# Chapter 7

## Impacts and Challenges of CBST

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Introduction</td>
<td>115</td>
</tr>
<tr>
<td>7.2</td>
<td>Component-Based Software Testing and Quality Issues</td>
<td>115</td>
</tr>
<tr>
<td>7.3</td>
<td>Component-Based Software Testing and Testing Infrastructure Inadequacies</td>
<td>117</td>
</tr>
<tr>
<td>7.4</td>
<td>Impacts of Inadequate Testing and Testing Infrastructure</td>
<td>118</td>
</tr>
<tr>
<td>7.5</td>
<td>Component Traceability and Testability</td>
<td>120</td>
</tr>
<tr>
<td>7.6</td>
<td>Software Component Features and Testability Challenges</td>
<td>122</td>
</tr>
<tr>
<td>7.7</td>
<td>Summary</td>
<td>127</td>
</tr>
</tbody>
</table>
CHAPTER-7 Impacts and Challenges of CBST

7.1 INTRODUCTION
Software testing remains a fundamentally important way to check that software behaves as required. In the era of software development, it was realized by both researchers and software developers that traditional testing was not enough helpful for software with fast changing requirements of software industry. The implementation of new system “from scratch” takes long development and testing time, high production costs and difficulties in achieving further evolution and adaptations to new demand (Antonia et al., 2003), due to these reasons, the researchers and software developers has shifted from object-oriented approach to Component-Based Software Engineering (CBSE) approach to develop and test software. In order to develop and test complex software, CBSE approach is much better which is based on reusability of existing and testable components. But due to the inadequacy of Component-based Software Testing (CBST) tools, techniques and processes this study presents the various issues of CBST and testing infrastructure inadequacies. This chapter highlights the various impacts and challenges of inadequate testing and testing infrastructure which hinder the software development by increasing failures due to poor quality and software development costs.

7.2 COMPONENT-BASED SOFTWARE TESTING AND QUALITY ISSUES
Component-based Software Testing (CBST) is an important activity of Component-based Software Development (CBSD) and is based on two main perspectives. First is component providers that refer to the testing of components by their developers on the basis of the source code and the second is the component user that refers to the testing of components by their users, without access to their source code (Voas et al., 1995). Even if source code is available, components should be only black-box tested because of the missing knowledge about the internal structure of the component and the danger of losing oneself in too much detail. The component provider perspective addresses testing issues that are of interest to the providers of the software components. The component
providers view the components independently of the context in which the components are used, and effectively test all configurations of the components in a context-independent manner. The component user perspective, in contrast, addresses testing issues that concern the user of software components. The component users view the components as context-dependent units because the component users’ application provides the context in which the components are used. The user is thus concerned only with those configurations or aspects of the behavior of the components that are relevant to the component users’ applications. Testing issues in the software testing (Hans, 2002) include redundant testing during integration of components, availability of source code, heterogeneity of language, platforms and architectures, monitoring and control mechanism in distributed software testing, deadlocks and race conditions. These issues can be removed if the testing requirements and test process documentation are clearly defined because these are the two main factors that help in improving the Component-Based Testing (CBT) and quality (Gill and Tomar, 2007). Quality is defined as the bundle of attributes present in a commodity and, where appropriate, the level of the attribute for which the consumer holds a positive value. Defining the attributes of component software quality and determining the metrics to assess the relative value of each attribute are not formalized processes. Compounding the problem is that numerous metrics exist to test each quality attribute. Because users place different values on each attribute depending on the product’s use, it is important that quality attributes be observable to consumers. However, with component software there exist not only asymmetric information problems, but also instances where the developer truly does not know the quality of his own product. It is not unusual for component software to become technically obsolete before its performance attributes have been fully demonstrated under real-world operation conditions. As software has evolved over time so has the definition of component software quality attributes. McCall, Richards, and Walters’, 1997 software quality model characterizes attributes in terms of three categories - product operation, product revision, and product transition. In 1991, the
International Organization for Standardization (ISO) adopted ISO 9126 as the standard for software quality to reduce challenges. It is structured around six main attributes are (McCall et al., 1977) functionality, reliability, usability, efficiency, maintainability, and portability.

