# A Conceptual Framework for Maximizing Software Reuse

**Mrs. Sadhana Ghalsasi***



## Abstract

Software reuse is seen as a key factor for companies interested in the improvements of quality, productivity, and time-to-market. However, without adequate processes, it is too difficult to obtain the desired benefits. Moreover, current reuse processes present lack of details in important activities and are most concentrated in specific stages of the reusable software development process. This paper presents empirical guidelines for maximizing the software reuse activity by designers in the context of object oriented design. This study focuses on the aspect of software reuse which focuses the interaction between some design processes, e.g. constructing a problem representation, identifying a reusable entity and designing the solution such that reuse will be maximized. Thus, this work presents a proposal for a practical and efficient reuse process.

**Keywords :** Software reuse, Framework, Object Oriented Design, Component

## 1. Objective

Objective of this research is to create a conceptual framework for maximizing software reuse.

## 2. Introduction

Software Reuse is defined as the process of building or assembling software applications and systems from previously developed software. The notion of building software from reused components is similar to building electronic circuits from prefabricated ICs. [1] In traditional monolithic application development approach identical (or at least similar) functionality gets replicated numerous times across applications in the enterprise. Thus software must be designed in such a way that its components can be reused in future for various applications. Systematic software reuse is a technique that is employed to address the need for improvement of software development quality and efficiency.

Reuse reduces development time, increases team productivity, increases quality and reduces redundancy. Thus maximizing reuse is regarded as one of the major area for improving software development productivity. Quality and productivity could be improved by reusing all forms of proven experience,

*Symbiosis Centre for Information Technology, SIU, Pune, India

including products and processes, as well as quality and productivity models. Productivity could increase by using existing experience, rather than creating everything from the beginning.

Through the years, several research works, including company reports, informal research and empirical studies have shown that an effective way to obtain the software reuse benefits is to adopt a reuse process [7]. However, the existing reuse processes present some gaps and are not complete. Even today, with the ideas of software product lines arising, there is still no clear consensus between the activities (inputs, outputs, artifacts) and the requirements that an effective reuse process must have [2]. Existing methods have been either not flexible enough to meet the needs of various industrial situations, or they have been too vague, not applicable without strong additional interpretation and support [3]. A flexible method that can be customized to support various enterprise situations with enough guidance and support is needed.

And the question that still remains is "How to design efficiently so that we maximize reuse?" Under this motivation, this research proposes a practical and effective framework for software reuse. This paper aims to present a model which will help a designer to design reusable components.

The model will provide a systematic way to identify and design software through a set of guidelines plan, specify, model, design, and implement components and applications in a problem domain. The goal is to develop a robust framework for code reuse, in conjunction with the industry, involving processes, methods, environment and tools.

The main contribution of this work will be the development of a systematic model in order to allow the development of reusable components. The model will be based on the analysis of the state-of-the-art in the area, including models for domain engineering and software product lines. The key aspect of the model is the combination of guidelines, reuse metrics, economic aspects, and a set of requirements for an effective software reuse. The results can assist organizations in maximizing code reuse to meet their reuse goals.

## 3. Methodology

The concept of software reuse was thoroughly understood by reviewing the existing literature. The various component development methodologies were studied and analyzed to find out the similarities and differences. It was found that the knowledge of domain plays an important role in making the software more reusable. So some factors were identified, which should be considered at various levels to make more reuse possible. This framework can be partially or totally implemented by the users to achieve maximum software reuse at that level.

## 4. Analysis of Problem

Code reuse allows us to design and implement something once and to use it over and over again in different contexts. This will realize large productivity gains, taking the advantage of best-in-class solutions, the consequent improved quality etc.

There are different levels of code reuse. For instance, source code copy is the lowest level of reuse. Procedural function libraries are a better form of reuse than source code copy, but not extensible. Class libraries are better form of reuse and they are extensible. However it requires lot of understanding before classes can be reused.

Moreover, it supports only white-box reuse; clients will be affected if internals of classes are changed. For example, in OOP language such as java or C++, derived classes are coupled to base class implementation. Changes in any of the base classes in the inheritance hierarchy would break derived classes. Furthermore, this level of reuse is language specific; no reuse across code in other languages.

