Information Technology Journal 3 (2): 176-183, 2004
ISSN 1682-6027
© 2004 Asian Network for Scientific Information

A Software Process Model for Component-Based Development

L.F. Capretz
Department of Electrical and Computer Engineering,
University of Western Ontario, London, ON, Canada, N6A 5B9

Abstract: In the present study software life cycle and reusability issues that arise during component-based software development are discussed. The first part concentrates on mechanisms to achieve software reusability, such as composition and inheritance, it also outlines the main reasons why software is not extensively reused and examines the difficulties associated with software reusability. In the second part, the main issues concerning a seamless software life cycle model are considered, its purpose was to present a software life cycle model that takes component-based software development into account. Finally, the article covers other general aspects that form a more complete assessment of the whole software process; for instance, the most frequently used abstraction mechanisms and an estimate of the time and effort spent on each phase of the described life cycle model.

Key words: Component-based software development, software life cycle model, software process, software reusability

INTRODUCTION

A component is a self-contained piece of software that provides clear functionality, has open interfaces and offers plug-and-play services. Component-based software engineering is expected to have a significant impact on the software industry and hopefully on how software engineers construct systems, so this technique is here to stay[1]. There are initiatives in that direction, such as COM+[2], Enterprise JavaBeans[3], Component-Broker[4] and CORBA[5], among others. Component-based software engineering has broad implications for how software engineers acquire, build and evolve software systems. Hence, it is expected dramatic change in designer's primary roles and required skills for software development.

Numerous software life cycle models have been proposed. Their primary utility is to identify and arrange the phases and stages involved in software development and evolution. They also accrue guidance to the sequence in which the major tasks to construct and maintain a software system should be performed. So, it is appropriate to examine different software life cycle models in general and point out their strengths and weaknesses before an alternative one is put forward.

The Waterfall Model[6] has been long used by software engineers, but it takes no account of bottom-up development and prototyping. The Spiral Model[7] has been proposed mainly to speed up software development through prototyping, but without a clear and explicit goal, this process can degenerate into uncontrollable hacking. The Fountain Model[8] supports incremental and iterative software development, which takes place during the production of object-oriented software.

A growing number of companies in the software industry are following a process that iterates among design, building components, testing and feedback from customers[9]. Fingar[10] discusses the application of frameworks to e-business technical environment and addresses the issues that how component-based e-commerce frameworks are essential to the agility companies need to respond to rapidly changing e-commerce business models. However, one of the main shortcomings of such models is that none of them explicitly encourages reusability along their phases. Therefore, a software life cycle model that emphasizes the importance of component reuse during software development is very much in demand.

Reusing components: The concepts of software reusability and component-based software development are so interrelated that it is often difficult to talk about one without mentioning the other. This section describes an approach to software reuse while the analysis and design

Corresponding Author: L.F. Capretz, Department of Electrical and Computer Engineering, University of Western Ontario, London, ON, Canada, N6A 5B9 Tel: +1 519 6612111 x85482 Fax: +1 519 8502436
E-mail: lcapretz@eng.uwo.ca

