LTAT.05.003
Software Engineering

Lecture 9:
Verification & Validation (Testing) I

Fall 2020

Dietmar Pfahl
email: dietmar.pfahl@ut.ee
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
Week 06: Continuous Development and Integration
Week 07: Project Estimation / Architecture and Design I
Week 08: Architecture and Design II

Week 09: Verification and Validation I
Week 10: Verification and Validation II
Week 11: Refactoring (and TDD)
Week 12: Agile/Lean Methods
Week 13: Industry Guest Lecture
Week 14: Course wrap-up, review and exam preparation
Week 15: Reserve time slot (no lecture scheduled as of today)
Structure of Lecture 9

- 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 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

Safety?
Software Product Quality Model
– ISO 25010 Standard

Quality in Use Model
* Effectiveness
  Efficiency
  Satisfaction
  Usefulness
  Trust
  Pleasure
  Comfort

Freedom from risk
  Economic risk mitigation
  Health and safety risk mitigation
  Environmental risk mitigation

Context coverage
* Context completeness
  Flexibility

Testability
  Compliance

Software Product Quality

Functional Suitability
  Appropriateness
    Accuracy
    Compliance
Reliability
  Availability
    Fault tolerance
    Recoverability
    Compliance
Performance efficiency
  Time-behaviour
    Resource-utilisation
    Compliance
Operability
  Appropriateness
    recognisability
    Learnability
    Ease of use
    Helpfulness
    Attractiveness
    Technical accessibility
    Compliance
Security
  Confidentiality
    Integrity
    Non-repudiation
    Accountability
    Authenticity
    Compliance
Compat
  Replace
    Co-exist
    Interoperability

Universität Tübingen
Institute of Computer Science
Software Quality Assurance (SQA)

versus

Software Quality Control (SQC)
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 (static/dynamic):
- Reviews (manual/automated)
- Testing (manual/automated)

**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 versus Validation
Validation versus Verification

Requirements Backlog
(e.g., User Stories)

Work Product
Work Product
...

Development Process

End Product
(i.e., the Software that is delivered/deployed)
Validation versus Verification

Requirements Backlog
(e.g., User Stories)

Work Product
Work Product
...
Work Product

Development Process

End Product
(i.e., the Software that is delivered/deployed)
Validation versus Verification

Requirements Backlog (e.g., User Stories)

Work Product

Work Product

Work Product

... Development Process

End Product (i.e., the Software that is delivered/deployed)
Verification

Definition
• The process of evaluating work-products (not the final end 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)

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 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.

http://SoftwareTestingFundamentals.com/
# 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

User review of external behavior as it is determined or becomes visible
More vocabulary …
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 and 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 passes or fails  
  (→ ‘fail’ happens if actual output is different from expected output)

**Test Verdict** = decision of whether a test has passed or failed

---

<table>
<thead>
<tr>
<th>ID</th>
<th>Condition to be tested</th>
<th>Execution condition</th>
<th>Test data</th>
<th>Expected result</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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. Try 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, …
JUnit

@Test
public void shortRegularRental() {
    Customer customer = new Customer("Cust");
    Movie movie = new Movie("Groundhog Day", REGULAR);
    Rental rental = new Rental(movie, 2);
    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());
}

Sikuli

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

Input

Expected Output

Oracle
What is a ‘Bug’ in SE?
First ‘Computer Bug’ in 1947

The term "bug" was used in an account by computer pioneer Grace Hopper, who publicized the cause of a malfunction in an early electromechanical computer. (Harvard’s Mark II relay computer)

Source: https://en.wikipedia.org/wiki/Software_bug
What is a Bug in SE?
What is a Bug in SE?

```
if amountOf(baby) > 1
answer = "Twins";
print(answer);
...
```
What is a Bug in SE?

What's the plural of baby? Twins

Failure?

Fault?

```
... if amountOf(baby) > 1
    answer = "Twins";
    print(answer);
...```

What is a Bug in SE?

Fault?

```java
... if amountOf(baby) > 1
    answer = "Twins";
if equals(baby, babyl)
    answer = "Twins";
print(answer);
...```

Failure?

What's the plural of baby?

Twins
What is a Bug in SE?

Error?

Fault?

... if amountOf(baby) > 1
   answer = "Twins";
if equals(baby, baby1)
   answer = "Twins";
print(answer);
...
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:

```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;
}
```

Inputs:
- Correct (=Expected) result?
  - x = [2,7,0]
  - x = [0,7,2]

Actual result?
- x = [0,7,2]

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;
}
```

Inputs:
- Correct (=Expected) result? ?
- Actual result? ?

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;
}
```

Inputs:

- Correct (=Expected) result? 1
- Actual result? ?