Components support highest level of reuse because it allows various kinds of reuse including white-box, gray-box and black-box. White-box reuse means that the source code of a software component can be made available which can be studied, reused, adapted or modified. Black-box reuse is based on the principle of hiding information.

The interfaces specify the services a client may request from a component. The component provides implementation of the interface that the clients relay on. As long as the interfaces remain unchanged, components can be changed internally without affecting clients. Gray-box reuse is somewhere in between white-box reuse and black-box reuse.

We can say that Code reuse is possible by using Components. Proper identification and designing of components will facilitate code reuse maximization. Thus to analyze the problem of how to design to maximize code reuse, we must first understand how the Components can be classified.

All components are not created identically. They differ in complexity, scope, functionality level, set of abilities needed to use them and required infra-structure. To facilitate differentiation, we can divide components in three categories, as shown in Figure 3.1.

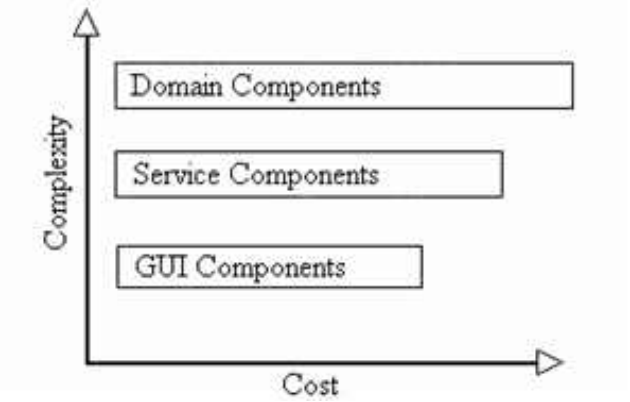


**Figure 1: Classification of Components**

### 4.1 Classification of Components

The following classification differentiates the components by cost and complexity [8].

**4.1.1 GUI Components**. They are the most common

components found in the marketplace because they have a low complexity level. Simple examples are: buttons, sliders, text fields and other widgets used in building user interfaces for applications. Reusing such pre built components is fairly easy and has a quick payback. Reusing components from this category may increase the productivity in about 40%.

**4.1.2 Service Components.** This category introduces more complexity to the components. They provide necessary common services for the applications. Also, they include database access, access to messaging and transaction services and system integration services. One common characteristic of service components is that all of them use additional infrastructure or systems to perform their functions. The deployment cost of this kind of component is high because they may integrate many different systems. Productivity increase, by the reuse of these components may reach 150%, however this value does not take deployment cost into consideration [8].

**4.1.3 Domain Components.** They are the most difficult kind of component for developing and reusing. For example, an application used by an insurance company may include Bill, Policy and Claim components. It is, unlikely, that they would find broad reuse outside that domain [8]. If a class diagram of the application is built, it would be possible to see that these kinds of components are the key abstractions of the application. They are normally monolithic, because they interact with other abstractions, as in Order and Bill system, for example. The use of domain components is more costly because they need a huge infrastructure for being deployed, but, productivity may increase 1000%.

**4.2 Component Based Development Methods**

Developing components requires a systematic approach the same as any other software development process. Therefore, using methods or processes for this kind of development eases the construction of more complex systems that connect all the "parts". Many Component Based Development methods have been suggested. Let us discuss about two important and well known methods, Catalysis and UML Components.

**4.2.1. Catalysis.** Catalysis is a research initiative developed at the University of Brighton by D'Souza and Wills [13]. This research had, as result, an integrated method to model and construct object oriented distributed systems. According to them, Catalysis covers all the component development phases, from its specification to its implementation. This method is based on a set of principles for software development that are: abstraction, precision, refinement, pluggable parts and reuse laws.

The abstraction principle guides the developer to search for the essentials aspects of the system, not getting close to the details that are not relevant for the context. According to Catalysis, The only way we can handle complexity is by avoiding it.

The precision principle aims to discover errors and modeling inconsistencies. Successive refinement between the phase transitions helps on getting more precise artifacts that are tended to be reused.

The pluggable parts principle supports the components reuse with the intention of building large systems just plugging the smaller reused parts.

