Component-Based Software Engineering

ECE493-Topic 5
Winter 2007

Lecture 33 – Testing Component-Based Systems (Part B)

Ladan Tahvildari
Assistant Professor
Dept. of Elect. & Comp. Eng.
University of Waterloo

March 26, 2007

Component Testability: Issues and Challenges

- Component Testability issues in CBSE:
  - How to construct components with high testability? (in other words, how to create testable software components?)
  - How to increase component testability in a component reuse process?
  - How to check component testability during a component development process?
  - How to measure component testability in a component development process?

- Challenges in Studying Component Testability:
  - Creating component testability models
  - Finding systematic methods to create testable components
  - Developing systematic methods to verify component testability
  - Defining measurement methods and metrics for component testability

Design for Component Testability

- Design for component testability refers to *all engineering activities to enhance component testability for software components in a component development process.*

- Challenges in Building Testable Components:
  - How to specify testability requirements for components?
  - How to construct components to achieve high testability? (including construction approaches, component architecture, test interface, ….)
  - How to support test automation for testable components?
  - How to verify generated component testability in a systematic solution?
  - How to measure and analyze the testability of components during a component development process in a systematic approach?

Three Approaches

- Method #1: Framework-Based Testing Facility
  - Creating well-defined framework (such as a class library) is developed to allow engineers to add program test-support code into components according to the provided application interface of a component test framework.

- Method #2: Build-in Tests
  - Adding test-support code and built-in tests inside a software component as its parts to make it testable.

- Method #3: Systematic Component Wrapping for Testing
  - Using a systematic way to convert a software component into a testable component by wrapping it with the program code that facilitates software testing.
**Built-in Test Components**

- **Definition:**
  - a special type of software component in which special member functions are included as its source code for enhancing software testability and maintainability.

- **Major Features:**
  - Built-in test components are able to operate in two modes:
    - Normal Mode – a component behaves as its specified functions.
    - Maintenance Mode – its internal built-in tests can be activated by interacting a tester (or user).
  - Built-in tests as a part of a component.

- **Major Limits:**
  - Only limited tests can be built-in tests due to component complexity
  - It is costly to change and maintain built-in tests during a component development process.

---

**Comparison of Three Approaches**

<table>
<thead>
<tr>
<th>Different Perspectives</th>
<th>Framework-Based Testing Facility</th>
<th>Built-in Tests</th>
<th>Systematic Component Wrapping for Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming Overhead</td>
<td>Low</td>
<td>High</td>
<td>Very Low</td>
</tr>
<tr>
<td>Testing Code Separated from Source Code</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Software Tests inside Components</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Test Change Impact on Components</td>
<td>No</td>
<td>High</td>
<td>No</td>
</tr>
<tr>
<td>Software Change Impact on Component Testing Interfaces</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Component Complexity</td>
<td>Low</td>
<td>Very High</td>
<td>High</td>
</tr>
<tr>
<td>Usage Flexibility</td>
<td>High</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Applicable Components</td>
<td>In-house components and newly developed components</td>
<td>In-house components and newly developed component</td>
<td>In-house components and COTS as well as newly constructed components</td>
</tr>
</tbody>
</table>

---

**What is a Testable Component?**

- "A testable bean is a testable software component that is not only deployable and executable, but is also testable with the support of standardized components test facilities." (by Jerry Zeyu Gao et al.)

- **Requirement #1:** A testable bean should be deployable and executable.
  - A Java Bean is a typical example.

- **Requirement #2:** A testable bean must be traceable by supporting basic component tracking capability that enables a user to monitor and track its behaviors.

- **Requirement #3:** A testable bean must provide a consistent, well-defined, and built-in interface, called component test interface, to support external interactions for software testing.

- **Requirement #4:** A testable bean must include built-in program code to facilitate component testing by interacting with the two provided test interfaces to select tests, set up and run tests, and check test results.

---

**Why Do We Need Testable Components?**

- The major goal of introducing testable components is to find a new way to develop software components which are easily to be observed, traced, tested, deployed, and executed.

