International Journal of
Emerging Research in Management &Technology
ISSN: 2278-9359 (Volume-4, Issue-6)

Research Article

June 2015

Component-Based Testing Optimization: A Review for Component-Based Testing

Kanika Rani
Mtech Student (CSE)
JMIT Radur, Haryana, India

Vivek Sharma
H.O.D and Asst. Prof (CSE)
JMIT Radur, Haryana, India

Abstract—

Component-based software development method is based on the concept to develop software systems by selecting correct components and then to assemble them with a well-defined software architecture. In order to integrate components available from previous projects or from external suppliers, it is essential to check whether the components have functional and non-functional properties sufficient for the new system. This paper studies the various features (other than functional) that a component should possess to be fit for reuse. In this paper, we draw current component-based software technologies comparison, describe their advantages and disadvantages, and discuss the features they inherit.

Key words –quality software, reusable components, nonfunctional requirements, reusability metrics, and software reuse, software crisis, components based systems.

I. INTRODUCTION

The software industry has adopted component based software development approach because it promises high quality software. In the traditional approaches of software development, quality of the software was sacrificed in order to deliver the software product within time and budget limits .This phenomenon was called as 'software crisis' [4]. Component based software development method focuses on reuse of already existing software artifacts (known as software components). This approach better manages the complexity of software, the major cause of software crisis. The basic notion of this approach is that complex software should not be developed from scratch. Rather we should look for those subparts (components) of the software that already exist. These components may be used as such or they can be customized for the new application [1]. This will not only result in reduced cost and shorter time to market of the software but will also increase the productivity of the developer. So that she can concentrate on the critical aspects of the problem which will lead to the desired goal of high quality software.

II. REUSE SUCCESS DEPENDS UPON QUALITY OF COMPONENTS

Component based development and software reuse places new demands on software testing and quality assurance. This is true if the components are to be traded between organizations. A component will die or live by its quality. If the component does not possess the properties requiredfor reuse, it will die its own death. Component should undergo energetic tests, perhaps more than others programs because errors will be introduced into every application that utilizes the components. If the project has very high reliability and availability requirements, the reputation of the developer is at stake.

Blaming the third party for the unreliable behavior of the project will not relieve the developer from his responsibility. Every user reusing the component has some questions in her mind · Will the component provide required functionality in the new application? · Is it (the component) tested according to my proposed use? · What are its effects on the new system regarding performance, reliability, robustness, maintainability, portability etc? A component is fit for reuse, if it provides the required behavior in the intended context [4]. If we want high levels of reuse, then various contexts for the usage of the components should be considered. A software component may be used for many different applications, in different business and technical environments, by different developers using different methods and tools, for different users in different organizations.

A. Why Component-Based Development?

1) *Contain Complexity***:**In any complex situation there are a few basic techniques that can be used to understand and manage that complexity. These are the techniques of abstraction, decomposition, and incremental development. Any solution to application development must provide ways to support these techniques.
2) *Reduce Delivery Time:*The ability to deliver solutions in a timely manner is an essential feature of any software development project. With the increased rate of change of technology, this aspect is even more critical. This need for reduced delivery time for software-intensive systems is often referred to as working at "Internet speed."
3) *Improve Consistency***:** Most software-intensive systems share characteristics with others previously developed, in production, or yet to be produced. It must be possible to take advantage of this commonality to improve consistency and reduce development expense.

4) *Make Use Of Best-In-Class:*.In a number of areas there are well developed solutions offering robust, best-in-class functionality and performance. Taking advantage of these solutions as part of a larger development effort is essential.
5) *Increase Productivity***:**Thelack of software development skills is causing a major bottleneck for systems users. Any new approaches must improve the productivity of skilled employees to allow them to produce quality results at a faster rate.
6) *Improve Quality***:**As the economic and human impact of failure of software-intensive systems increases, greater attention must be turned to the quality of the deployed systems. A goal must be to support the building of systems correctly the first time, without extensive (and expensive) testing and rewriting.
7) *Increase Visibility Into Project Progress:*Managing large software projects is a high-risk undertaking. To help this, greater visibility must be possible throughout the software life cycle. This requires an incremental approach to development, delivery, and testing of software artifacts.
8) *Support Parallel And Distributed Development***:** Distributed development teams require methods that encourage and enable parallel development of systems. This requires attention be given to manage complexity due to the need to partition and resynchronize results. This list represents a discouraging set of challenges for any approach. Yet it is components and component based approaches that offer the most promising attempt to meet the challenges head-on, and provide the basis of a new set of techniques supporting the next generation of software-intensive solutions.

