MTAT.03.094
Software Engineering

Lecture 09:
Verification & Validation (Testing)

Dietmar Pfahl
email: dietmar.pfahl@ut.ee

Fall 2016
Schedule of Lectures

Week 01: Introduction to SE
Week 02: Requirements Engineering I
Week 03: Requirements Engineering II
Week 04: Analysis
Week 05: Development Infrastructure I
Week 06: Development Infrastructure II
Week 07: Architecture and Design
Week 08: Refactoring
Week 09: Verification and Validation I
Week 10: Guest Lecture I (Testing)
Week 11: Guest Lecture II (Software Quality Management)
Week 12: no lecture
Week 13: Verification and Validation II + Agile/Lean Methods
Week 14: Guest Lecture III (Software Craftsmanship)
Week 15: Course wrap-up, review and exam preparation
Week 16: no lecture
Structure of Lecture 09

- Testing Basics
- Testing Levels
- Testing Methods
- Testing Types
- Testing Artefacts
- Metrics
Exercise: A Pen

• Quality?
• Testing?
Software Quality – Definition

• **Software quality is the degree of conformance to explicit or implicit requirements and expectations**

Explanation:
• *Explicit*: clearly defined and documented
• *Implicit*: not clearly defined and documented but indirectly suggested
• *Requirements*: business/product/software requirements
• *Expectations*: mainly end-user expectations
Software Quality – Dimensions (1)

- **Accessibility**: The degree to which software can be used comfortably by a wide variety of people, including those who require assistive technologies like screen magnifiers or voice recognition.
- **Compatibility**: The suitability of software for use in different environments like different Operating Systems, Browsers, etc.
- **Concurrency**: The ability of software to service multiple requests to the same resources at the same time.
- **Efficiency**: The ability of software to perform well or achieve a result without wasted energy, resources, effort, time or money.
- **Functionality**: The ability of software to carry out the functions as specified or desired.
- **Installability**: The ability of software to be installed in a specified environment.
- **Localizability**: The ability of software to be used in different languages, time zones etc.
Software Quality – Dimensions (2)

- **Maintainability**: The ease with which software can be modified (adding features, enhancing features, fixing bugs, etc).
- **Performance**: The speed at which software performs under a particular load.
- **Portability**: The ability of software to be transferred easily from one location to another.
- **Reliability**: The ability of software to perform a required function under stated conditions for stated period of time without any errors.
- **Scalability**: The measure of software’s ability to increase or decrease in performance in response to changes in software’s processing demands.
- **Security**: The extent of protection of software against unauthorized access, invasion of privacy, theft, loss of data, etc.
- **Testability**: The ability of software to be easily tested.
- **Usability**: The degree of software’s ease of use.
Software Product Quality Model – ISO 25010 Standard
Software Quality Assurance (SQA)

- **SQA is a set of activities for ensuring quality in software engineering processes (that ultimately result in quality in software products).**

It includes the following activities:
- Process definition
- Process implementation
- Auditing
- Training

Processes could be:
- Software Development Methodology
- Project Management
- Configuration Management
- Requirements Development/Management
- Estimation
- Software Design
- Testing
- ...
Software Quality Control (SQC)

- **SQC is a set of activities for ensuring quality in software products.**

  It includes the following activities:
  - Reviews
  - Testing

  **Reviews:**
  - Requirement Review
  - Design Review
  - Code Review
  - Deployment Plan Review
  - Test Plan Review
  - Test Cases Review

  **Testing:**
  - Unit Testing
  - Integration Testing
  - System Testing
  - Acceptance Testing
Verification

Definition
• The process of evaluating work-products (not the actual final product) of a development phase to determine whether they meet the specified requirements for that phase.

Objective
• To ensure that the product is being built according to the requirements and design specifications. In other words, to ensure that work products meet their specified requirements.

Question
• Are we building the product right?

Evaluation Items:
- Plans, Requirement Specs, Design Specs, Code, Test Cases

Activities:
- All kinds of reviews and testing with exception of requirements review and acceptance testing
Validation

Definition
• The process of evaluating software during or at the end of the development process to determine whether it satisfies specified (or implicit) business requirements.

Objective
• To ensure that the product actually meets the user’s needs, and that the requirements were correct in the first place. In other words, to demonstrate that the product fulfills its intended use when placed in its intended environment.

Question
• Are we building the right product?

Evaluation Items:
- Requirement, actual product/software

Activities:
- Requirements review, acceptance testing
Software Development Life Cycle (SDLC)

- The SDLC, or Software Development Process, defines the steps/stages/phases in the building of software.

