Lecture 06: Test Levels / BDD & Behavior Testing / GUI Testing / Visual Testing

Dietmar Pfahl
email: dietmar.pfahl@ut.ee

Spring 2019
Lectures (J. Liivi 2-111)

- Lecture 1 (14.02) – Introduction to Software Testing
- Lecture 2 (21.02) – Basic Black-Box Testing Techniques
- Lecture 3 (28.02) – BBT advanced: Combinatorial Testing
- Lecture 4 (07.03) – Basic White-Box Testing Techniques
- Lecture 5 (14.03) – Test Lifecycle, Test Tools, Test Automation
- Lecture 7 (28.03) – BBT advanced: State-Transition Testing & Exploratory Testing
- Lecture 8 (04.04) – BBT advanced: Security, Usability and A/B Testing
- Lecture 9 (11.04) – WBT advanced: Data-Flow Testing / Mutation Testing
- Lecture 11 (25.04) – Defect Estimation / Test Documentation, Organisation and Process Improvement (Test Maturity Model)
- 02.05 - no lecture
- Lecture 12 (09.05) – Industry Guest Lecture (to be announced)
- Lecture 13 (16.05) – Exam Preparation
Structure of Lecture 6

• Test Levels
• Lab 6 = Bonus Lab
• BDD and Behavior Testing
• GUI Testing
• Visual Testing
V-model

- Requirements
  - Functional specification
  - Architecture design
    - Module design
      - Coding
  - Test
    - Unit testing
      - Integration testing
        - System testing
          - Acceptance testing
  - Build
Test Levels

Unit/Component Level

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

Driver

Classes to be tested

Tool example: JUnit

(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");
    }
}
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");
    }
}
```

https://github.com/junit-team/junit4/wiki/test-fixtures
Integration Testing

- More than one (tested) unit
- Detecting defects in …
  - Interfaces of units (APIs)
  - Communication between units
- Helps assembling incrementally a whole system
- Non-functional aspects if possible
- Done by developers/designers or independent testers
  - Preferably developers and testers in collaboration
- Often omitted due to setup difficulties
  - Time is more efficiently spent on unit and system tests
Integration Testing – Procedural

Note: This has nothing to do with class hierarchy in OO (inheritence)
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.
Integration Testing – OO
Integration Testing – OO

What if Class 4 not yet ready or not accessible?

Fig. 6.8
Integration Testing – OO

What if Class 4 not yet ready or not accessible?

What if an external service not yet ready or accessible (or should not be accessed – because we are just testing)?

What if an external service not yet ready or accessible (or should not be accessed – because we are just testing)?

What if Class 4 not yet ready or not accessible?
Integration Testing / Scaffolding

Driver

Classes to be tested

Stubs

Tool example: JUnit

Tool examples: JMockIt, Mockito

(c) 2008 Mauro Pezzè & Michal Young
Mockito

<table>
<thead>
<tr>
<th>RecordService</th>
</tr>
</thead>
<tbody>
<tr>
<td>DatabaseDao database</td>
</tr>
<tr>
<td>NetworkDao network</td>
</tr>
<tr>
<td>saveFile()</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

Example:
Assume a class RecordService which stores a given file in the database and a network location with help of classes DatabaseDao and NetworkDao.
Mockito

RecordService

DatabaseDao database
NetworkDao network
saveFile()

Dependencies

DatabaseDao

save()

NetworkDao

save()
Mockito

RecordService.java

```java
public class RecordService {
    DatabaseDAO database;
    NetworkDAO network;

    //setters and getters

    public boolean saveFile(String fileName) {
        database.save(fileName);
        System.out.println("Saved in database in Main class");
        network.save(fileName);
        System.out.println("Saved in network in Main class");
        return true;
    }
}
```
DatabaseDao.java

```java
public class DatabaseDao {
    public void save(String fileName) {
        System.out.println("Saved in database");
    }
}
```

NetworkDao.java

```java
public class NetworkDao {
    public void save(String fileName) {
        System.out.println("Saved in network");
    }
}
```

In the test environment, it is neither possible to access the database nor the network location ➔ we create mocks for both repositories
Mockito

To test the `RecordService` class, we create a unit test.