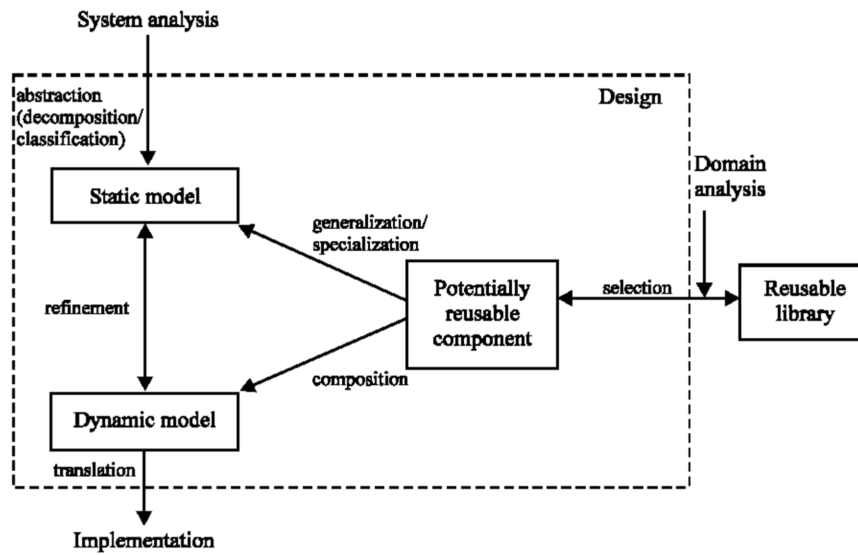

Fig. 1: Mechanisms for software reuse

phases are carried out. The strategy to be explained encourages the reuse of software in such way that while its steps are performed, reusability through composition, generalization and specialization mechanisms is considered.

The approach focuses on a collection of software systems within a certain application domain and encourages reuse of software components from an existing reusable library for that application domain. It addresses the mechanisms employed when components are reused from the reusable library. Moreover, it recognizes the iterative nature of software creation, hence repetitions are incorporated into the reuse process where appropriate.

The reuse process is well-suited for component-based software development because the composition, generalization and specialization are natural mechanisms, which are used in any software process. Underlying the reuse of software components, there are activities that address the identification of reusable components and their deployment into the developing system (Fig. 1).

Reusability implies development with reuse and development for reuse. Initially, the software engineer identifies potentially reusable components from an existing reusable library, the components are then selected and reused through composition, generalization and specialization mechanisms. At the end of the software development, there may be many new potentially reusable components that need to be classified and stored into the reusable library. In the future, such components can be reused in other systems.

Relationship between components: So far, most of the work that has been done in the reusability arena involves storing and recovering components from reusable libraries, but there are yet many complications related to reusing such components. To illustrate, as software systems become mature, the libraries may grow as domain-specific reusable libraries and reusable components can be added over time. It does not take long for such libraries to expand to enormous proportions and often with multiple versions of a component, which makes it difficult for designers to look for components that might meet their needs.

Reusable libraries are usually large and their organization turns them problematic to find potentially reusable components. One of the great difficulties in identifying reusable components lies in the fact that there is discordance in terminology between different people, that is, a component which someone is looking for is described in the libraries by unfamiliar or unexpected terminology.

In order to reuse a component, naturally, the software engineer must find it first. Therefore, a good classification scheme for arranging components is vital to the selection process. The arrangement can be an aid to understanding a component when software reuse demands adaptability of that component to match new requirements. Besides, the search for a component is a difficult task in that it must be selected the one which requires the least effort to adapt, with the goal being an exact matching between what is needed and what is available. The learning curve for reusable components may be substantial. However, the time and effort required for this selection process is

decreased by the presence of semantic information within the reusable library. Therefore, software engineers must be able to find a connection between what is needed and what is available. Relationships between components can be used to facilitate the search for potentially reusable ones.

The relationships between components could be: has-a, is-a, uses-a, is-part-of-a. Such relationships could be taken as a classification scheme to provide a network of pre-defined links between components, thus introducing some semantic information and a vocabulary into the reusable library.

A scheme for representing relationships between components of a reusable library entails on organizing them through a set of pre-defined relations. Such relations allow components to be arranged and connected to others that can also be reused. In addition, relations express links between different components, facilitating the understanding of the components. Relations applied to express design information between two or more reusable components can help solve the problem of discordance of terminology between people because the relations establish fixed semantic information between components. Four different relations to link components are proposed:

Compose (<component-1>, <list-of-components>): This relation represented <component-1> as a composition of components in a <list-of-components> (has-a relationship). Complex software system behaviour can be achieved with compositions that combine the simple behaviour of several types of components.

Inherit (<component-1>, <component-2>): This relation indicated that <component-1> is a generalization of <component-2> or the other way round that <component-2> is a specialization of <component-1> (is-a relationship).