- Various kinds of software development models exist, e.g.:
  - Waterfall model
  - Spiral model
  - Iterative and incremental development (like ‘Unified Process’ and ‘Rational Unified Process’)
  - Agile development (like ‘Extreme Programming’ and ‘Scrum’)

SDLC in summary:
- Project Planning
- Requirements Development
- Estimation
- Scheduling
- Design
- Coding
- Test Build/Deployment
- Unit Testing
- Integration Testing
- User Documentation
- System Testing
- Acceptance Testing
- Production Build/Deployment
- Release
- Maintenance
Software Testing Life Cycle (STLC)

- The STLC defines the steps/stages/phases in testing of software.
## Software Testing Life Cycle (STLC)

<table>
<thead>
<tr>
<th>Phase</th>
<th>Activity</th>
<th>Deliverables</th>
<th>Attitude needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements/ Design Review</td>
<td>You review software requirements/design (if existing)</td>
<td>Review Reports (listing the defects)</td>
<td>Curiosity</td>
</tr>
<tr>
<td>Test Planning</td>
<td>After gathering a general idea of what needs to be tested, you ‘plan’ for the tests</td>
<td>Test Plan, Test Estimation, Test Schedule</td>
<td>Farsightedness</td>
</tr>
<tr>
<td>Test Designing</td>
<td>You design/detail your tests on the basis of detailed requirements/design of the software</td>
<td>Test Cases, Test Data, Test Scripts, Requirements, Traceability Matrix</td>
<td>Creativity</td>
</tr>
<tr>
<td>Test Environment Setup</td>
<td>You setup the test environment (tools)</td>
<td>Test Environment</td>
<td>Interest in test technology</td>
</tr>
<tr>
<td>Test Execution</td>
<td>You execute your Test Cases/Scripts in the Test Environment to see whether they pass</td>
<td>Test Results (intermediate), Defect Reports</td>
<td>Patience</td>
</tr>
<tr>
<td>Test Reporting</td>
<td>You prepare various reports for various stakeholders</td>
<td>Test Results (final), Test/Defect Metrics, Test Closure Report</td>
<td>Accuracy, Diplomacy</td>
</tr>
</tbody>
</table>
STLC integrated with SDLC

Actual Needs and Constraints

User Acceptance (alpha, beta test)

Delivered Package

Review

System Specifications

System Test

Analysis / Review

Subsystem Design/Specs

Integration Test

Analysis / Review

Subsystem

Module Test

User review of external behavior as it is determined or becomes visible

Test Levels

User/Component Specs

Module

Unit/Component Specs

Subsystem

System Integration

MTAT.03.094 / Lecture 09 / © Dietmar Pfahl 2016
Test Case

- A test case is a set of conditions or variables under which a tester will determine whether a system under test satisfies requirements or works correctly.

- Templates and examples of formal test case documentation can be found here:

  http://softwaretestingfundamentals.com/test-case/
Test Case

A **Test Case** consists of:

- A set of inputs + expected outputs
- Execution conditions

Example of ‘execution condition’:

When pressing the ‘save’ button of a word processor, what happens depends on what you did previously (e.g., what you typed in or deleted)

---

**Test Suite** = set of Test Cases

**Test Data** = input to a Test Case

**Test Oracle** = condition that determines whether a test case passed or failed (-> fail happens if actual output is different from expected output)

**Test Verdict** = decision of whether a test passed or failed
Test Case – Recommendations

• As far as possible, write test cases in such a way that you test only one thing at a time. Do not overlap or complicate test cases. Attempt to make your test cases ‘atomic’.

• Ensure that all positive scenarios and negative scenarios are covered.

• Language:
  • Write in simple and easy to understand language.
  • Use active voice: Do this, do that.
  • Use exact and consistent names (of forms, fields, etc).

• Characteristics of a good test case:
  • Accurate: Exacts the purpose.
  • Economical: No unnecessary steps or words.
  • Traceable: Capable of being traced to requirements.
  • Repeatable: Can be used to perform the test over and over.
  • Reusable: Can be reused if necessary.
Test Script

• A **Test Script** is a set of instructions (written using a scripting/programming language) that is performed on a system under test to verify that the system performs as expected. Test scripts are used in automated testing.

• Examples of Test Frameworks supporting test scripting:
  • JUnit, Selenium, Sikuli, …
Test Script – Examples

JUnit

Sikuli

def sample_test_script(self):
    type ("TextA")
    click (ImageButtonA)
    assertExist (ImageResultA)

@Test
public void shortRegularRental() {
    Customer customer = new Customer("Cust");
    Movie movie = new Movie("Groundhog Day", REGULAR);
    Rental rental = new Rental(movie, 2); // 2 days rental = short
    customer.addRental(rental);
    String expected = "Rental Record for Cust\n"
    expected += "\tGroundhog Day\t2.0\n"
    expected += "Amount owed is 2.0\n"
    expected += "You earned 1 frequent renter points"
    Assert.assertEquals(expected, customer.statement());
}
Definition 1: Error – Fault – Failure (according to IEEE Standard)

- **Failure** is an event caused by a **fault**, and a **fault** is an anomaly of the software caused by an **error**

- **Error** – mistake made by human (e.g., programmer)
- **Fault** – wrong/missing statement in the software (code)
- **Failure** – inability to perform the program’s required functions (correctly)
- **Defect** – Bug?