```java
import static org.junit.Assert.assertEquals;
import static org.mockito.Mockito.times;
import static org.mockito.Mockito.verify;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.MockitoJUnitRunner;

import com.howtodoinjava.demo.mockito.DatabaseDAO;
import com.howtodoinjava.demo.mockito.NetworkDAO;
import com.howtodoinjava.demo.mockito.RecordService;

@RunWith(MockitoJUnitRunner.class)
public class ApplicationTest {
    @InjectMocks
    RecordService recordService;

    @Mock
    DatabaseDAO databaseMock;

    @Mock
    NetworkDAO networkMock;

    @Test
    public void saveFileTest() {
        boolean saved = recordService.saveFile("temp.txt");
        assertEquals(true, saved);

        verify(databaseMock, times(1)).save("temp.txt");
        verify(networkMock, times(1)).save("temp.txt");
    }
}
```
ApplicationTest.java

... 

@RunWith(MockitoJUnitRunner.class)
public class ApplicationTest
{
    @InjectMocks
    RecordService recordService;

    @Mock
    DatabaseDao databaseMock;

    @Mock
    NetworkDao networkMock;

    @Test
    public void saveFileTest()
    {
        boolean saved = recordService.saveFile("temp.txt");
        assertEquals(true, saved);

        verify(databaseMock, times(1)).save("temp.txt");
        verify(networkMock, times(1)).save("temp.txt");
    }
}
ApplicationTest.java

...@RunWith(MockitoJUnitRunner.class)
public class ApplicationTest
{
    @InjectMocks
    RecordService recordService;

    @Mock
    DatabaseDao databaseMock;

    @Mock
    NetworkDao networkMock;

    @Test
    public void saveFileTest()
    {
        boolean saved = recordService.saveFile("temp.txt");
        assertEquals(true, saved);

        verify(databaseMock, times(1)).save("temp.txt");
        verify(networkMock, times(1)).save("temp.txt");
    }
}
ApplicationTest.java

```java
@RunWith(MockitoJUnitRunner.class)
public class ApplicationTest
{
    @InjectMocks
    RecordService recordService;

    @Mock
    DatabaseDao databaseMock;

    @Mock
    NetworkDao networkMock;

    @Test
    public void saveFileTest()
    {
        boolean saved = recordService.saveFile("temp.txt");
        assertEquals(true, saved);

        verify(databaseMock, times(1)).save("temp.txt");
        verify(networkMock, times(1)).save("temp.txt");
    }
}
```

Annotate the dao fields with the @Mock annotation to have a mock object instantiated for each of them.
@RunWith(MockitoJUnitRunner.class)
public class ApplicationTest
{
    @InjectMocks
    RecordService recordService;

    @Mock
    DatabaseDao databaseMock;

    @Mock
    NetworkDao networkMock;

    @Test
    public void saveFileTest()
    {
        boolean saved = recordService.saveFile("temp.txt");
        assertEquals(true, saved);

        verify(databaseMock, times(1)).save("temp.txt");
        verify(networkMock, times(1)).save("temp.txt");
    }
}
ApplicationTest.java

```java
@RunWith(MockitoJUnitRunner.class)
public class ApplicationTest {

    @InjectMocks
    RecordService recordService;

    @Mock
    DatabaseDao databaseMock;

    @Mock
    NetworkDao networkMock;