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;
}
```

Inputs:

- Correct (=Expected) result? 1
- Actual result? 1

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;
}
```

Inputs:               Correct (=Expected) result?

x = [2,7,0]            Actual result?    1
Fault? Error? Failure? 1

Program state: x, i, count, PC

? / ? / No
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;
}
```

Inputs:
- Correct (=Expected) result? 1
- Actual result? 1
- Fault? Error? Failure? Yes / ? / No

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;
}
```

Inputs:
- x = [2, 7, 0]
- Correct (=Expected) result? 1
- Actual result? 1
- Fault? Error? Failure? Yes / Yes / No

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;
}
```

Inputs:

- Correct (=Expected) result?  ?
- Actual result?  ?

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;
}
```

Inputs:

- Correct (=Expected) result? 1
- Actual result? 0

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;
}
```

Inputs:

<table>
<thead>
<tr>
<th>Correct (=Expected) result?</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual result?</td>
<td>0</td>
</tr>
</tbody>
</table>

x = [0,7,2]

Fault? Error? Failure?

Yes / Yes / Yes

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

We have seen …

Fault = Yes
Error = Yes

------------------

Failure = No
or
Failure = Yes
Definition 2: Error – Fault – Failure

Could any of this happen?

Fault = No
Error = No
--------
Failure = No
or
Failure = Yes

Fault = No
Error = Yes
--------
Failure = No
or
Failure = Yes

Fault = Yes
Error = No
--------
Failure = No
or
Failure = Yes
Definition 2: Error – Fault – Failure

Could any of this happen?

Fault = No  Fault = No  Fault = Yes
Error = No   Error = Yes   Error = No
-------------  -------------  -------------
Failure = No  Failure = No  Failure = No
or            or            or
Failure = Yes Failure = Yes Failure = Yes
Definition 2: Error – Fault – Failure

Example:

```java
public static int numZero (int[] x) {
    // Effects: if x==null throw NullPointerException
    //          else return the index of the 1st occurrence of 0 in x

    for (int i = 0; i < x.length-1; i++) {
        if (x[i] == 0) {
            return i;
        }
    }
    return -1;
}
```

Inputs:  Correct (=Expected) result?  ?
         Actual result?  ?


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 index of the 1st occurrence of 0 in x

    for (int i = 0; i < x.length-1; i++) {
        if (x[i] == 0) {
            return i;
        }
    }
    return -1;
}
```

Inputs:
- Correct (=Expected) result? 0
- Actual result? 0
- x = [0,7,2]

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 index of the 1st occurrence of 0 in x
    for (int i = 0; i < x.length-1; i++) {
        if (x[i] == 0) {
            return i;
        }
    }
    return -1;
}
```

Inputs: Correct (=Expected) result? 0
        Actual result? 0
        Fault? Error? Failure? Yes / ? / No

x = [0,7,2]
Definition 2: Error – Fault – Failure

Example:

```java
public static int numZero (int[] x) {
    // Effects: if x==null throw NullPointerException
    // else return the index of the 1st occurrence of 0 in x