Use(<component-1>,<list-of-components>): This relation indicated that <component-1> interacts with components in a <list-of-components> (uses-a relationship). It means that any operation of <component-1> uses any operations defined in any component in a <list-of-components>.

Framework (<component-1>, <framework-1>): This relation associates a <component-1> with a <framework-1> defined by the software engineer (is-part-of-a relationship).

The relations compose, inherit and use can be perceived straightforwardly, whereas the framework relation should be made explicit by the software engineer. The relations presented may be viewed as an alternative textual notation to describe a software system. The information present in a reusable library is compatible with the information dealt with by the software process model to be presented later in the article. Therefore, enquiries to a reusable library, which stores such relations and manipulates the same concepts as component-based process model to be described, can be undertaken.

Reusability process: The decisions involving the reuse of a component are very important in that, the one which requires the least effort to adapt must be selected, with an exact match between what is needed and what is available being the goal. Basically, the selection of a component from a reusable library imposes four steps:

1. Identifying the required (target) component
2. Selecting potentially reusable components
3. Understanding the components
4. Adapting (specializing, generalizing, composing or adjusting) the components to satisfy the needs of the developing software system

The search for a component in a reusable library can lead to one of the following possible results:

- An identical match between the target and an available component is reached
- Some closely matching components are collected, then adaptations are necessary
- The requirements are changed in order to fit available components
- No reusable component can be found, then the target should be created from scratch

Following a procedure which helps select potentially reusable components is fundamental to the reuse process. The procedure (Fig. 2) illustrates a typical attempt to reuse a component from a reusable library. The procedure elucidates exclusively the selection of potentially reusable components; classification and storage issues are considered later in the next section. By properly arranging a component using the relations described previously, the chance of finding potentially reusable components is increased. Furthermore, the effort required to get a suitable component is reduced because the classification scheme based on relations can guide designers through the various components quickly and efficiently.

Selection involves browsing to find a component, retrieving it and transferring it from the reusable library to the system database. While searching for components it is necessary to address the equivalence between the target component and any near matching components. The best component selected for reuse may also require

```
begin
 //The process of component reuse.
 //Given a keyword for a target component,
 //search libraries for potentially reusable components and their relations.

if identical match between the target and an available component exists
 then
 //reuse by composition
 retrieve it and reuse it
 else
 collect fitting components
 for   each collect component
 assess the degree of matching
 endfor
 rank the components
 select the best component
 if     the target can be a subclass of the best component
        then
        //reuse by specialization
        put the target as subclass and inherit commonalities
        else
              if the target shares commonalities with the best component
              then
                    //reuse by generalization
                    create a new abstract superclass
                    put the target and the best component as subclasses
              else
                    //specialization and generalization are not convenient
                    if possible
                    then
                          adjust the best component to the requirements or
                          adjust the requirements to the best component
                    else
                          //reusability is not possible
                          create the target component from scratch
                    endif
              endif
        endif
 endif
end
```

Fig. 2: A procedure for software reuse

specialization, generalization or adjustment to the requirements of the new software system in which it will be reused. Sometimes, it is preferable to change the requirements in order to reuse the available components. The adaptability of components depends on the difference between the requirements and the features offered by existing components, as well as the skill and experience of the designer. The process of adapting components is the least likely to become automated in the reusability process.