    @Test
    public void saveFileTest()
    {
        boolean saved = recordService.saveFile("temp.txt");
        assertEquals(true, saved);

        verify(databaseMock, times(1)).save("temp.txt");
        verify(networkMock, times(1)).save("temp.txt");
    }
}
```

Verify that methods in the mocked objects have been invoked (here: once).
Mockito – verify()

Simple behavior verification ➞ verify(); using List class as an example ...

//Let's import Mockito statically so that the code looks clearer
import static org.mockito.Mockito.*;

//mock creation
List mockedList = mock(List.class);

//using mock object
mockedList.add("one");
mockedList.clear();

//verification
verify(mockedList).add("one");
verify(mockedList).clear();

Once created, a mock will remember all interactions. Then you can selectively verify whatever interactions you are interested in.

Interface java.util.List

Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(int index, Object element)</td>
<td>Inserts the specified element at the specified position in this List (optional operation).</td>
</tr>
<tr>
<td>addObject (Object)</td>
<td>Appends the specified object to the end of this List (optional operation).</td>
</tr>
<tr>
<td>addAll(Collection)</td>
<td>Appends all of the elements in the specified Collection to the end of this List, in the order that they are returned by the specified Collection’s iterator (optional operation).</td>
</tr>
<tr>
<td>addAll(int index, Collection)</td>
<td>Inserts all of the elements in the specified Collection into this List at the specified position (optional operation).</td>
</tr>
<tr>
<td>clear()</td>
<td>Removes all of the elements from this List (optional operation).</td>
</tr>
<tr>
<td>contains(Object)</td>
<td>Returns true if this List contains the specified element.</td>
</tr>
<tr>
<td>containsAllCollection ()</td>
<td>Returns true if this List contains all of the elements of the specified Collection.</td>
</tr>
<tr>
<td>equals(Object)</td>
<td>Compares the specified Object with this List for equality.</td>
</tr>
<tr>
<td>get(index)</td>
<td>Returns the element at the specified position in this List.</td>
</tr>
<tr>
<td>hashCode()</td>
<td>Returns the hash code value for this List.</td>
</tr>
<tr>
<td>indexOf(Object)</td>
<td>Returns the index in this List of the first occurrence of the specified element, or -1 if the List does not contain this element.</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>Returns true if this List contains no elements.</td>
</tr>
<tr>
<td>iterator()</td>
<td>Returns an iterator over the elements of this List in proper sequence.</td>
</tr>
<tr>
<td>listIterator(Object)</td>
<td>Returns the index in this List of the last occurrence of the specified element, or -1 if the List does not contain this element.</td>
</tr>
</tbody>
</table>
Mockito – when ... then

Simple stubbing => “when() ... then()”

//You can mock concrete classes, not just interfaces
LinkedList mockedList = mock(LinkedList.class);

//stubbing: when – then (like if – else)
when(mockedList.get(0)).thenReturn("first");
when(mockedList.get(1)).thenThrow(new RuntimeException());

//following prints "first"
System.out.println(mockedList.get(0));

//following throws runtime exception
System.out.println(mockedList.get(1));

//following prints "null" because get(999) was not stubbed
System.out.println(mockedList.get(999));

By default, for all methods that return a value, a mock will return either null, a primitive/primitive wrapper value, or an empty collection, as appropriate. For example 0 for an int/Integer and false for a boolean/Boolean.
Mockito – when … then

Argument matchers …

//stubbing using built-in anyInt() argument matcher
when(mockedList.get(anyInt())).thenReturn("element");

//stubbing using custom matcher (let's say isValid() returns your own
matcher implementation):
when(mockedList.contains(argThat(isValid()))).thenReturn("element");

//following prints "element"
System.out.println(mockedList.get(999));

//you can also verify using an argument matcher
verify(mockedList).get(anyInt());

Mockito verifies argument values in natural java style: by using an equals() method. Sometimes, when extra flexibility is required then you might use argument matchers; with verify() you check whether a method has been used.

Compare with previous slide!
Mockito

Annotations: https://howtodoinjava.com/mockito/mockito-annotations/

- **@Mock** is used for mock creation. It makes the test class more readable; alternative: Mockito.mock(classToMock).
- **@Spy** is used to create a spy instance; similar to @Mock but creates a real instance of the mocked class and tracks every interaction with it. Can be used instead of spy(Object) method.
- **@InjectMocks** is used to instantiate the tested object automatically and inject all the @Mock or @Spy annotated field dependencies into it (if applicable).
- **@Captor** is used to create an argument captor

To process annotations, “MockitoAnnotations.initMocks(testClass);” must be used at least once. Alternatively, one can use the built-in runner MockitoJUnitRunner or rule MockitoRule (then use @Rule annotation).
Mockito – Annotations: @Mock

The @Mock annotation is used to create and inject mocked instances. We do not create real objects, rather ask mockito to create a mock for the class.

The @Mock annotation is alternative to Mockito.mock(classToMock). They both achieve the same result. Using @Mock is usually considered “cleaner“, as we don’t fill up the tests with boilerplate assignments that all look the same.

Using the @Mock annotation
• allows shorthand creation of objects required for testing.
• minimizes repetitive mock creation code.
• makes the test class more readable.
• makes the verification error easier to read because field name is used to identify the mock.
Mockito – Annotations: @Mock

@Mock Example:

```java
@Mock
HashMap<String, Integer> mockHashMap;

@Test
public void saveTest()
{
    mockHashMap.put("B", 2);

    Mockito.verify(mockHashMap, times(1)).put("B", 2);
    Mockito.verify(mockHashMap, times(0)).get("B");

    assertEquals(0, mockHashMap.size());
}
```

Here we mock the HashMap class. In real tests, we shall be mocking actual application classes.

We put a key-value pair in the map, then we verify that method invocation was performed on the mocked map instance, then we check its size.
Mockito – Annotations: @Mock

@Mock Example:

```java
@Mock
HashMap<String, Integer> mockHashMap;

@Test
public void saveTest() {
    mockHashMap.put("B", 2);

    Mockito.verify(mockHashMap, times(1)).put("B", 2);
    Mockito.verify(mockHashMap, times(0)).get("B");

    assertEquals(0, mockHashMap.size());
}
```

When using @Mock, Mockito creates a bare-bones shell instance of the Class, entirely instrumented to track interactions with it. This is not a real object and does not maintain the state changes to it.

When using @Spy, Mockito creates a real instance of the class and tracks every interactions with it. It maintains the state changes to it.

➡ Next slide
Mockito – Annotations: @Spy

@Spy Example:

@Spy
HashMap<String, Integer> hashMap;

@Test
public void saveTest() {
    hashMap.put("A", 10);

    Mockito.verify(hashMap, times(1)).put("A", 10);
    Mockito.verify(hashMap, times(0)).get("A");

    assertEquals(1, hashMap.size());
    assertEquals(new Integer(10), (Integer) hashMap.get("A"));
}

The @Spy annotation is used to create a real object and spy on that real object. A spy helps to call all the normal methods of the object while still tracking every interaction, just as we would with a mock.

Notice in given example, how the size of map equals 1 because we added one key-value pair to it. We are also able to get back the value added to map using it’s key (not possible in mocked instances).
Mockito – Annotations: @Captor

@Captor Example:

```java
@Mock
HashMap<String, Integer> hashMap;

@Captor
ArgumentCaptor<String> keyCaptor;

@Captor
ArgumentCaptor<Integer> valueCaptor;

@Test
public void saveTest() {
    hashMap.put("A", 10);
    Mockito.verify(hashMap).put(keyCaptor.capture(), valueCaptor.capture());

    assertEquals("A", keyCaptor.getValue());
    assertEquals(new Integer(10), valueCaptor.getValue());
}
```

The @Captor annotation is used to create an ArgumentCaptor instance which is used to capture method argument values for further assertions.

Note that Mockito verifies argument values using the equals() method of the argument class.
Mockito – How to initialize Annotations?

Option 1: Use @RunWith(MockitoJUnitRunner.class) at the top of unit test class.

```java
@RunWith(MockitoJUnitRunner.class)
public class ExampleTest {

    @Mock
    private List list;

    @Test
    public void shouldDoSomething() {
        list.add(100);
    }
}
```
Mockito – How to initialize Annotations?

Option 1: Use MockitoAnnotations.initMocks(this) in @Before method of unit test class.