- The major advantages of testable components:
  - **Increasing component testability** by enhancing component understandability, observability, controllability, and test support capability.
  - **Standardizing component test interfaces and interaction protocols** between components and test management systems and test suite environments.
  - **Reducing the effort of setting up component test beds** by providing a generic plug-in-and-test environment to support component testing and evaluation.
  - Providing the basic support for a systematic approach to automate the derivation of component test drivers and stubs.
Principles of Building Testable Components

- The essential needs in constructing testable components are:
  - Well-defined component models concerning test support
  - Consistent test interfaces between components and external test tools and facilities
  - Effective ways and mechanisms to construct testable components

- The basic principles of building testable components:
  - It is essential to minimize the development efforts and program overheads when we increase component testability by providing systematic mechanisms and reusable facilities.
  - It is important to standardize component test interfaces for testable beans so that they can be tested in a reusable test bed using a plug-in-and-play approach.
  - It is always a good idea to separate the component functional code from the added and built-in code that facilitates component testing and maintenance.

Maturity Levels for Testability

- Level #1- Initial – At this level, component developers and testers use an ad hoc approach to enhance component testability in a component development process.

- Level #2- Standardized – At this level, component testability requirements, design methods, implementation mechanisms, and verification criteria are defined as standards.

- Level #3- Systematic – At this level, a well-defined component development and test process and systematic solutions are used to increase component testability at all engineering phases.

- Level #4- Measurable – At this level, component testability can be evaluated and measured using systematic solutions and tools in all component development phases.

Verification of Component Testability

- Check component testability of software components using well-defined verification means during a component development process.

  - Static Verification Approach
    - Using various verification methods to check the generated component artifacts in all phases, including component requirements, interface specifications, design logic, implementation, and test cases and results.
    - This enhances component testability by discovering testability issues in all phases of a component development process

  - Statistic Verification Approach
    - Using statistical methods to analyze and estimate component testability by examining how a given component will behave when it contains faults.
    - This suggests the testing intensity or testing difficulty in discovering a fault at a specific location.
    - This suggests the number of tests necessary to gain quality confident.
Static Verification Approach

- **Component Specification Phase:**
  - Checking component requirements are clearly specified so that they can be tested and measured for a given test criteria.
  - How to specify them? How to verify them for testability?

- **Component Design Phase:**
  - Checking component design for testability -> focusing how the current component design to meet the given testability requirements, including component model, architecture, interfaces for testing, test facility design
  - How to verify design artifacts for component testability?

- **Component Implementation Phase:**
  - Checking if component design for testability has been properly implemented

- **Component Testing Phase:**
  - Checking if component tests based on the given test criteria
  - Measuring component testability based on a component testability model

Statistical Verification Approach

- Use a statistical approach to examine how a given program behave when it contains a fault.
  - A proposed verification approach (sensitivity analysis) to check program testability.
    - Execution probability
    - Infection probability
    - Propagation probability
  - Its major objective is to predict the probability of a software failure occurring if the particular software contains a fault for a given set of test set for black-box testing.

Measurement of Software Testability

- What is software testability measurement?
  
  Software testability measurement refers to the activities and methods that study, analyze, and measure software testability during a product development cycle.

- Three types of measurement methods:
  - **Program-Based Measurement Methods**
    - Measure program testability by considering the single faults in a program
  - **Model-Based Measurement Methods**
    - Use the data flow model to measure software testability
  - **Dependability Assessment Methods**
    - Measure software testability based on the dependency relationships between inputs and corresponding outputs.

Program-Based Measurement Methods

- The basic idea of this approach is similar to software mutation testing.

- To compute the testability of a software at a specific location based on a single failure assumption:
  - A single fault is instrumented into the program at a specific location.
  - The newly instrumented program is compiled and executed with an assumed input distribution.
  - Three basic techniques (execution, infection, and propagation estimation) are used to compute the probability of failure that would occur when that location has a fault.
Model-Based Measurement Methods

- Normalizing a program before the testability measurement using a systematic tool.
  - Structure normalization and block normalization
- Identifying the testable elements of the target program based on its normalized data flow model.
  - Including number of non-comment lines, nodes, edges, p-uses, defs, uses, d-u paths, and dominating paths
- Measuring the program testability based on data flow testing criteria
  - Including ALL-NODES, ALL-EDGES, ALL-P-USES, ALL-DEFS, ALL-USES, ALL-DU-PAIRS and ALL-DOMINATING PATH

Dependability Assessment Methods

- A black-box approach for testability measurement
- Testability is computed based on the probability of a test of the program based on a given input setting is rejected by the program due to its faulty
- The basic approach consists of the following steps:
  - Decide in a manual (or systematic) mode whether a given program behave correctly on a given test
  - Analyze the behavior of the program against its specification
  - Observes the input and the output of each test against the expected output, and looks for failures