), and model transformer transforms the model to code according to specific platform.

In such a process, the specification/model language may be quite formal, thus the generation/transformation process could be completely automated, and what is reused is the automatically generated code. They are like a kind of libraries, which are called by applications written in other programming languages. But these libraries may not provide specific interface.

On the other hand, the specification language can be quite informal, and much manual intervention is needed. Thus the reuse can be a mixture of design reuse and code reuse.

	Frame Reuse	Ware Reuse
Design Reuse	Generation logic	
Code Reuse	Generated code	

**Figure 3-15**

	Actor	Challenges
Develop for reuse	Specification language designer, Application Generator developer	Abstract domain specific language, Support to generate application in a target language according to specification written in the specification language
Develop with reuse	Software developer	Write specification according to the specification language. Generate application with the specification using generator. Manually intervene if needed.

**Figure 3-16**

### 3.8 COTS commercial off-the-shelf

COTS means a software product, supplied by a vendor, that has specific functionality as part of a system—a piece of prebuilt software that is integrated into the system and must be delivered with the system to provide operational functionality or to sustain maintenance efforts. [11]

COTS-based is the term used to indicate component- or package-based development—rapid configuration of software systems based on COTS packages, Government off-the-shelf (GOTS) packages, and some custom-built reusable packages.[11]

Here we can see the common concept of COTS includes all kinds of software product that could be reused, code or design, framework or component. It emphasizes the property from a business/marketing perspective instead of perspective.

	Frame Reuse	Ware Reuse
Design Reuse	Any means of commercial software product	
Code Reuse		

**Figure 3-17**

On the other hand, for any specific COTS product, the reuse assets commonly will aim to a specific domain, providing reuse in design level, or code level. Thus the examining on the reuse nature will be more meaningful.

## 4. Conclusion

By this essay, we try to find a unified way to examine various reuse techniques' nature, by finding out the reuse

assets, as well as its form and kernel logic. With this in head, it may be easier to find out involved actors and activities which are organized in two categories (Develop for reuse and Develop with reuse) in this essay.

In future, we will further identify whether there is any other dimension that is essential to understand reuse techniques or improve current ones. For instance, when the reuse happens- build time, construction time or run-time, may be another useful perspective to observe reuse. Besides that, frame use and ware use may be not quite perfect to examine some reuse techniques, thus may needs to be improved.

#### REFERENCES

1. W. Frakes, and K. Kang, "Software Reuse Research: Status and Future," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 31, no. 7, 2005, pp. 529.
2. G. Sindre, R. Conradi, and E. Karlsson, "The REBOOT approach to software reuse," *Journal of Systems and Software*, vol. 30, no. 3, 1995, pp. 201-212.
3. O. Nierstrasz, "A survey of object-oriented concepts," *Object-Oriented Concepts, Databases and Applications*, 1989, pp. 3-22.
4. I. Deligiannis, M. Shepperd, S. Webster, and M. Roumeliotis, "A review of experimental investigations into object-oriented technology," *Empirical Software Engineering*, vol. 7, no. 3, 2002, pp. 193-231.
5. O. Madsen, and B. Moller-Pedersen, "Virtual classes: A powerful mechanism in object-oriented programming," ACM New York, NY, USA, pp. 397-406.
6. L. Capretz, and P. Lee, "Object-oriented design: guidelines and techniques," *Information and Software Technology*, vol. 35, no. 4, 1993, pp. 195-206.
7. F. Buschmann, *Pattern-oriented software architecture: a system of patterns*, Wiley, 2002.
8. A. Brown, and K. Wallnau, "The current state of CBSE," *IEEE software*, vol. 15, no. 5, 1998, pp. 37-46.
9. D. Perry, and A. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, 1992, pp. 40.
10. R. Johnson, "Frameworks=(components+ patterns)," 1997.
11. M. Morisio, C. Seaman, A. Parra, V. Basili, S. Kraft, and S. Condon, "Investigating and improving a COTS-based software development," ACM New York, NY, USA, pp. 32-41.