7.3 COMPONENT-BASED SOFTWARE TESTING AND TESTING INFRASTRUCTURE INADEQUACIES

Component-based Software Testing (CBST) is the action of carrying out one or more tests, where a test is a technical operation that determines one or more characteristics of a given software element or system, according to a specified procedure. The means of CBST of hardware and software require a procedure, including the executable test suite used to carry out the testing (NIST, 1997). Historically, CBSE focused on writing code and testing specific lines of that code. Very little effort was spent on determining its fit within a larger system. Testing was seen as a necessary evil to prove to the final consumer that the product. More recently, software developers started to invest more time and resources in integrating the different pieces of software and testing the software as a unit rather than as independent entities. The amount of effort spent on determining the developmental requirements of a particular software solution has increased its importance. Forty percent of the software developer effort is now spent in the requirements analysis phase. The worldwide market for software testing tools was $931 million in 1999 and is projected to grow to more than $2.6 billion by 2004 (Shea, 2000). However, such testing tools are still fairly primitive. The lack of CBST technique, infrastructure and quality metrics leads most companies to simply count the number of defects that emerge when testing occurs. Few organizations engage in other advanced testing techniques, such as forecasting field reliability based on test data and calculating defect density to benchmark the quality of their product against others. Numerous issues affect the software testing infrastructure and may lead to inadequacies. For example, competitive market pressures may encourage the use of a less than optimal amount of time, resources, and training for the testing function (River et al., 1998) and with current software
testing tools developers have to determine whether applications and systems will interoperate. In addition, the need for certified standardized test technology is increasing. The development of these tools and the accompanying testing suites often lag behind the development of new software applications (ITToolbox, 1999). Standardized testing tools, suites, scripts, reference data, reference implementations, and metrics that have undergone a rigorous certification process would have a large impact on the inadequacies listed above. For example, the availability of standardized test data, metrics, and automated test suites for performance testing would make benchmarking tests less costly to perform. Standardized automated testing scripts along with standard metrics would also provide a more consistent method for determining when to stop testing. In some instances, developing conformance testing code can be more time consuming and expensive than developing the software product being tested. Addressing the high testing costs is currently the focus of several research initiatives in industry and academia.

7.4 IMPACTS OF INADEQUATE TESTING AND TESTING INFRASTRUCTURE

Currently, there is a lack of readily available testing technique, performance metrics, procedures, and tools to support CBST. If these infra technologies were available, the costs of performance certification programs would decline and the quality of software would increase. This would lead to not only better testing for existing products, but also to the testing of products that are not currently tested. This study introduces the several impacts on the software industry due to lack of robust, standardized test technology that can be grouped into four general categories (RTI, 2002) - failures due to poor quality, software development costs, time to market due to inefficient testing and market transaction costs.

7.4.1 Increased Failures Due to Poor Quality

The most troublesome effect of a lack of standardized test technology is the increased incidence of avoidable product defects that emerge after the product has been shipped. According to RTI, 2002, in the aerospace industry over a billion
dollars has been lost in the last several years that might be attributed to problematic software. Large failures tend to be very visible. They often result in loss of reputation and loss of future business for the company. Recently legal action has increased when failures are attributable to insufficient testing. Software defects are typically classified by type, location introduced, when found, severity level, frequency, and associated cost. The individual defects can then be aggregated by cause according to the following approach. Lack of conformance to standards, where a problem occurs because the software functions and/or data representation, translation, interpretation do not conform to the procedural process or format specified by a standard.