Lifetime of reusable components: Not only does reusability involve reusing existing components in a new software system, but also designing components that are meant for reuse. While a piece of software is being developed, it might be realized that some components can be generalized and reused in future software development. An important issue in the quest of reusability is how to make a potentially reusable component available to other

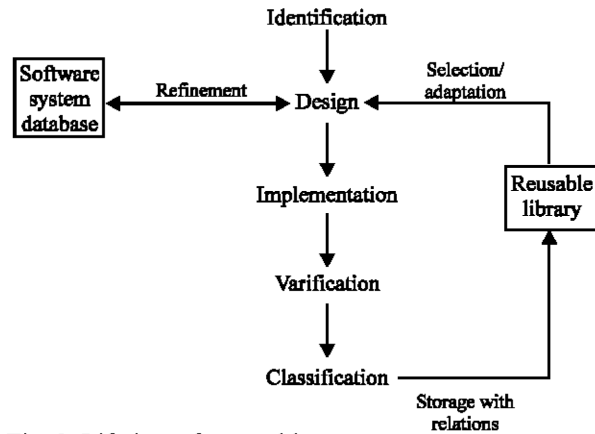

Fig. 3: Lifetime of a reusable component

people. The component must be understandable, well-written and well-documented. Lastly, the component must be easily adaptable for different uses, either in original or in modified form. Therefore, developing reusable components is considerably more difficult and imposes much greater expense than producing ordinary components, although it may still be worth the investment.

As component-based software is produced essentially out of interrelated collections of independently developed components, it is important to understand the stages that such components go through. The stages reflect the activities involving the identification, design, implementation, verification, classification, storage, refinement, selection and reuse of the component. Figure 3 depicts the lifetime of a reusable component.

Through the promotion of the specialization, generalization and composition mechanisms during the design phase, application-dependent components should be revised, so that they can be sufficiently generic to be of use in a wider range of applications rather than in the single system for which they were originally developed. This generality requires extra effort during the design and implementation phases in the short term, but in the long term, after a sufficiently broad reusable library is created, it will lead to a significant reduction in overall software development time and effort.

Development for reuse should therefore be with generality in mind with perhaps less emphasis on satisfying the specific needs of the application that is being designed. In contrast, the specific parts of a design are those parts that turn a general set of components into a specific software design for a particular application. For example, one can have the design of a general set of classes for dealing with text string in a windowing environment. One specific application could use that set of classes to design a text-formatting program; another

application could use the same classes to define a text editor.

Tools can play an important role during the manipulation of reusable libraries by selecting or storing components and browsing the libraries. There must be in fact two kinds of libraries: reusable libraries from which components of interest can be picked up and to which new generic reusable components can be added, as well as the software system database that keeps information concerning a particular software under construction. Modification of components in the reusable library is not recommendable; a copy of the component should be taken into the software system database and refinements carried out there.

If a newly implemented component does not exist in the reusable library, then a decision has to be made as to whether that new component should be considered as reusable and be incorporated into the reusable library. Before a component is added to the library, it must be verified and frozen. The verification is just applied to the component, not to the whole software system and should include treatment of exceptional conditions. Storing a component also involves getting it from the software system database, classifying it, relating it to other components and putting it into the reusable library.

For a component to be a viable candidate for inclusion into a reusable library, it must first:

- Be clearly defined in terms of its interface and functionality
- Have a reasonable performance in terms of time and space required to execute its operations
- Be a generic abstraction, which means that the functionality it provides must be sufficient enough to model the real-world entities abstracted
- Have a robust behaviour if it is misused or pushed to its limits, that is, exceptions must be handled

It is also very important to separate sets of client components from server components. Client components are often application-dependent and they make decisions and switch the control flow among several server components. Client components should not directly perform calculations or implement complex algorithms. On the other hand, server components perform specific and detailed operations, executing general computations to implement a certain self-contained algorithm and rarely change the control flow. Therefore, server components are more likely to be reused in other systems than client components since the former are more application-independent and basically wait to receive requests from client components. Thus, as far as design for reuse is concerned, sets of server components are preferable.

A component-based process model: The creation of software is characterized by change and instability and therefore any diagrammatic representation of the component-based process model should consider overlapping and iteration between its phases. A consensus may be drawn on the phases pertinent to a software life cycle. Although the main phases may overlap each other and iteration is also possible, the planned phases are: system analysis, domain analysis, design (static and dynamic) and implementation. Maintenance is an important operational phase, in which bugs are corrected and extra requirements met.