III. CHARACTERISTICS OF A GOOD QUALITY COMPONENT

Apart from the functionality that a component provides, another aspect of a component is its nonfunctional properties, such as understandability, portability etc. These properties are expressed in terms of quality attributes. In the context of building systems from existing components, the characterization of the component's "ileitis" and their impact on the system are particularly important because the components are usually provided as black boxes [4]. The characteristics, a good quality component should possess, are discussed below:

A. Understandability

Understandability is defined based on the estimated effort needed by a user to recognize the concept behind a component and its applicability [3]. Understandability further can be attributed to documentation. Documentation of a component covers how to use the component (e.g. user manual) and how to configure the component (e.g. setup or reference manual). If the documentation of the component is provided, users of the component can easily understand the component's usage, which the component's developer assumes.

B. Adaptability

The ease with which component can accommodate to change. The ease with which a component can be modified for use in applications or environments other than those for which it was specifically designed.

C. Interoperability

It is the ability of a component to interact with other components of a host software system in a unified manner, or the ability to adapt the component with minimal effort to achieve the same goal. Dependencies on other components can be assessed by the required interface and the provided interface that the component requires and offers.

D. Portability

It is the ability of a component to operate on a wide variety of computer platforms with little modification, if required. The component should be easily and quickly portable to specified new environments if and when necessary, with minimized porting costs andschedules.

E. Generosity

A reusable component must be generic, that is, it has appropriate features that enable the re-user to create specific instances of the components to satisfy application specific requirements. Assessment of generosity is related to how difficult it is put the component is operation such as installation, un- installation.

F. Dependability

This property refers to the trustworthiness of a component which allows confidence to be justifiably placed on the service it offers. It encompasses aspects of reliability, availability, safety, security, usability and extendibility.

G. Cohesive

It should carry out a set of related services. It addresses a specific need. The need may be across domains such as a word processor (horizontal reuse), or domain specific such as a system for emergency services for Airlines (Vertical reuse). Components that are domain specific most often supply an internal business need. But the developers should not make the component so specific that it is not easily reused by other applications. It should have the potential of reuse.

H. Independence

A component should be as independent as possible from other components. Minimizing dependence on other components makes a component more useful and easily interoperable. Complex components tend to have some dependencies on other components. But the dependencies on such components, which may change, should be minimized.

I. Having Well Defined Interface

In component based development, each component will provide (export) and require (import) pre-specified services from other components. Component Interface is a quality concern because it plays a major role in joining and persevering independently developed component.

Table: Comparison among current component technologies

	Cobra	**Ejb**	**Com?Dcom**
Development environment	Underdeveloped	Emerging	Supported by a wide range of strong development environments
Binary interfacing standard	Not binary standards	Based on **com** java specific	A binary standard for component interaction is the heart of **COM**
Compatibility & portability	Particularly strong i standardizing language bindings; but not so portable	Portable by java language specification ; but not very compatible	Not having any concept of source-level standard of standard language binding
Modification & maintenance	**COBRA IDL** for defining component interfaces, need extra modification & maintenance	Not involving **IDL** files, defining interfaces between component and containers easier modification & maintenance	Microsoft **IDL** for defining component interfaces, need extra modification & maintenance
Services provided	A full set of standardized services; lack of implementations	Neither standardized nor implemented	Recently supplemented by a number of key services
Platform dependency	Platform independent	Platform independent	Platform dependent
Language dependency	Language independent	Language dependent	Language independent
Implementation	Strongest for traditional enterprise computing	Strongest on general web clients	Strongest on the traditional desktop applications

IV. MEASURING REUSABILITY - STATE OF THE ART

Software measurement is a challenging but important component of a healthy software engineering culture. Mostly software projects run over schedule and budget limits, and still end up with quality problems. The goal of software measurement is to quantify the schedule, work effort, product size, and project status and quality performance. Comparison of the desired data with the actual data indicates the progress status of the project. The statistics thus obtained can also help in the future projects [5]. Software metrics provide a quantitative basis for the development activities of a software project. Metrics can be used to improve software productivity and quality [3- 4]. Poulin [6] gives two approaches for measuring reusability - empirical methods and qualitative methods. Realistic methods depend upon objective data. An analyst can calculate them easily and cheaply, which is a veryneeded property of a metric. Empirical studies compare attributes of reusable components to attributes of components which are not reused. The attributes of reusable software must influence reusability. Qualitative methods define the attributes of reusable software and assessors subjectively assess how the software to be studied adheres to these attributes. The use of assessments makes qualitative methods more expensive than empirical methods. Selby [7] has studied the NASA projects in which software reuse was a success. He has noted down factors behind this success. Mainly that a reusable software module is small in size and its interface is simple. It has few dependencies on other modules. Good documentation is also one of the properties. The reuse of the low level system and utility functions is more common than the reuse of human interface functions. He validated these results statistically. The ESPRIT-2 project REBOOT [6] (reuse based on object oriented techniques) has also given four reusability factors which are specified using a given number of criteria. Each criterion has at least one metric. Reusability is a number varying from 0 to 1, which is calculated by normalizing the metrics. The four reusability factors are portability, flexibility, understandability, and the confidence of the re-user. Portability expresses the ease of reuse in another environment. The principles of flexibility are generality and modularity. Criteria of under stability include code complexity, self descriptive business, documentation quality and component complexity. Confidence is the probability of error free reuse as assessed by the re-user. IBM method [6] stresses that the developer need to have access to the development project's documentation in addition to the code. Fonash [2] grouped the software component reuse metrics into five major categories: general, quality, parameterization, coupling and cohesion. General metrics are related to size, type and understandability of the component. Quality metrics measure the ease of change, formatting of code, comments etc. Parameterization metrics measure the number of functional or data parameters in a module. Coupling