7.4.2 Increased Software Development Costs
Historically, the process of identifying and correcting defects during the software development process represents over half of development costs. Depending on the accounting methods used, testing activities account for 30 to 90 percent of labor expended to produce a working program (Beizer, 1990). Early detection of defects can greatly reduce costs. Defects can be classified by where they were found or introduced along the stages of the software development life cycle, namely, requirements, design, coding, unit testing, integration testing, system testing, installation/acceptance testing, and operation and maintenance phases. Longer a defect stays in the program; the more costly it becomes to fix it. There are two type of failure cost, first one is internal failure costs and second one is external failure costs according to (Arne, 2008). Internal failure costs are costs that are caused by products or services not conforming to requirements or customer/user needs and are found before delivery of products and services to external customers. They would have otherwise led to the customer not being satisfied. Deficiencies are caused both by errors in products and inefficiencies in processes. Examples include the costs for rework, delays, re-designing, shortages, failure analysis, re-testing, downgrading, downtime, and lack of flexibility and adaptability. External failure costs are costs that are caused by deficiencies found after delivery of products and services to external customers, which lead to
CHAPTER-7 Impacts and Challenges of CBST

customer dissatisfaction. Examples include the costs for complaints, repairing goods and redoing services, warranties, customers' bad will, losses due to sales reductions, and environmental costs.

7.4.3 Increased Time to Market
The lack of standardized test technology also increases the time that it takes to bring a product to market. Increased time often results in lost opportunities. For instance, a late product could potentially represent a total loss of any chance to gain any revenue from that product. Lost opportunities can be just as damaging as post-release product failures. However, they are notoriously hard to measure. If standardized testing procedures were readily available, testers would expend less time developing custom test technology. According to RTI, 2002, standardized test technology would accelerate development by decreasing the need to develop specific test software for each implementation, first is develop specific test data for each implementation, and second is use the “trial and error” approach to figuring out how to use nonstandard automated testing tools.

7.4.4 Increased Market Transaction Costs
Due to the lack of standardized test technology, purchasers of component and Component-Based System (CBS) incur difficulties in comparing and evaluating systems. This information problem is so common that manufacturers have warned purchasers to be cautious when using performance numbers for comparison and evaluation purposes. Standardized test technology would alleviate some of the uncertainty and risk associated with evaluating software choices for purchase by providing consistent approaches and metrics for comparison.

7.5 COMPONENT TRACEABILITY AND TESTABILITY
In CBSD, component traceability and testability are important capabilities, which support productivity and quality assurance. CBSD are increasingly being used to reduce the cost and time of the software development. Components intend to be reused across various software, possibly in different environment so all component must be easily traceable and testable adequately. In the CBSD
software reuse support design methodologies for which the main activity is not the building of new systems from scratch, but the integration, modification, and expansion of existing one. The CBSD focuses on developing large software system by integrating previously existing software component, software architecture, software framework and software design pattern (Gerhard et al., 2005). This approach can potentially be used to reduce software development cost, assemble system rapidly, and reduce the maintenance burden. However according to (Antonia et al., 2003), it is only achieved via a severe design discipline and by adopting standard modeling notations as well as strict documentations and design rules that components independently built and effectively interact.

7.5.1 Traceability of Component-Based Software

Traceability of CBS refers to the extent of its built-in capability of tracking the status of component attributes and component behavior. Traceability is of two types, first one is behavior traceability, and second one is trace controllability. Behavior traceability refers to the degree to which a component facilitates the tracking of its internal and external behaviors. In the real world, component engineers have learned to check and monitor the internal and external behaviors of software components by adding a program tracking mechanism in software. In the real practice, engineers are not successful in delivering software components with good behavior traceability due to the following two reasons. First, in the development of software components, engineers used to pay much attention to track component internal behaviors than external behaviors. Therefore, component testers, integration engineers and customers have the difficulty in monitoring and checking the external behaviors of components (Jerry et al., 1999). This study advice from the above discussion that component developers learn design and development of traceable components due to the lack of standardized component trace formats and tracking mechanism.
CHAPTER-7  Impacts and Challenges of CBST

7.5.2 Testability of Component-Based Software
Software testability is one of important concepts in design and testing of software program and components. Building programs and components with good testability always simplifies test operations, reduces test cost, and increases software quality. As pointed out by James Bach, there is a set of program characteristics that lead to testable software, including operability, observability, controllability, understandability. Mary, 2000 views' software testability is one of three pieces of the reliability puzzle. They pointed out that software testability analysis is useful to examine and estimate the quality of software testing using an empirical analysis approach (Jerry, 2000). In the component engineering paradigm, software development of CBS, engineers have several questions concerning component testability - what is component testability and related factors? How to check, measure, or evaluate the testability of software components? How to design and develop testable components to achieve good testability? Design and development process of testable component will discuss in next chapter to increase the testability.