- **Debugging** / Fault localization – localizing, repairing, re-testing.
Definition 1: Error – Fault – Failure (according to IEEE Standard)

Fault sources
- Lack of skills/training
- Oversight
- Poor communication
- ‘Lost in translation’
- Immature process

Impact on / of software program
- Errors
- Faults
- Failures

Fault context

User’s point of view
- Poor quality software
- User dissatisfaction
Definition 2: Error – Fault – Failure (as it is often used in IDEs/tools)

- **Failure** is an event caused by an **error**, **error** is a state of the program caused by a **fault** in the code
  - **Fault** – wrong/missing statement in code (resulting in error)
  - **Error** – incorrect program state (may result in a failure)
  - **Failure** – inability to perform its required functions (correctly)
  - **Defect**? – **Bug**?

- **Debugging** / Fault localization – localizing, repairing, re-testing.
Definition 2: Error – Fault – Failure

Example:

Inputs:
Correct (=Expected) result?
Actual result?
Fault? Error? Failure?

Program state: x, i, count, PC
Definition 2: Error – Fault – Failure

Example:

```java
public static int numZero (int[] x) {
    //Effects: if x==null throw NullPointerException
    //        else return the number of occurrences of 0 in x
    int count = 0;
    for (int i = 1; i < x.length; i++) {
        if (x[i]==0) {
            count++;
        }
    }
    return count;
}
```

Wrong State:
- `x = [2,7,0]`
- `i = 1`
- `count = 0`
- `PC=first iteration of if`

Expected State:
- `x = [2,7,0]`
- `i = 0`
- `count = 0`
- `PC=first iteration of if`

Fix: `for (int i = 0, i < x.length; i++)`  

- `x = [2,7,0]`, fault executed, error; no failure
- `x = [0,7,2]`, fault executed, error; failure

State of the program: `x, i, count, PC`

Correct/expected result: count = 1
Software V & V Myths – and Facts (1)

**MYTH:** Quality Control = Testing.
**FACT:** Testing is just one component of software quality control. Quality control includes other activities such as reviews.

**MYTH:** The objective of Testing is to ensure a 100% defect-free product.
**FACT:** The objective of testing is to uncover as many defects as possible while ensuring that the software meets the requirements. Identifying and getting rid of (debugging) all defects is impossible.
Software V & V Myths – and Facts (2)

**MYTH:** Testing is easy.
**FACT:** Testing can be difficult and challenging (sometimes, even more so than coding).

**MYTH:** Anyone can test.
**FACT:** Testing is a rigorous discipline and requires many kinds of skills.

**MYTH:** There is no creativity in testing.
**FACT:** Creativity can be applied when formulating test approaches, when designing tests, and even when executing tests.
Software V & V Myths – and Facts (3)

**MYTH:** Automated testing eliminates the need for manual testing.

**FACT:** 100% test automation cannot be achieved. Manual Testing, to some level, is always necessary.

**MYTH:** When a defect slips, it is the fault of the Testers.

**FACT:** Quality is the responsibility of all members/stakeholders, including developers, of a project.
Software V & V Myths – and Facts (4)

**MYTH:** Software Testing does not offer opportunities for career growth.

**FACT:** Gone are the days when users had to accept whatever product was dished to them; no matter what the quality. With the abundance of competing software and increasingly demanding users, the need for software testers to ensure high quality will continue to grow.

http://softwaretestingfundamentals.com/software-testing-jobs/
Structure of Lecture 09

• Testing Basics
• Testing Levels
• Testing Methods
• Testing Types
• Testing Artefacts
• Metrics
## Testing Levels

<table>
<thead>
<tr>
<th>Level</th>
<th>Definition and Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acceptance Testing (AT)</td>
<td>The level of the software testing process where a system is tested for acceptability. The purpose of AT is to evaluate the system’s compliance with the business requirements and assess whether it is acceptable for delivery.</td>
</tr>
<tr>
<td>System Testing (ST)</td>
<td>The level of the software testing process where a complete, integrated system/software is tested. The purpose of ST is to evaluate the system’s compliance with the specified requirements.</td>
</tr>
<tr>
<td>Integration Testing (IT)</td>
<td>The level of the software testing process where individual units are combined and tested as a group. The purpose of IT is to expose faults in the interaction between integrated units.</td>
</tr>
<tr>
<td>Unit Testing (UT)</td>
<td>The level of the software testing process where individual units/components of a software/system are tested. The purpose of UT is to validate that each unit of the software performs as designed.</td>
</tr>
</tbody>
</table>
Testing Levels