An outcome of this software life cycle model is the emphasis on reusability during software development and evolution and the production of reusable components meant to be useful in future projects. This is naturally supported by the object-oriented paradigm due to inheritance and encapsulation. Reusability also implies the use of composition techniques during software development. This is achieved by initially selecting reusable components and aggregating them, or by refining the software to a point where it is possible to pick out components from the reusable library, as explained above.

Figure 4 represent a pictorial representation of how the system analysis, domain analysis, design, implementation and maintenance phases proceed iteratively over time and how reuse of components from the reusable library is taken into consideration within the software life cycle model. Reusability within this life cycle is smoother and more effective than within the traditional models because it integrates at its core the concern for reuse and its mechanisms.

System analysis: This phase involves high-level analysis of the application for the purpose of understanding its essential features. The system analysis phase demands the system analyst to:

- Study the application and its constraints
- Understand the requirements expected to be satisfied by the software system
- Create an abstract model of the application in which these requirements are met

This phase may conduce to the identification of the major parts of the application, so that the system can be divided into large components based on the functionality that should be offered. A glimpse of the preliminary components that model the application can come up as well.

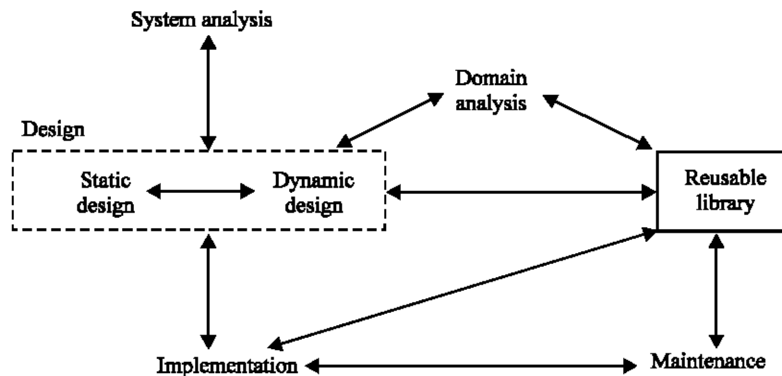

Fig. 4: A component-based life cycle model

At this stage, the services delivered by a software system helps figure out its subsystems and major components. However, as compared to functional decomposition, this phase is neither concerned with the details of functions in terms of algorithms, nor which functions can be refined into other sub-functions, but it worries over mapping the application in terms of components. The result of this phase is an abstract model of the application, which may be graphical or textual, using a formal or informal method, as the system analyst wishes.

Domain analysis: Domain analysis involves the examination of a certain application domain and seeks to identify and arrange entities that commonly occur in systems within the application domain. Thus, domain analysis is an activity that should be carried out at the beginning of software production.

The domain analysis phase primarily seeks to abstract and arrange concepts that form the vocabulary of the application domain. At this stage a common terminology is drawn. Large applications should be broken into parts, so that specialists in a specific application domain can carry out the domain analysis in that application domain.

During this phase, the abstract model of the application comprising high-level abstractions of software components may be refined and new components can be defined. Therefore, the boundary between system analysis and domain analysis may at times seems fuzzy because identifying key abstractions in the application domain may be viewed as part of system analysis or domain analysis. Nevertheless, at this level, domain analysis is also concerned with the identification and organization of potentially reusable components.

Design: Design is an exploratory process. When designers face an application, they should not ask "How do I work out a solution to this problem?" Instead, they should ask, "Where are components that I can directly or indirectly reuse to solve this problem?" A procedure to guide them over this task has been presented above. At this point, they should be able to examine a reusable library and to select components that closely match the component necessary to build the software.

The designer looks for components trying out a variety of schemes in order to discover the most natural and reasonable way to model the application. There has been a tendency to present software design in such a manner that it looks easy to do. Nevertheless, in the design of large and complex software, identification of key components is likely to take some time. During the design phase the primary concern is to build a design model that fulfils the overall software functionality.