7.6 TESTABLE COMPONENT FEATURES AND TESTABILITY CHALLENGES
Software testability is one of the important concepts of designing and testing of software components. Building programs and components with good testability simplifies test operations, reduces test cost, and enhances software quality, productivity and reusability (Jerry, 2000). Testable components require support for development of components on different platforms, support for development of different variants of components for different software, support for maintenance, enhancement and independent development of components, support for how component programmed and tested again when reused, how combination and composition of component possible and lastly support for the adaptation of selected components in the development process of CBSD. The tests of software components are performed in order to verify the correctness of the functional interface. An ideal testable component is not only deployable and executable but
also testable and reusable with the support of standardized components' test facilities (Pressman, 1997). Unlike normal components, testable components must be traceable. Traceable components are the ones constructed with a built-in tracking mechanism for monitoring various component behaviors in a systematic manner and must have a set of built-in interfaces. It reduces the effort of a tester to test the component and helps in controlling the traceability of the component. Lastly, the testable components must have their distinct functional features, data and interfaces, and must have a well-defined test architecture model and built-in test interfaces to support their interactions to component test suites and component test-bed (Robert et al., 1988). There are several testability challenges of software components such as how to reuse component tests cases, how to construct testable and reliable components, how to construct component test drivers and stubs, and how to build a generic and reusable component testbed (Pressman, 1997). In the component engineering paradigm, one of the primary goals is to generate reusable components as software products. The third-party engineers use the components as parts to build specific software systems according to the requirements given by customers. Therefore, the testability of a program highly depends on the testability of involved components and their integration. There are several concerns on building and testing of software components. An ideal testable component is not only deployable and executable, but also testable with the support of standardized components' test facilities. Unlike normal components, testable components have the following features and these features help to reduce the testability challenges.

7.6.1 Test Cases
The study has identified two distinct categories of test cases - First is state inspection test cases and second is state comparison test cases to reduce the testability challenge “how to reuse component tests”. State inspection test cases involve executing sequences of methods that yield “results”, and then physically examining those results. State comparison test cases involve executing two sequences of methods and comparing to satisfy some predetermined relationship.
The primary key to the reuse of component test cases is to develop some systematic methods and tools to set up reusable component test suites to manage and store various component test resources, including test cases, test data, and test scripts. In the current engineering practice, software development teams use an ad-hoc approach to creating component test suites through a test management tool. Since existing tools usually depend on different test information formats, repository technologies, database schema, and test access interfaces, it is difficult for engineers to deal with diverse software components (e.g. third-party components) with a consistent test suite technology (Robert et al., 1988). This problem affects the reuse of component tests in the component acceptance testing and component integration. To solve this problem this study introduces two alternatives. The first is to create a new test suite technology for components with plug-in-and-test techniques. With this technology, engineers are able to construct a test suite for any component, and perform component tests using a plug-in-and-test technique. Clearly, it is necessary for us to standardize software component test suites, including test information formats, test database schema, test access interfaces, and to define and develop new plug-in-test techniques to support component unit testing at the unit level. The other alternative approach is to create component tests inside components, known as build-in tests. Unlike the first approach, where component tests created and maintained in a test suite outside of a component, this approach creates component tests inside components. Clearly, it simplifies component testing and reduces the component test cost at the customer side if a friendly test-operation interface is available. To execute the built-in component tests, extra function facilities to perform test execution, test reporting, and test result checking is required. Therefore, there is a need to standardize test access interfaces supporting the interactions among components, test suites, and built-in tests.