metrics measure the degree of independence of modules. Cohesion metrics measure functional cohesion as well as data cohesion. The functional cohesion measures the degree to which every part of a module is necessary for performing a single function. The data cohesion metric addresses the degree to which the module has a single data type associated with it. HironariWashizaki et al [3] have given five metrics for measuring the reusability of components, which, according to them, consists of understandability, testability, usability and portability. The proposed five metrics are

- Existence of Meta Information (EMI) - if EMI = 1, users of the component can easily understand components' usage.
- In order to have high understandability, testability and usability, a 1/2 or 1/3rd of component's attributes should be readable properties.
- In order to have high usability of component, 1/3rd of component's attributes should be writable properties.
- In order to have high portability of the component, 3/4th of the component's business methods should be without return value.
- The presence or absence of method parameters is not strongly related to the component' quality. In all these studies, the criteria chosen for assessing the quality of a component are more or less same. We can identify that understandability, portability, modifiability, good documentation, independence, confidence of the re-user are some of the features that a component meant for reuse should have. Need of the hour is to formalize all these characteristics and quantify them using empirical methods. Consider a new application, *X*, that requires 60 percent new code and the reuse of three structure points, SP1, SP2, and SP3. Average costs for qualification, adaptation, integration, and maintenance are available.

overall effort = Enew + Equal + Eadapt + Eintwhere
Enew= effort required to engineer and construct new software components .
Equal = effort required to qualify SP1, SP2, and SP3.
Eadapt= effort required to adapt SP1, SP2, and SP3.
Eint= effort required to integrate SP1, SP2, and SP3.
The effort required to qualify, adapt, and integrate SP1, SP2, and SP3 is determined by taking the
average of historical data collected for qualification, adaptation, and integration of the reusable
components in other applications. The benefit associated with reuse within a system *S* can be expressed as a ratio
Rb(S)= [Cnoreuse – Creuse]/Cnoreuse where Cnoreuse is the cost of developing *S* with no reuse.
Creuse is the cost of developing *S* with reuse. Devanbuand his colleagues [DEV95] suggest that
Rbwill be affected by the design of the system since Rb is affected by the design, it is important to make Rb a part of an assessment of design alternatives the benefits associated with reuse are closely aligned to the cost benefit of each individual reusable component.
A general measure of reuse in object-oriented systems, termed *reuse leverage* [BAS94], is defined as
Rlev= OBJ reused/OBJ built where
OBJ reused is the number of objects reused in a system.
OBJ built is the number of objects built for a system.

V. CONCLUSIONS

Organizations considering existing software for reuse must consider various factors (including cost of reuse) to decide if the software is fit for reuse. One such factor is quality of software. Hence the organization must be able to assess the quality of components. Organizations, which are developing software that is proposed for reuse, must also be able to assess whether the software meets the reusability criteria. This paper identifies certain quality characteristics that can be used to assess the reuse potential of a software component. However more research is needed to ensure the completeness of these quality characteristics.

REFERENCES

[1] C. Szyperski, 'Component Software: Beyond Object-Oriented Programming', AddisonWesley, 1999.
[2] Fonash P.,' Characteristics of reusable software code components', Ph.D. Dissertation,George Mason University, 1993.
[3] HironariWashizaki et al , 'A Metrics suite for measuring Reusability ofsoftwareComponents' , 9th IEEE International Symposium on Software Metrics, 2003.
[4] Ian Sommerville,' Software Engineering', Pearson Education, 6th edition.
[5] Pentti Virtanen , 'Measuring and Improving Component Based software Development' ,Ph. D. thesis , department of Computer Science, university of Turku , Turku , Finland.
[6] Poulin J. S., 'Measuring software reuse, principles, Practices and Economic Models',Addison Wesley Publishing, 1997.
[7] Selby, R.W: ,'Quantitative studies of software reuse', Software reusability, vol 2, ed.Biggerstaff, T.J and Perlis, A.J. Addison Wesley, 1989