<table>
<thead>
<tr>
<th>Level</th>
<th>Who and How?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acceptance Testing (AT)</td>
<td>Who: Product Management, Sales, Customer Support, Customers&lt;br&gt;How: Usually, Black Box Testing method is used; often the testing is done ad-hoc and non-scripted</td>
</tr>
<tr>
<td>System Testing (ST)</td>
<td>Who: Normally, independent Testers perform System Testing&lt;br&gt;How: Usually, Black Box Testing method is used</td>
</tr>
<tr>
<td>Integration Testing (IT)</td>
<td>Who: Either Developers themselves or independent Testers&lt;br&gt;How:&lt;br&gt;- Any of Black Box, White Box, and Gray Box Testing methods can be used&lt;br&gt;- Test drivers and test stubs are used to assist in Integration Testing.</td>
</tr>
<tr>
<td>Unit Testing (UT)</td>
<td>Who: Developers&lt;br&gt;How:&lt;br&gt;- White-Box Testing Method&lt;br&gt;- UT frameworks (e.g., jUnit), drivers, stubs, and mock/fake objects are used</td>
</tr>
</tbody>
</table>
Unit Testing

(c) 2008 Mauro Pezzè & Michal Young

Tool example: JUnit

Tool example: JMockIt
Random Class Testing

```java
import org.junit.*;
import static org.junit.Assert.*;
import java.util.*;

public class JunitTest1 {
    private Collection collection;

    @BeforeClass
    public static void oneTimeSetUp() {
        // one-time initialization code
        System.out.println("@BeforeClass - oneTimeSetUp");
    }

    @AfterClass
    public static void oneTimeTearDown() {
        // one-time cleanup code
        System.out.println("@AfterClass - oneTimeTearDown");
    }

    @Before
    public void setUp() {
        collection = new ArrayList();
        System.out.println("@Before - setUp");
    }

    @After
    public void tearDown() {
        collection.clear();
        System.out.println("@After - tearDown");
    }

    @Test
    public void testEmptyCollection() {
        assertTrue(collection.isEmpty());
        System.out.println("@Test - testEmptyCollection");
    }

    @Test
    public void testOneItemCollection() {
        collection.add("itemA");
        assertEquals(1, collection.size());
        System.out.println("@Test - testOneItemCollection");
    }
}
```
import org.junit.*;
import static org.junit.Assert.*;
import java.util.*;

public class JunitTest1 {
    private Collection collection;

    @BeforeClass
    public static void oneTimeSetUp() {
        // one-time initialization code
        System.out.println("@BeforeClass - oneTimeSetUp");
    }

    @AfterClass
    public static void oneTimeTearDown() {
        // one-time cleanup code
        System.out.println("@AfterClass - oneTimeTearDown");
    }

    @Before
    public void setUp() {
        collection = new ArrayList();
        System.out.println("@Before - setUp");
    }

    @After
    public void tearDown() {
        collection.clear();
        System.out.println("@After - tearDown");
    }

    @Test
    public void testEmptyCollection() {
        assertTrue(collection.isEmpty());
        System.out.println("@Test - testEmptyCollection");
    }

    @Test
    public void testOneItemCollection() {
        collection.add("itemA");
        assertEquals(1, collection.size());
        System.out.println("@Test - testOneItemCollection");
    }
}
Integration Testing

Approaches

• **Big Bang**
  • All or most of the units are combined together and tested at one go.
  • This approach is taken when the testing team receives the entire software in a bundle.

• **Top Down**
  • Top level units are tested first and lower level units are tested step by step after that.
  • Test Stubs are needed to simulate lower level units which may not be available during the initial phases.

• **Bottom Up**
  • Bottom level units are tested first and upper level units step by step after that.
  • Test Drivers are needed to simulate higher level units which may not be available during the initial phases.

• **Sandwich/Hybrid**
  • A combination of Top Down and Bottom Up approaches.
System Testing

- The process of testing an integrated system to verify that it meets specified requirements.
Acceptance Testing

- **Internal Acceptance Testing** (Also known as **Alpha Testing**) is performed by members of the organization that developed the software but who are not directly involved in the project (Development or Testing). Usually, it is the members of Product Management, Sales and/or Customer Support.

- **External Acceptance Testing** is performed by people who are not employees of the organization that developed the software.
  - **Customer Acceptance Testing** is performed by the customers of the organization that developed the software. They are the ones who asked the organization to develop the software.
  - **User Acceptance Testing** (Also known as **Beta Testing**) is performed by the end users of the software. They can be the customers themselves or the customers’ customers.
Structure of Lecture 09

- Testing Basics
- Testing Levels
- Testing Methods
- Testing Types
- Testing Artefacts
- Metrics
Black Box Testing (BBT)

• BBT, also known as Behavioral Testing, is a software testing method in which the internal structure/design/implementation of the item being tested is not known to the tester.
  • These tests can be functional or non-functional

Aims to detect these types of issues:
- Incorrect or missing functions
- (User) Interface problems
- Problems in data structures or external database access
- Behavior or performance problems
- Initialization and termination problems
BBT Techniques

Following are some techniques that can be used for designing black box tests

- **Equivalence Class Partitioning**: It is a software test design technique that involves dividing input values into valid and invalid partitions and selecting representative values from each partition as test data.