```java
public class ExampleTest {

    @Mock
    private List list;

    @Before
    public void setup() {
        MockitoAnnotations.initMocks(this);
    }

    @Test
    public void shouldDoSomething() {
        list.add(100);
    }
}
```
Mockito – How to initialize Annotations?

Option 3: Use MockitoJUnit.rule() to create MockitoRule class.

```java
public class ExampleTest {

    @Rule
    public MockitoRule rule = MockitoJUnit.rule();

    @Mock
    private List list;

    @Test
    public void shouldDoSomething() {
        list.add(100);
    }
}
```
Mockito

More possibilities can be found here:

https://static.javadoc.io/org.mockito/mockito-core/2.25.1/org/mockito/Mockito.html#1

For web applications: use MockWebServer to mock the behavior of another web server (➡ lab 6)
Overview of System 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 – What? How?

- **Test Focus**
  - **Understandability**
    - Easy to understand?
  - **Ease of learning**
    - Easy to learn?
  - **Operability**
    - Matches purpose & environment of operation?
    - Ergonomics: color, font, sound,...
  - **Communicativeness**
    - In accordance with psychological characteristics of user?

- **Test Environments**
  - Free form tasks
  - Procedure scripts
  - Paper screens
  - Mock-ups
  - Field trial
Usability Testing – What? How?

- Test Focus
  - Understandability
    - Easy to understand?
  - Ease of learning
    - Easy to learn?
  - Operability
    - Matches purpose & environment of operation?
    - Ergonomics: color, font, sound, ...
  - Communicativeness
    - In accordance with psychological characteristics of user?

- Test Environments
  - Free from tasks
  - Procedure scripts
  - Paper screens
  - Mock-ups
  - Field trial

More in Lecture 8
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

- 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.

More in Lecture 8
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** 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:

| Lorem ipsum dolor sit amet | ✓ |
| consectetur adipiscing elit | ✓ |
| sed do eiusmod tempor incididunt ut labore et dolore magna aliqua | ✓ |
| Ut enim ad minim veniam | ✗ |
| quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat | ✓ |
Structure of Lecture 6

• Test Levels
• Lab 6 = Bonus Lab
• BDD and Behavior Testing
• GUI Testing
• Visual Testing
Lab 6 – Automated Integration Testing

Lab 6 (week 30: Mar 26 & 27) – Integration Testing (2 bonus points: attend & be active)

Lab 6 Instructions & Tools

Submission Deadlines:
- Tuesday Labs: Monday, 01 Apr, 23:59
- Wednesday Labs: Tuesday, 02 Apr, 23:59

- Penalties apply for late delivery: 50% penalty, if submitted up to 24 hours late; 100 penalty, if submitted more than 24 hours late
Lab 6 – Automated Integration Testing

<table>
<thead>
<tr>
<th>MERCHANT</th>
<th>MERCHANT COMMUNICATOR</th>
<th>OPERATOR COMMUNICATOR</th>
<th>OPERATOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requests user payment for product</td>
<td>Interprets request as payment</td>
<td>Requests user to be charged</td>
<td>Tries to charge user based on request</td>
</tr>
<tr>
<td>Gives user access to their paid products</td>
<td>Notifies merchant that user was charged</td>
<td>Sends successful charge callback</td>
<td></td>
</tr>
<tr>
<td>Refuses to give user access to paid content</td>
<td>Notifies merchant that user was not charged</td>
<td>Translates as failed charge callback with reason</td>
<td></td>
</tr>
</tbody>
</table>

Successful charge?

Yes

Translates as OK

No

Sends successful charge callback
Lab 6 – Automated Integration Testing

System under test
(web app)
Lab 6 – Automated Integration Testing

[Diagram showing a process flow with boxes labeled Merchant Communicator, Payment Service, Mock, Operator, and connections between them. The diagram highlights the role of Mock in integration testing.]
Structure of Lecture 6

• Test Levels
• Lab 6 = Bonus Lab
• BDD and Behavior Testing
• GUI Testing
• Visual Testing
Test-Driven Development (TDD)

1. Write a test
2. See it fail
3. Make it run
4. Make it right (refactor)
TDD and BDD

- Developer TDD => Unit Tests