The construction of the design model involves identifying relevant components and producing both the static design and the dynamic design. The static design captures the generic and essential features of a system and can be expanded to other systems within the same application domain. In contrast, the dynamic design captures behavioural aspects of a certain system and is therefore more difficult to generalize to other systems.

As more components are identified along the design, re-evaluation of the complete set of components is required. Repetitions are not unusual, since a good design usually takes several iterations. The number of reiterations also depends on the designer's insight, experience and knowledge about the application domain. A bottom-up strategy should be considered if the software engineer does not have a good perception of the application domain.

Some components picked out during the design phase should undergo further refinements (e.g. treatment of exceptional conditions) until they become generic and robust enough to be placed in a reusable library. This surely adds an overhead to software construction, which is more than compensated for by the long term savings when such components are reused in future projects.

Table 1: Input, tasks and output of each phase

Phases vs I/O	Input	Tasks	Output
System analysis	application: user need and software system requirements	create an abstract model of the application	abstract model of the application
Domain Analysis	abstract model of the application	identify possible reusable components	potentially reusable components
Design	abstract model of the application and potentially reusable components	build static and dynamic models (design model)	static (generic) and dynamic (behaviour) models
Implementation	static (generic) and dynamic (behaviour) models	implement the models	software system solution to the application
Maintenance	delivered software system plus changes to be introduced	implement the changes	updated release of the software system

Table 2: Phases versus abstraction mechanisms

Phases vs mechanisms	System analysis	Domain analysis	Design	Implementation	Maintenance
Classification	X	X	X	X	X
Instantiation			X	X	X
Generalization		X	X	X	X
Specialization		X	X	X	X
Decomposition	X		X		
Composition		X	X	X	X

Implementation: The implementation phase is characterized by the translation of a design model into a programming language. The design model comprises static concepts and dynamic behaviour represented by the output of the design phase. In this phase the major tasks involve the implementation of the identified components, along with the cooperation among them.

The line between design and implementation is also a blurred one. Implementing a component requires defining the data structures reciprocal to attributes and the algorithms corresponding to operations of that component. It is also necessary to implement the control flow that realizes the interaction between components and specify the overall software behaviour. The best idea is to isolate a component and decide whether a match can be reused, or if it has to be implemented from scratch.

Maintenance: During software maintenance, changes are introduced to a delivered software system. Such changes are not meant only for correcting errors occurred during operational software. These changes may be also for enhancing, updating the system to anticipate future errors or adapting the system in response to a modification in the environment. After changes are introduced to the system, an updated release of the software is generated.

During the maintenance phase, software components may be accessed from, as well as new ones may be added to the reusable library of the concerned application domain. For instance, a change to adapt the software to a new environment may specialize already existing component, so that characteristics of the new environment are taken into consideration, hence expanding the spectrum of environments the reusable components are able to deal with.

The input, tasks performed and output of each phase, which evolves dynamically as the understanding a software engineer has about the system grows (Table 1). The phases are traceable during software construction and evolution, as well as determine a component-based software life cycle model.

Mechanisms prevalent in each life cycle phase: The most frequently used mechanisms in each phase of the software life cycle model are pointed out in Table 2. These mechanisms are part of the abstraction process inherent to software development. The results are based on the development of a generic graphical interface for CASE environments.

The system analysis phase emphasizes arrangement of high-level concepts in a real-world application and decomposition of the software system. Several mechanisms are relevant to the domain analysis stage, but specialization, generalization and composition are vital to achieve reusability. In the design phase all mechanisms are fundamental. During the implementation and maintenance phases, almost all mechanisms are essential except decomposition, since at these latter stages the foremost partition of the software have been done.