- **Boundary Value Analysis**: It is a software test design technique that involves determination of boundaries for input values and selecting values that are at the boundaries and just inside/outside of the boundaries as test data.
Example

Assume a ’magic’ Function M

Spec:
The program accepts integers x & y
The program calculates sum = x + y
The program displays the result ‘sum’

M (x, y) → sum = x + y
with x, y: int (32 bit)

Exhaustive testing:
$2^{32} \times 2^{32}$
$= 2^{64} \sim 1.8 \times 10^{19}$ test cases (input data + expected output)
Equivalence Class Partitioning (ECP)

Input: x & y | Output: sum

Classes

- C1: InputX: [MinInt, MaxInt]
- C2: InputY: [MinInt, MaxInt]
- C3: OutputSum: [MinInt, MaxInt]
- C4: InputX: notInt
- C5: InputY: notInt
- C6: OutputSum: exception

Test Cases

Data: x, y, sum

- TC1: 0, 0 -> 0
- TC2: notInt, 0 -> WrongInputException
- TC3: 0, notInt -> WrongInputException

minimal, TCs cover all classes

If we consider output=exception to be an error message caused by invalid input (notInt), then it's good practice to check for the effect of each invalid input class independently.
**Equivalence Class Partitioning (ECP)**

Input: x & y | Output: sum

**Test Cases**

Data: x, y, sum

TC1: 0, 0 -> 0  
TC2: notInt, 0 -> WrongInputException  
TC3: 0, notInt -> WrongInputException

**Classes covered:**

C1, C2, C3  
C4, C2, C6  
C1, C5, C6

minimal,  
TCs cover all classes
Boundary Value Analysis (BVA)
Input: x & y | Output: sum

Classes
C1: InputX: [MinInt, MaxInt]
C2: InputY: [MinInt, MaxInt]
C3: OutputSum: [MinInt, MaxInt]

C4: InputX: notInt
C5: InputY: notInt
C6: OutputSum: exception

Test Cases

Data: x, y, sum

TC1: 0, 0 -> 0
TC2: notInt, 0 -> WrongInputException
TC3: 0, notInt -> WrongInputException

TC4: MinInt, MinInt -> ArithmeticException
TC5: MaxInt, MaxInt -> ArithmeticException
TC6: MaxInt/2, MaxInt/2 -> MaxInt
TC7: MinInt/2, MinInt/2 -> MinInt
Other BBT Methods

• Cause Effect Graphing (CEG)
• Combinatorial Testing
• Exploratory Testing
• Model-driven Testing
• …
BBT – Advantages & Disadvantages

Advantages
• Tests are done from a user’s point of view and will help in exposing discrepancies in the specifications.
• Tester need not know programming languages or how the software has been implemented.
• Tests can be conducted by a body independent from the developers, allowing for an objective perspective and the avoidance of developer-bias.
• Test cases can be designed as soon as the specifications are complete.

Disadvantages
• Only a small number of possible inputs can be tested and many program paths will be left untested.
• Without clear specifications, which is the situation in many projects, test cases will be difficult to design.
• Tests can be redundant if the software designer/developer has already run a test case.
• Ever wondered why a soothsayer closes the eyes when foretelling events? So is almost the case in Black Box Testing.
White Box Testing (WBT)

- WBT, also known as Clear Box Testing, Open Box Testing, Glass Box Testing, Transparent Box Testing, Code-Based Testing or Structural Testing) is a testing method the internal structure/design/implementation of the item being tested is known to the tester.
- The tester chooses inputs to exercise paths through the code and determines the appropriate outputs. Programming know-how and the implementation knowledge is essential.

<table>
<thead>
<tr>
<th>Aims to detect these types of issues:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Control flow issues</td>
</tr>
<tr>
<td>- Data flow issues</td>
</tr>
<tr>
<td>- Algorithms</td>
</tr>
</tbody>
</table>
Example

Assume a 'magic' Function $M$

$M(x, y) \rightarrow \text{sum} = x + y$
with $x, y: \text{int (32 bit)}$

Possible approaches:
- Execute each statement
- Execute paths based on:
  - Control-flow (decisions, conditions, loops, independent paths, etc.)
  - Data-flow (definition and usage of variables)
- Read (review) code

White Box

...$
\text{if } (x - 100 \leq 0) \text{ }
\text{if } (y - 100 \leq 0) \text{ }
\text{if } (x + y - 200 == 0) \text{ }
\text{crash()};$

print($x + y$)

How many statements covered with BBT suite?
Control-Flow Testing: Statement Coverage

- If we try to cover all statements, we must find input data such that all three if-statements are ‘true’:

Traverse code and combine conditions:
(x<=100) and (y<=100) and (x+y=200) ->
(200-y<=100) and (y<=100)
(y>=100) and (y<=100)
y = 100
x = 100

White Box

... if ( x - 100 <= 0 )
    if ( y - 100 <= 0 )
        if ( x + y - 200 == 0 )
            crash();
print(x + y)
Control-Flow Testing: Statement Coverage