Acceptance TDD => Acceptance Tests
also called: Behavior-driven testing (BDD)
Behavior Testing

Typical Process:
• Business analyst writes a user story
• (Acceptance) tester writes scenarios based on user story
• Business team reviews scenarios
• Test engineer writes the step definitions for the scenario steps
• QA team writes test scripts (to automate the scenarios)
• The test scripts are run, issues analysed and bugs fixed
• The test scripts are run as regression tests
• End user accepts the software if tests pass (acceptance criteria met)
Behavior Testing

Typical Process:

• Business analyst writes a user story
• (Acceptance) tester writes scenarios based on user story
• Business team reviews scenarios
• Test engineer writes the step definitions for the scenario steps
• QA team writes test scripts (to automate the scenarios)
• The test scripts are run, issues analysed and bugs fixed
• The test scripts are run as regression tests
• End user accepts the software if tests pass (acceptance criteria met)

Scenario definition language: Gherkin (DSL)
Scripting language: Java (programming language)
Test automation framework: Cucumber, JBehave
Behavior Testing – Features/Scenarios

Feature = Describes "what" the software shall do (not: "how")

User Story:
As a project manager, I want to know what day is today, so I don’t make planning mistakes

Example/Scenario:
- Given today is Monday
- When I ask whether it is Friday yet
- Then I should get the answer "nope"
Gherkin

- A Domain Specific Language (DSL) that helps non-programmers express requirements (features) in a structured manner
Gherkin – Example

**Feature:** Is it Friday yet?

PMs want to know whether it's Friday

**Scenario:** Monday isn't Friday

*Given* today is Monday

*When* I ask whether it's Friday yet

*Then* I should be told "Nope"
Gherkin – Example

**Feature:** Is it Friday yet?
PMs want to know whether it's Friday yet.

**Scenario:** Monday isn't Friday

*Given* today is Monday
*When* I ask whether it's Friday yet
*Then* I should be told "Nope"

The first line of this file starts with the keyword Feature: followed by a name. Features will be saved in *.feature* files in Cucumber. It's a good idea to use a name similar to the file name.

The second line is a brief description of the feature. Cucumber does not execute this line, it's just documentation.

The fourth line, Scenario: Sunday is not Friday is a scenario, which is a concrete example illustrating how the software should behave.

The last three lines starting with Given, When and Then are the steps of our scenario. This is what Cucumber will execute.
Gherkin – Example

**Feature:** Is it Friday yet?
PMs want to know when it's Fri

**Scenario:** Monday isn't Friday

**Given** today is Monday

**When** I ask whether it's Frid

**Then** I should be told "Nope"

---

The purpose of **Given** steps is to put the system in a known state before the user (or external system) starts interacting with the system (in the **When** steps). Avoid talking about user interaction in givens. If you have worked with use cases, givens are your preconditions.

The purpose of **When** steps is to describe the key action the user performs.

The purpose of **Then** steps is to observe outcomes. The observations should be related to the business value/benefit in your feature description. Thus, it should be related to something visible from the outside (behavior).
Gherkin – Template

**Feature:** Some terse yet descriptive text of what is desired
  In order to realize a named business value
  As an explicit system actor
  I want to gain some beneficial outcome which furthers the goal

**Scenario:** Some determinable business situation
  **Given** some precondition
    **And** some other precondition
  **When** some action by the actor
    **And** some other action
    **And** yet another action
  **Then** some testable outcome is achieved
    **And** something else we can check happens too

**Scenario:** A different situation
  ...

---
Gherkin – Template

**Feature**: Some terse yet descriptive text
In order to realize a named business value
As an explicit system actor