Lastly, the main Catalysis reuse law is: "Do not reuse code without reusing the specification of that code."
The software development process of Catalysis is divided in three levels

Problem domain - specifying "what" the system does to solve the problem;

Component specification - where the system behavior is described;

Component internal design - where the internal implementation details are specified.

**4.2.2 UML.** UML Components [14] an UML extension, represents a process for component-based software specification. In this process, the systems are structured as four distinct layers: user interface, user communication, system services and business services. This structure represents the system architecture. Besides that, UML Components support the use of UML for modeling all the phases of the development of component-based systems, including activities as requirements definition, identification and description of components' interfaces, modeling and specification, and also implementation and composition.

The method uses a simple way of extending UML that is to use stereotypes on the entities. It means that the UML purpose, already well disseminated in the modeling area, can be followed by the software analyst. A concern with the method is to offer a solution without specifying the platform, so the method could be used in many platforms with many different technologies.

**4.3 Framework for maximizing software reuse**

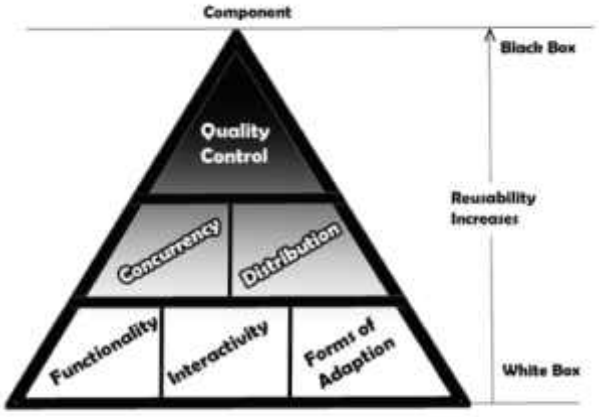


**Figure 2. Code Reuse Maximization Framework**

Diagram above represents the proposed framework for code reuse maximization. We have already seen the levels of code reuse i.e. White-Box, Gray-Box and Black-Box. In White-Box reuse you can customize the module to fit to a specific situation. Intrinsic knowledge about the code to be reused is required. While in black-box (components) reuse you just need to know

the specifications of the code (component) and you can directly use it.

"White-Box" reuse, or cut and paste, where source code is modified in order to use in another context, while useful, is not typically as beneficial as "Black-Box" reuse. Systematic reuse programs encourage reusing software without change because of the superior benefit they receive from black-box reuse throughout the life cycle. Thus we can say reusability increases from white-box to black-box reuse. The pyramid depicts the minimum and necessary attributes for code reuse. As we go higher, the reusability increases.

The lowest level attributes i.e. Functionality, Interactivity and Form of Adaption are necessary for black-box reuse but not sufficient. These will just provide white-box reuse. For example, a function written for sorting can be reused in different codes. It provides functionality of sorting, interactivity through parameters and we can change it as per our requirement (i.e. we can adapt it). But while doing so care must be taken that functionality is not changed while adapting the code.

The next higher level attributes i.e. Concurrency and Distribution are sufficient for creating a black-box component but not sufficient for black-box reuse. This level talks about considering the environment in which the code will be used and how communication between component and the environment will be enabled.

The highest level is Quality Control. To make a component reusable it is necessary that it is tested and verified. While reusing a component it should not add any defects and also that it fulfills all the requirements mentioned in its specification. Conformance to specifications must also be tested. Quality standards or certifications for components are still in research phase. Once they are standardized a designer will have to make their components adhere to these standards and certificates. But even without standards it is necessary that a component must be assured for its quality.

The lowest level depicts “White-Box” reuse, the middle one represents “Gray-Box” reuse and the highest level is “Black-Box” reuse. But to gain black-box reuse the component must fulfill the necessities of white-box and gray-box reuse too. Thus black-box reuse is possible only if the component has all the six attributes. Code reuse maximization is possible at highest level i.e. through components. The framework also depicts that many of these reusable components lie at the base as more space is available at the base, and very few lie at the apex. A component should always thrive to reach the top. So let us discuss the six attributes with reference to components in detail to maximize reuse.