Percentage of time per each development stage: Although it is difficult to draw distinct lines between two adjacent phases, Figure 5 indicated an approximate percentage of the amount of time likely to be spent on each phase for a complete development of a system. These statistics have been taken from the construction of a few software systems. Despite the system analysis, design and implementation phases being deeply interrelated, it is clear that the design phase takes longer because most of the tasks are done during such a phase.

Domain analysis is relevant to discover potentially reusable components during software production. Consequently, the amount of time spent on this phase, naturally, must not be longer than that spent on other phases. If the perceived cost of finding a certain component is higher than the cost of creating a new component from scratch, then all hope for reuse is lost. For this reason, it is important to have at least minimal library tools that allow software engineers to quickly select and add reusable components as they are

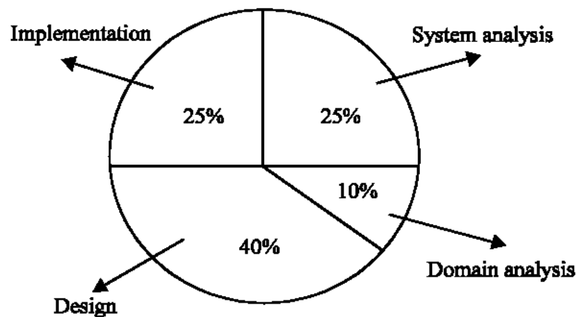

Fig. 5: Phases versus software development time

identified. Although maintenance accounts for the majority of software costs, it is not included in Fig. 5 because it can be viewed as an operational phase that succeeds software development. It is felt that the basic reuse issues that the software development model encourages forms a useful basis for supporting software development and evolution.

One great advantage of this software process model is the conceptual continuity across all phases of the software life cycle. Not only do the software concepts remain the same from system analysis down through implementation and maintenance, but they also stay uniform during the refinement of a design. Therefore, when the described model is employed, the design phase is linked more closely to the system analysis and the implementation phases because software engineers have to deal with similar abstract concepts throughout software construction and evolution.

During the system analysis and domain analysis phases, user needs, software requirements, functionalities, objectives and constraints of the system are very much of interest. Thus, it is important to understand the real-world application and an abstract model of that application should be achieved. When the design phase is entered, the abstractions are refined. The design process should stop when the key generic abstractions and the software behaviour are detailed enough to be translated into a programming language. Thus, the design phase generates the templates for the implementation stage.

A software system is not merely produced out of reusable components. On the contrary, usually, components selected and derived from reusable libraries are combined with newly written components and all of them have to be bound together in the final software. It is natural that with some of the components, the software designer will face the decision of whether to reuse them straightforwardly, adapt them and reuse, or write them from scratch. The break-even-point of reusing versus redoing lies where the cost of search plus adaptation exceeds the cost of producing the respective piece of software.

REFERENCES

1. Capretz, L.F., M.A.M. Capretz and D. Li, 2001. Component-based Software Development. 27th IEEE Conference of the Industrial Electronics Society (IECON'01), Denver, pp: 1834-1837.
2. Microsoft, 2004. COM+, www.microsoft.com/com/tech/complus.asp.
3. SUN, 2004. Enterprise Java Beans. www.java.sun.com/products/ejb/index.html.
4. IBM., 2004. Component Broker. www.software.ibm.com.
5. Object Management Group, 2004. The Common Object Request Broker Architecture. www.omg.org.
6. Royce, W.W., 1987. Managing the Development of Large Software Systems. Proceedings of IEEE 9th International Conference on Software Engineering, Monterey, CA, IEEE Press, pp: 328-338.
7. Boehm, B.W., 1988. A spiral model of software development and enhancement. IEEE Computer, 21: 61-72.
8. Henderson-Sellers, B. and J.M. Edwards, 1990. The object-oriented systems life cycle. Communications of the ACM., 33: 142-159.
9. Cusumano, M.A. and R.W. Selby, 1997. How Microsoft builds software. Communications of the ACM., 40: 53-61.
10. Fingar, P., 2000. Component-based framework for E-commerce, Communications of the ACM., 43: 61-66.