Assume a ‘magic’ Function M

M (x, y) \rightarrow \text{sum} = x + y
with x, y: \text{int (32 bit)}

TC1: M(0, 0) \rightarrow 0
TC2: M(100, 100) \rightarrow \text{crash()}

1st if = true: x <= 100
2nd if = true: y <= 100
3rd if = true: x + y = 200

\rightarrow 100\% \text{ Statement Coverage}

White Box

\ldots

if ( x - 100 <= 0 )
  if ( y - 100 <= 0 )
    if ( x + y - 200 == 0 )
      \text{crash()};

print(x + y)
Control-Flow Testing: Branch Coverage

- If we try to cover all branches, we must find input data such that all three if-statements are once evaluated to ’true’ and once to ’false’:

  TC1: M(0, 0) -> 0
  TC2: M(100, 100) -> crash()

  1st if = true: x <= 100
  2nd if = true: y <= 100
  3rd if = true: x + y = 200

  How much branch coverage?
Control-Flow Testing: Branch Coverage

Assume a 'magic' Function M

TC1: M(0, 0) -> 0
TC2: M(100, 100) -> crash()

-> 66% Branch (or Decision) Coverage

TC1: if1 = true / if2 = true / if3 = true
TC2: if1 = true / if2 = true / if3 = false

Missing: if1=false and if2=false -> 2 additional TCs needed
Other WBT Methods

• Data Flow Testing
• Mutation Testing
WBT – Advantages & Disadvantages

Advantages
• Testing can be commenced at an earlier stage. One need not wait for the GUI to be available.
• Testing is more thorough, with the possibility of covering most paths.

Disadvantages
• Since tests can be very complex, highly skilled resources are required, with thorough knowledge of programming and implementation.
• Test script maintenance can be a burden if the implementation changes too frequently.
Other Testing Methods
Other Testing Methods

Gray Box Testing
• Combines BBT and WBT

Ad-hoc Testing (also known as Random Testing or Monkey Testing)
• A testing method without planning (and usually without documentation)
• The tests are conducted informally and randomly without any formal expected results
• Success relies highly on skills and creativity of the testers

Agile Testing
• Context: Test-Driven Development (TDD) and Behavior Driven Development (BDD)
• Exploratory Testing
TDD and BDD

- Developer TDD => Unit Tests
- Acceptance TDD => Acceptance Tests
  also called: Behavior-driven testing (BDD)
Structure of Lecture 09

• Testing Basics
• Testing Levels
• Testing Methods
• Testing Types
• Testing Artefacts
• Metrics
Overview of Testing Types

• Smoke Testing
• Functional Testing
• Usability Testing
• Security Testing
• Performance Testing
• Regression Testing
• Compliance Testing
Smoke Testing

• Smoke Testing, also known as “Build Verification Testing”, is a type of software testing that covers most of the major functions of the software but none of them in depth.

• The results of this testing is used to decide if a build is stable enough to proceed with further testing.
  • If the smoke test passes, go ahead with further testing.
  • If it fails, halt further tests and ask for a new build with the required fixes.
  • If an application is badly broken, detailed testing might be a waste of time and effort.
Functional Testing

- Functional Testing is a type of software testing whereby the system is tested against the functional requirements/specifications.
- This type of testing is not concerned with how processing occurs, but rather, with the results of processing.

- Functional testing is normally performed during the levels of System and Acceptance Testing
- During functional testing, BBT techniques are used

- Typically, functional testing involves the following steps:
  1. Identify functions that the software is expected to perform
  2. Create input data based on the function’s specifications
  3. Determine the output based on the function’s specifications
  4. Execute the test case
  5. Compare the actual and expected outputs
Usability Testing

• Usability Testing is a type of testing done from an end-user’s perspective to determine if the system is easily usable

• Usability Testing is normally performed during System and Acceptance Testing levels

• Tips:
  • Understand who the users of the system are.
  • Understand what their business needs are.
  • Try to mimic their behavior.
Usability Testing – Example of a Test Task

Let’s say a user needs to print a Financial Update Report, every 30 minutes, and he/she has to go through the following steps:

1. Login to the system
2. Click Reports
3. From the groups of reports, select Financial Reports
4. From the list of financial reports, select Financial Update Report
5. Specify the following parameters
   • Date Range, Time Zone, Departments, Units
6. Click Generate Report
7. Click Print
8. Select an option
   • Print as PDF
   • Print for Real

Good or Bad Usability?
Usability Testing
Security Testing

• Security Testing is a type of software testing that intends to uncover vulnerabilities of the system and determine that its data and resources are protected from possible intruders.

• 4 Focus Areas:
  • **Network security**: This involves looking for vulnerabilities in the network infrastructure (resources and policies).
  • **System software security**: This involves assessing weaknesses in the various software (operating system, database system, and other software) the application depends on.
  • **Client-side application security**: This deals with ensuring that the client (browser or any such tool) cannot be manipulated.
  • **Server-side application security**: This involves making sure that the server code and its technologies are robust enough to fend off any intrusion.
Security Testing

Example of a basic security test:
• Log into the web application.
• Log out of the web application.
• Click the BACK button of the browser (Check if you are asked to log in again or if you are provided the logged-in application.)