**4.4 Components in the Framework**

**4.4.1 Functionality.** The component functionality is essential for its reuse in some contexts. Depending on the quantity of operations executed by the component, it may be too specific for a context or too general, or even incomplete. The concept of functionality is complemented with the concepts of applicability, generality and completeness.

The *applicability* of a component indicates its likelihood to be a reuse candidate in the range of software systems for which it was designed to be reused. The component’s applicability can be high, for a certain application domain, and low or zero for others. For example, a component built to control a car brake system has high applicability in the automobile domain but has zero applicability within the mobile phones domain;

The *generality* indicates if a component has more specific functionality or not.

For example, a component that sorts numbers has less generality than a component which sorts arbitrary objects. High generality also means high applicability. However, care has to be taken, because excessive generality leads to more complex components and they can consume time and resources unnecessarily; and The completeness of a component describes if it offers the expected functionality in the applied domain. One classical example of an incomplete component is a menu screen which needs to be painted on a device, like a mobile phone, and does not implement the paint method to show its content.

**4.4.2. Interactivity.** Interactive components receive from other environments, which can be other components or subsystems, different inputs in many different situations. For better understanding the concept of interactivity, we can establish a parallel between functions and objects, demonstrating how components may or may not interact. Functions transform a system from an initial state to a final state through some computation. They do not have memory, in other words, they do not keep the internal state and, for the same input, they will always give the same output. However, objects react to messages received from other objects and resend other messages to the senders performing some computation. Despite functions, an object can keep its internal state and can react in many different ways to the same input according to its state. An attribute which can influence the reuse of a component is its interactivity. A component may interact with others or with the user. The main target to be reached when the subject is interactive components is: high cohesion and low coupling. This means that components should have a high degree of conceptual unit and that the dependency on other components should be small.

High coupling can discourage the reuse of a component even if it is technically possible, because all the other components on which it depends might have to be incorporated into the design.

**4.4.3. Forms of Adaption.** Before a component is going to be inserted into a system, some adaptations are required on it. This phase is called **optimization** of a component, on which it will pass through some adaptations that do not affect its essential behavior. After that, a component may pass through a modification, what is not recommended because, in most of the cases, it decreases the reusability considering that the primary functionalities may be changed.

A simple example of optimization can be imagined if a graphical menu component is going to be displayed on a mobile phone screen, which may have different dimensions.

Then the component would be optimized to work fine in that specific screen size.

Another example for modification would be the case of source code copied from one to another application and then changed to be compatible with the new application. This action may throw negative points, such as different versions of the component running, could bring some inconsistencies into the source code.

**4.4.4. Concurrency.** The parallel execution of components is called concurrency. Concurrency is used to solve nondeterministic problems which do not need sequential execution. The concurrency is also used to gain execution speed and to eliminate potential processor idle time.

A concurrent component become non-deterministic, in other words, may give different results for the same input. It happens because a component does not know exactly which sequence of instructions is being executed at any given moment. Thus it should be considered while designing a component whether it will be used in concurrent environment. If yes then appropriate measures must be taken such that it behaves as required.

**4.4.5. Distribution.** Distributed systems, by definition, are physically or logically separated systems. The main reasons for the popularity of distribution are not cost considerations but increased capabilities, greater flexibility of incremental expansion and choice of vendors. It is possible to use, buy or sell components for a variety of platforms. A component that has this attribute of distribution should support being inserted into a distributed environment.

A distributed system consists of independently executing components. They can be implemented in the same programming language using communication mechanisms provided by that language or can also be implemented in different programming languages and paradigms only making use of language-independent mechanisms of communication. Besides data sharing, message passing is the central means for these components to communicate. A designer will have to consider the model which will be used for communication (like, CORBA, Web Services, and Messaging etc)

**4.4.6 Quality Control.** Quality assurance for components is an extremely complex task. On the other side, verifying this quality is even harder. The process is so complex that there is no formal verification of quality even for small components. The larger of them, written in many different programming languages and for different operating systems, does not make possible component verification with the intention of applying to it a quality stamp.