    for (int i = 0; i < x.length-1; i++) {
        if (x[i] == 0) {
            return i;
        }
    }
    return -1;
}
```

Inputs:  
- Correct (=Expected) result? 0
- Actual result? 0
- Fault? Error? Failure? Yes / ? / No

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

Summary of the four possible situations:

Fault = No  Error = No  Failure = No
Fault = Yes  Error = No  Failure = Yes
Fault = Yes  Error = Yes  Failure = Yes
Fault = Yes  Error = Yes  Failure = No or Failure = No

Definition 2: Error – Fault – Failure

Summary of the four possible situations:

Fault = No
Error = No
Failure = No

Fault = Yes
Error = No
Failure = No

Fault = Yes
Error = Yes
Failure = No
or
Failure = Yes

Fault = Yes
Error = No
Failure = No
Definition 2: Error – Fault – Failure

Summary of the four possible situations:

Fault = No
Error = No
Failure = No

Fault = Yes
Error = No
Failure = No

Fault = Yes
Error = Yes
Failure = Yes or No

No failure when testing does not imply no fault!
Other often used terms for failure …

**Incident**
- Often used when something suspicious has happened but it is not yet clear what it is
- It is a symptom that something is wrong
- Alerts the tester or user that a failure might come

**Issue**
- Often used in a broad sense stating that something is going on but without making claims about where it comes from, if it is a failure due to some fault, or whether it should be fixed.
- Sometimes even referring to tasks or to requirements (e.g., in issue trackers)
Flaky Test

A test that sometime passes and sometimes fails due to environmental conditions that cannot be (fully) controlled during testing

Typical Causes:
• Cached data, order of tests (no proper cleanup), assertion timing inconsistent with global state, interaction with (unstable or buggy) 3rd party system, time bomb (accessing local time inconsistently)

Tips on how to deal with it:
• https://hackernoon.com/flaky-tests-a-war-that-never-ends-9aa32fdef359
Structure of Lecture 9

• Testing Basics
• Testing Levels
• Testing Methods
• Testing Types
• Testing Artefacts
• Metrics
STLC integrated with SDLC

Actual Needs and Constraints → User Acceptance (alpha, beta test) → Delivered Package

Review → System Specifications → System Test → Analysis / Review

System Specifications → Integration Test

Subsystem Design/Specs → Integration Test

Module Test → Subsystem

Analysis / Review → Subsystem

Unit/ Component Specs → Module Test

User review of external behavior as it is determined or becomes visible
# 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>
</table>
How: Usually, Black Box Testing method is used; often the testing is done ad-hoc and non-scripted |
How: Usually, Black Box Testing method is used |
| Integration Testing (IT)   | Who: Either Developers themselves or independent Testers  
How:  
- Any of Black Box, White Box, and Gray Box Testing methods can be used  
- Test drivers and test stubs are used to assist in Integration Testing. |
| Unit Testing (UT)          | Who: Developers  
How:  
- White-Box Testing Method  
- UT frameworks (e.g., jUnit), drivers, stubs, and mock(fake) objects are used |
Unit Testing

Driver

Classes to be tested

Stubs

Tool example: JUnit

Tool example: JMockIt

(c) 2008 Mauro Pezzè & Michal Young
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");
    }
}

JUnit4
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");
    }
}

JUnit4
From JUnit4 to JUnit5 – JDK Versions

- JUnit4 requires Java 5 or higher
- JUnit5 requires Java 8 or higher
## From JUnit4 to JUnit5 - Annotations

<table>
<thead>
<tr>
<th>FEATURE</th>
<th>JUNIT 4</th>
<th>JUNIT 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declare a test method</td>
<td>@Test</td>
<td>@Test</td>
</tr>
<tr>
<td>Execute before all test methods in the current class</td>
<td>@BeforeClass</td>
<td>@BeforeAll</td>
</tr>
<tr>
<td>Execute after all test methods in the current class</td>
<td>@AfterClass</td>
<td>@AfterAll</td>
</tr>
<tr>
<td>Execute before each test method</td>
<td>@Before</td>
<td>@BeforeEach</td>
</tr>
<tr>
<td>Execute after each test method</td>
<td>@After</td>
<td>@AfterEach</td>
</tr>
<tr>
<td>Disable a test method / class</td>
<td>@Ignore</td>
<td>@Disabled</td>
</tr>
<tr>
<td>Test factory for dynamic tests</td>
<td>NA</td>
<td>@TestFactory</td>
</tr>
<tr>
<td>Nested tests</td>
<td>NA</td>
<td>@Nested</td>
</tr>
<tr>
<td>Tagging and filtering</td>
<td>@Category</td>
<td>@Tag</td>
</tr>
<tr>
<td>Register custom extensions</td>
<td>NA</td>
<td>@ExtendWith</td>
</tr>
</tbody>
</table>
class CalculatorTest {
    Calculator calc;
    @BeforeAll
    static void start() {
        System.out.println("inside @BeforeAll");
    }
    @BeforeEach
    void init() {
        System.out.println("inside @BeforeEach");
        calc = new Calculator();
    }
    @Test
    void additionTest() {
        System.out.println("inside additionTest");
        assertAll(
            () -> assertEquals(2, calc.add(1,1), "Doesn't add two positive numbers properly"),
            () -> assertEquals(0, calc.add(-1,1), "Doesn't add a negative and a positive number properly"),
            () -> assertNotNull(calc, "The calc variable should be initialized")
        );
    }
    @Test
    void divisionTest() {
        System.out.println("inside divisionTest");
        assertThrows(ArithmeticException.class, () -> calc.divide(2,0));
    }
    @AfterEach
    void afterEach() {
        System.out.println("inside @AfterEach");
    }
    @AfterAll
    static void close() {
        System.out.println("inside @AfterAll");
    }
}
class CalculatorTest {

    Calculator calc;

    @BeforeAll
    static void start() {
        System.out.println("inside @BeforeAll");
    }

    @BeforeEach
    void init() {
        System.out.println("inside @BeforeEach");
        calc = new Calculator();
    }

    @Test
    void additionTest() {
        System.out.println("inside additionTest");
        assertAll(
            () -> assertEquals(2, calc.add(1,1), "Doesn't add two positive numbers properly"),
            () -> assertEquals(0, calc.add(-1,1), "Doesn't add a negative and a positive number properly"),
            () -> assertNotNull(calc, "The calc variable should be initialized")
        );
    }

    @Test
    void divisionTest() {
        System.out.println("inside divisionTest");
        assertThrows(ArithmeticException.class, () -> calc.divide(2,0));
    }

    @AfterEach
    void afterEach() {
        System.out.println("inside @AfterEach");
    }

    @AfterAll
    static void close() {
        System.out.println("inside @AfterAll");
    }
}

OBS: Despite the additionTest() method being declared first, it is not guaranteed that it will be executed first.

Console Output:
inside @BeforeAll
inside @BeforeEach
inside divisionTest
inside @AfterEach
inside @BeforeEach
inside additionTest
inside @AfterEach
class CalculatorTest {
    Calculator calc;
    @BeforeAll
    static void start() {
        System.out.println("inside @BeforeAll");
    }
    @BeforeEach
    void init() {
        System.out.println("inside @BeforeEach");
        calc = new Calculator();
    }
    @Test
    void additionTest() {
        System.out.println("inside additionTest");
        assertAll(
            () -> assertEquals(2, calc.add(1,1), "Doesn't add two positive numbers properly"),
            () -> assertEquals(0, calc.add(-1,1), "Doesn't add a negative and a positive number properly"),
            () -> assertNotNull(calc, "The calc variable should be initialized")
        );
    }
    @Test
    void divisionTest() {
        System.out.println("inside divisionTest");
        assertThrows(ArithmeticException.class, () -> calc.divide(2,0));
    }
    @AfterEach
    void afterEach() {
        System.out.println("inside @AfterEach");
    }
    @AfterAll
    static void close() {
        System.out.println("inside @AfterAll");
    }
}
class CalculatorTest {
    Calculator calc;
    @BeforeAll
    static void start() {
        System.out.println("inside @BeforeAll");
    }
    @BeforeEach
    void init() {
        System.out.println("inside @BeforeEach");
        calc = new Calculator();
    }
    @Test
    void additionTest() {
        System.out.println("inside additionTest");
        assertAll(
            () -> assertEquals(2, calc.add(1,1), "Doesn't add two positive numbers properly"),
            () -> assertEquals(0, calc.add(-1,1), "Doesn't add a negative and a positive number properly"),
            () -> assertNotSame(calc, "The calc variable should be initialized")
        );
    }
    @Test
    void divisionTest() {
        System.out.println("inside divisionTest");
        assertThrows(ArithmeticException.class, () -> calc.divide(2,0));
    }
    @AfterEach
    void afterEach() {
        System.out.println("inside @AfterEach");
    }
    @AfterAll
    static void close() {
        System.out.println("inside @AfterAll");
    }
}

JUnit5 – Test Life-Cycle

Must have void return type
Must be static
Must not be private

Console Output:
inside @BeforeAll
inside @BeforeEach
inside divisionTest
inside @AfterEach
inside @BeforeEach
inside additionTest
inside @AfterEach
inside @AfterAll
What unit tests to write?

For CalculatorApp:

```java
@Test
void additionTest() {
    System.out.println("inside additionTest");
    assertAll(
        () -> assertEquals(2, calc.add(1,1), "Doesn't add two positive numbers properly"),
        () -> assertEquals(0, calc.add(-1,1), "Doesn't add a negative and a positive number properly"),
        () -> assertNotNull(calc, "The calc variable should be initialized")
    );
}

@Test
void divisionTest() {
    System.out.println("inside divisionTest");
    assertThrows(ArithmeticException.class, () -> calc.divide(2,0));
}
```
What unit tests to write?

For CalculatorApp:

```java
@Test
generic additionTest() {
    System.out.println("inside additionTest");
    assertAll(
        () -> assertEquals(2, calc.add(1,1), "Doesn't add two positive numbers properly"),
        () -> assertEquals(0, calc.add(-1,1), "Doesn't add a negative and a positive number properly"),
        () -> assertNotNull(calc, "The calc variable should be initialized")
    );
}

@Test
generic divisionTest() {
    System.out.println("inside divisionTest");
    assertThrows(ArithmeticException.class, () -> calc.divide(2,0));
}
```

Why not check this:

(-1) + (-1)
0 + 0
???

2 / 1
1 / 2
-1 / 0
???
JUnit5

Many examples:
https://howtodoinjava.com/junit-5-tutorial/

- How to use Assertions
- How to test Exceptions
- …
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.

Fig. 6.10
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.
Next Lecture

• Date/Time:
  • Friday, 06-Nov, 10:15-12:00

• Topic:
  • Verification and Validation (Testing) II

• Labs:
  • Submit homework assignment 4
  • Start working on homework assignment 5