7.6.2 Reliability and Testability of Components

CBSE techniques are gaining substantial interest because of their potential to improve productivity and lower development costs of new software applications,
yet satisfying high reliability requirements. To reduce the second challenge "how to construct testable and reliable components" this study address high reliability and verified requirements are the need of the researchers and practitioners. To merit the attribute "reliable", a component should be extensively validated. As far as testing is the technique most commonly used for validation, this means that reusable components should be well tested. For tests to be applied efficiently and on time, a component should be testable. An ideal testable software component is not only deployable and executable, but also testable with the support of standardized components test facilities. Unlike normal components, testable components must be traceable. According to Jerry (Jerry et al., 1999), traceable components are ones constructed with a built-in tracking mechanism for monitoring various component behaviors in a systematic manner. Testable components must have a set of built-in interfaces to interact with a set of well defined testing facilities. This reduces the effort of tester to test the component and helps in controlling the traceability of the component. Although testable components have their distinct functional features, data and interfaces, they must have a well-defined test architecture model and built-in test interfaces to support their interactions to component test suites and a component test-bed by keeping all the testability challenges for enhancing testing in CBSD. To reduce the testability challenge "how to construct testable and reliable components", study proposes a process to create testable component and it will discuss in next chapter.

7.6.3 Reusable Component Test-Bed
This study introduces a approach to reduce the testability challenge "how to build a generic and reusable component testbed". According to this approach testbed element is to expand on the requirements, establish guidelines for each requirement, and prioritize the requirements. Hardware, software, environment, and reusable component guidelines are used to evaluate candidates for addition to the testbed. The testbed may contain several equivalent elements, e.g., compilers. The availability of similar but different capabilities will support more and varied
experiments than a single set of elements. Add tools or software to the testbed when tool or software that provides similar functionality. The guidelines will be applied to each element before it is added to the testbed. Thus, the guidelines will change over time as the needs of the project evolve. To experiment with reusable components in a subsystem development, the testbed environment needs to contain a reasonably complete software development environment that supports all phases of the life cycle. Since the testbed will not remain static during the life of the project, the environment (Robert et al., 1988). In general, a program test execution environment consists of several supporting functions - test retrieval, test execution, test result checking, and test report to create a reusable testbed.

7.6.4 Test Drivers and Stubs

Study introduces two general approaches of generating test stubs to reduce the testability challenge "how to construct component test drivers and stubs". The first approach is model-based, in which component stubs are developed to simulate a component, and the second approach is operational script-based, in which test stubs are constructed to simulate a specific functional behavior of a component in a black-box view. The major advantage of this approach is the flexibility and reusability of test stubs during the evolution process of components. Study also introduces a new systematic method to construct test drivers and stubs for diverse components and various customizations. The essential issue is how to generate reusable, configurable, manageable test drivers and stubs for a component. Software engineers use ad-hoc approaches to develop module-specific or product specific test drivers and stubs based on the given requirements and design specifications (Jerry, 2000). The major drawback of this approach is that the generated test drivers and stubs are only useful to a specific project. It is clear that the traditional approach causes a higher cost on the construction of component test drivers and stubs. In the component engineering paradigm, components might have the customization function to allow software engineers to customize them according to the given requirements. Study suggests that the traditional way to construct component test drivers and stubs is very
expensive and inefficient to cope with diverse software components and their customizable functions. Component test drivers must be script-based programs that only exercise its black-box functions. There are two groups. The first group includes function-specific test drivers. The second group contains scenario-specific test drivers. Component test stubs are needed in the construction of component frameworks.

7.7 SUMMARY
This chapter is divided into two main sections. The first section covers the various issues of CBST and testing infrastructure inadequacies. This chapter highlights and concludes various impacts of inadequate testing and testing infrastructure which hinder the software development by increasing failures due to poor quality, software development costs, time to market due to inefficient testing and market transaction costs. Second section covers the various challenges of CBST with component traceability and testability. The study also presents the main features of testable components with testability challenges such as test cases, testable components, reusable component test-bed, and test drivers and stubs. This chapter summarizes the various testability challenges that the software developer face during the CBSD and it is summarizes as how to reduce these challenges with the testable components features.