Some works on this area has been investigating it and presenting meaningful result. The main objective of this work is to define a software component certification process. Once these certification processes are in place, a designer will have to consider these while designing a component. Currently some tricks are used for ensuring that a component is also fault tolerant, such as pre and post conditions, constants and exceptions :

Pre-conditions are Boolean expressions that check if the given input parameters are valid and if the object is in a reasonable state to execute the requested task;

Post-conditions check if, after finishing the requested task, it was finished successfully according to the contract that the method or function is supposed to execute;

Constants are used every time that a method passes the control to a separate object or component; and

Exceptions represent another kind of quality assurance for the component.

They are used to handle atypical situations that happen during code execution. For example, an exception can be thrown in the moment that the system requires to read a file that is corrupted.

## 5. Conclusion

Code reuse is very common, but not commonly achieved by using tools and/or procedures especially developed for reuse. Developing reusable software is a complex task, which involves the systematic combination of methods, processes, and tools. On the other hand, the research community has continually discussed that a possible way of to obtain software reuse is with the adoption of a well-defined reuse process. However, the current processes present lack of details in important activities, plus some crucial gaps among its steps. In this context, this work proposes a simple three layer pyramid framework which represents necessary and sufficient attributes of a reusable piece of code. A large number of reusable code just end at the base layer. We find very few codes which have gained all the attributes and have reached the highest layer. Thus the framework is pyramid in nature which says it is easy to be at base and very few reach the top, but to gain maximum reuse a designer should thrive to put the code at the apex of the pyramid by decorating it with all the attributes of a reusable code.

## 6. References


1. *Sajjan G. Shiva, Lubna Abou Shala, "Software Reuse: Research and Practice", Department of Computer Science, The University of Memphis*
2. *Julio Cesar Sampaio do Prado Leite, Yijun Yu2, Lin Liu, Eric S. K. Yu, John Mylopoulos, "Quality-Based Software Reuse", 2Department of Computer Science, University of Toronto, M5S 3E4, Canada*
3. *Lisa Wold Eriksen ,"Code Reuse in Object Oriented Software Development", Norwegian University of Science and Technology, NTNU, Department of Computer and Information Science, IDI, Fall 2004*
4. *Prieto-Diaz, R. "Domain Analysis: An Introduction". In: ACM SIGSOFT Software Engineering Notes, April, 1990.*
5. *Bachmann, F.; Bass, L.; Buhman, C.; Comella-Dorda, S.; Long, F.; Robert, J.; Seacord, R.; Wallnau, K. "Technical Concepts of Component-Based Software Engineering 2nd Edition". Carnegie Mellon Software Engineering Institute, May, 2000.*
6. *Sametinger, J. "Software Engineering with Reusable Components". Springer-Verlag, 1997*
7. *Jean-Marie Burkhardt, Françoise Detienne, "An Empirical*

*Study Of Software Reuse By Experts In Object-Oriented Design", Institut National De Recherche En Informatique Et En Automatique*

8. *Heineman, G. T.; Councill, W. T. "Component-based Software Engineering : Putting the Pieces Together", Addison-Wesley, 2001.*
9. *Hamilton, G. "JavaBeans". Sun Microsystems, http://java.sun.corn/beans/. July, 1997*
10. *DeMichiel, L. G.; Yal L. El. C.; Krishnan, S. "Enterprise JavaBeans Specification, Version 2.0.", Sun Microsystems, Proposed Final Draft 2. April 2001.*
11. *Microsoft Corporation. "The Component Object Model Specification", 1995.*
12. *Object Management Group (OMG). "CORBA Components", 3.0 edition, 2002.*
13. *D'Souza, D.; Wills, A. C. Objects, "Components, and Frameworks with UML - The Catalysis Approach", Addison-Wesley, 2001.*
14. *Cheeseman, J.; Daniels, J. "UML Components: A Simple Process for Specifying Component-Based Software", Addison-Wesley, 2001.*
15. *Martin L. Griss, "Software Reuse Experience at Hewlett-Packard", Hewlett-Packard Laboratories,1501 Page Mill Road Palo Alto, CA 94301-1 126*
16. *Stan Garfield, "HP Consulting & Integration Knowledge Network ", Hewellet-Packard Development Company, December 15, 2005*
17. *Martin L. Griss, "Software Reuse at Hewlett-Packard", Software and Systems Laboratory, HPL-91-38, March, 1991*