• Most types of security testing involve complex steps and out-of-the-box thinking but, sometimes, it is simple tests like the one above that help expose the most severe security risks.
The Open Web Application Security Project (OWASP) is a great resource for software security professionals. Be sure to check out the Testing Guide: https://www.owasp.org/index.php/Category:OWASP_Testing_Project

OWASP Top 10 security threats for 2013 were:
- Injection
- Broken Authentication and Session Management
- Cross-Site Scripting (XSS)
- Insecure Direct Object References
- Security Misconfiguration
- Sensitive Data Exposure
- Missing Function Level Access Control
- Cross-Site Request Forgery (CSRF)
- Using Known Vulnerable Components
- Unvalidated Redirects and Forwards

SELECT * FROM Users WHERE Username='$username' AND Password='$password'

$username = 1' or '1' = '1
$password = 1' or '1' = '1
SELECT * FROM Users WHERE Username='1' OR '1' = '1' AND Password='1' OR '1' = '1'
How to avoid SQL injection vulnerability?

Instead of:

```java
String query = "SELECT * FROM Users WHERE Username= "
    + request.getParameter("username")
    + "AND Password= "
    + request.getParameter("password");

try {
    Statement statement = connection.createStatement();
    ResultSet results = statement.executeQuery(query);
}
```

Which might result in a SQL query string like this:

```
SELECT * FROM Users WHERE Username='1' OR '1' = '1' AND
    Password='1' OR '1' = '1'
```
How to avoid SQL injection vulnerability?

Use ‘prepared statement’:

```java
String username = request.getParameter("username");
String password = request.getParameter("password");

String query = "SELECT * FROM Users WHERE Username= ? AND Password= ? ";

PreparedStatement pstmt = connection.prepareStatement(query);
pstmt.setString( 1, username);
pstmt.setString( 2, password);

ResultSet results = pstmt.executeQuery();
```

Example with Hibernate Query Language (HQL) can be found here: https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet
Performance Testing

- Performance Testing is a type of software testing that intends to determine how a system performs in terms of responsiveness and stability under a certain load.

Types:
- **Load Testing** is a type of performance testing conducted to evaluate the behavior of a system at increasing workload.
- **Stress Testing** is a type of performance testing conducted to evaluate the behavior of a system at or beyond the limits of its anticipated workload.
- **Endurance Testing** is a type of performance testing conducted to evaluate the behavior of a system when a significant workload is given continuously.
- **Spike Testing** is a type of performance testing conducted to evaluate the behavior of a system when the load is suddenly and substantially increased.
Regression Testing

- Regression testing is a type of software testing that intends to ensure that changes (enhancements or defect fixes) to the software have not adversely affected it.

**How much?**

- In an ideal case, a full regression test is desirable but oftentimes there are time/resource constraints. In such cases, it is essential to do an impact analysis of the changes to identify areas of the software that have the highest probability of being affected by the change and that have the highest impact to users in case of malfunction and focus testing around those areas.
Regression Testing – Retest All

• Assumption:
  • Changes may introduce faults anywhere in the code

• BUT: expensive, prohibitive for large systems

• Reuse existing test suite
• Add new tests as needed
• Remove obsolete tests

[Skoglund, Runeson, ISESE05]
Regression Testing – Selective Testing

- Conduct impact analysis
  - Only code impacted by change needs to be retested
  - Select tests that exercise such code

- Add new tests if needed
- Remove obsolete tests
Compliance Testing

- Compliance Testing, also known as conformance testing, regulation testing, standards testing, is a type of testing to determine the compliance of a system with internal or external standards.

Checklist:

<table>
<thead>
<tr>
<th>Lorem ipsum dolor sit amet</th>
<th>✓</th>
</tr>
</thead>
<tbody>
<tr>
<td>consectetur adipiscing elit</td>
<td>✓</td>
</tr>
<tr>
<td>sed do eiusmod tempor incididunt ut labore et dolore magna aliqua</td>
<td>✓</td>
</tr>
<tr>
<td>Ut enim ad minim veniam</td>
<td>✗</td>
</tr>
<tr>
<td>quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat</td>
<td>✓</td>
</tr>
</tbody>
</table>
Structure of Lecture 09

• Testing Basics
• Testing Levels
• Testing Methods
• Testing Types
• Testing Artefacts
• Metrics
Test Documentation

IEEE 829-2008: Standard for Software and System Test Documentation

FIG. 7.4
Test Plan

- A Software Test Plan is a document describing the testing scope and activities. It is the basis for formally testing any software/product in a project.

One can have the following types of test plans:
- **Master Test Plan:** A single high-level test plan for a product that unifies all other test plans.
- **Testing Level Specific Test Plans:**
  - Unit Test Plan
  - Integration Test Plan
  - System Test Plan
  - Acceptance Test Plan
- **Testing Type Specific Test Plans:** Plans for major types of testing like Performance Test Plan and Security Test Plan
# Test Case – Template

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Suite ID</td>
<td>The ID of the test suite to which this test case belongs.</td>
</tr>
<tr>
<td>Test Case ID</td>
<td>The ID of the test case.</td>
</tr>
<tr>
<td>Test Case Summary</td>
<td>The summary / objective of the test case.</td>
</tr>
<tr>
<td>Related Requirement</td>
<td>The ID of the requirement this test case relates/traces to.</td>
</tr>
<tr>
<td>Prerequisites</td>
<td>Any prerequisites or preconditions that must be fulfilled prior to executing the test.</td>
</tr>
<tr>
<td>Test Procedure</td>
<td>Step-by-step procedure to execute the test.</td>
</tr>
<tr>
<td>Test Data</td>
<td>The test data, or links to the test data, that are to be used while conducting the test.</td>
</tr>
<tr>
<td>Expected Result</td>
<td>The expected result of the test.</td>
</tr>
<tr>
<td>Actual Result</td>
<td>The actual result of the test; to be filled after executing the test.</td>
</tr>
<tr>
<td>Status</td>
<td>Pass or Fail. Other statuses can be ‘Not Executed’ if testing is not performed and ‘Blocked’ if testing is blocked.</td>
</tr>
<tr>
<td>Remarks</td>
<td>Any comments on the test case or test execution.</td>
</tr>
<tr>
<td>Created By</td>
<td>The name of the author of the test case.</td>
</tr>
<tr>
<td>Date of Creation</td>
<td>The date of creation of the test case.</td>
</tr>
<tr>
<td>Executed By</td>
<td>The name of the person who executed the test.</td>
</tr>
<tr>
<td>Date of Execution</td>
<td>The date of execution of the test.</td>
</tr>
<tr>
<td>Test Environment</td>
<td>The environment (Hardware/Software/Network) in which the test was executed.</td>
</tr>
</tbody>
</table>
Test Case – Example

Test Suite ID: TS001
Test Case ID: TC001
Test Case Summary: To verify that clicking the Generate Coin button generates coins.
Related Requirement: RS001
Prerequisites: User is authorized; Coin balance is available.

Test Procedure:
Select the coin denomination in the Denomination field.
Enter the number of coins in the Quantity field.
Click Generate Coin.

Test Data:
Denominations: 0.05, 0.10, 0.25, 0.50, 1, 2, 5
Quantities: 0, 1, 5, 10, 20

Expected Result:
Coin of the specified denomination should be produced if the
specified Quantity is valid (1, 5)
A message ‘Please enter a valid quantity between 1 and 10’ should
be displayed if the specified quantity is invalid.

Actual Result:
If the specified quantity is valid, the result is as expected.
If the specified quantity is invalid, nothing happens; the expected
message is not displayed

Status: Fail
Remarks: This is a sample test case.
Created By: John Doe
Date of Creation: 01/14/2020
Executed By: Jane Roe
Date of Execution: 02/16/2020
Test Environment: OS: Windows Y, Browser: Chrome N
Defect Report

- In most companies, a defect reporting tool is used and the elements of a report can vary. However, in general, a defect report can consist of the following elements.
Structure of Lecture 09

- Testing Basics
- Testing Levels
- Testing Methods
- Testing Types
- Testing Artefacts
- Metrics
Code Coverage

- The relative amount of covered items with regards to a coverage criterion, e.g., statement, branch, condition, …

  \[
  \text{statement\_coverage} = \frac{\text{executed\_statements}}{\text{total\_number\_statements}} \\
  \text{branch\_coverage} = \frac{\text{executed\_branches}}{\text{total\_number\_branches}}
  \]

Usage

- To control the comprehensiveness of a test suite
- Often used as test stopping criterion
Defect Density

Defect Density is the number of confirmed defects detected in software/component during a defined period of development/operation divided by the size of the software/component.

\[
\text{defect\_density} = \frac{\text{number\_confirmed\_defects}}{\text{size}}
\]

Usage:

- For comparing the relative number of defects in various software components (or software products) so that high-risk components can be identified and resources focused towards them.
Defect Coverage

• Defect Coverage is the number of confirmed defects detected in software/component during a defined period of development/operation divided by the total number of defects.

\[
defect\text{\_coverage} = \frac{\text{number\_confirmed\_defects}}{\text{total\_number\_defects}}
\]

Usage:
• To assess the effectiveness of a test suite.
• Might be applied for certain types of defects (e.g., severity=major; priority=high)
Next Lecture

- Date/Time:
  - Friday, 04-Nov, 10:15-12:00
- Topic:
  - Industry Guest Lecture 1: *What is QA in embedded software? The Practical Industrial Experience*
  - by Jason Lee, Dolby Laboratories, Australia
- Labs:
  - Go to labs: introduction to homework assignment 5