I want to gain some beneficial outcome

**Scenario**: Some determinable business situation
*Given* some precondition
*And* some other precondition
*When* some action by the actor
*And* some other action
*And* yet another action
*Then* some testable outcome is achieved
*And* something else we can check

**Scenario**: A different situation...

Gherkin is a line-oriented language that uses indentation to define structure.
Line endings terminate statements (called steps).
Finally, most lines in Gherkin start with a special keyword:
Gherkin – Another Example using ‘And’

**Feature:** Serve coffee

- In order to earn money
- Customers should be able to buy coffee at all times

**Scenario:** Buy last coffee

- **Given** there are 1 coffees left in the machine
- **And** I have deposited 1 dollar
- **When** I press the coffee button
- **Then** I should be served a coffee
Gherkin – Yet Another Example

Scenario: Wilson posts to his own blog
  Given I am logged in as Wilson
  When I try to post to "Expensive Therapy"
  Then I should see "Your article was published."

Scenario: Wilson fails to post to somebody else's blog
  Given I am logged in as Wilson
  When I try to post to "Greg's anti-tax rants"
  Then I should see "Hey! That's not your blog!"

Scenario: Greg posts to a client's blog
  Given I am logged in as Greg
  When I try to post to "Expensive Therapy"
  Then I should see "Your article was published."
Gherkin – Yet Another Example

Scenario: Wilson posts to his own blog
   Given I am logged in as Wilson
   When I try to post to "Expensive Therapy"
   Then I should see "Your article was published."

Scenario: Wilson fails to post to somebody else's blog
   Given I am logged in as Wilson
   When I try to post to "Greg's anti-tax rants"
   Then I should see "Hey! That's not your blog!"

Scenario: Greg posts to a client's blog
   Given I am logged in as Greg
   When I try to post to "Expensive Therapy"
   Then I should see "Your article was published."

Feature files can contain several scenarios.

Note: Scenarios that are to be used across features should be collected in a common.feature file.
Gherkin – Yet Another Example

Scenario: Eat 5 out of 12
Given there are 12 cucumbers
When I eat 5 cucumbers
Then I should have 7 cucumbers

Scenario: Eat 5 out of 20
Given there are 20 cucumbers
When I eat 5 cucumbers
Then I should have 15 cucumbers

Adding scenarios with just different values is tiresome.
Gherkin – Yet Another Example

Scenario Outline: Eating
  Given there are <start> cucumbers
  When I eat <eat> cucumbers
  Then I should have <left> cucumbers

Examples:
<table>
<thead>
<tr>
<th>start</th>
<th>eat</th>
<th>left</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>20</td>
<td>5</td>
<td>15</td>
</tr>
</tbody>
</table>

Solution:
Use Scenario Outlines instead!
Gherkin – Tables

Scenario:

Given the following people exist:

<table>
<thead>
<tr>
<th>name</th>
<th>email</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aslak</td>
<td><a href="mailto:aslak@email.com">aslak@email.com</a></td>
<td>123</td>
</tr>
<tr>
<td>Joe</td>
<td><a href="mailto:joe@email.com">joe@email.com</a></td>
<td>234</td>
</tr>
<tr>
<td>Bryan</td>
<td><a href="mailto:bryan@email.org">bryan@email.org</a></td>
<td>456</td>
</tr>
</tbody>
</table>

NOT the same as Scenario Outlines!

Tables as arguments to steps are handy for specifying a larger data set - usually as input to a Given or as expected output from a Then.
Cucumber – Gherkin Tutorial

A 10 min tutorial that explains how to:
• Install Cucumber
• Write your first Scenario using the Gherkin syntax
• Write your first step definition in Java
• Run Cucumber
• Learn the basic workflow of Behaviour-Driven Development (BDD)

Can be found here:
https://docs.cucumber.io/guides/10-minute-tutorial/
Cucumber – Gherkin Tutorial

A 10 min tutorial that explains how to:
• Install Cucumber
• Write your first Scenario using the Gherkin syntax
• Write your first step definition in Java
• Run Cucumber
• Learn the basic workflow of Behaviour-Driven Development (BDD)

Can be found here:
https://docs.cucumber.io/guides/10-minute-tutorial/

In Lab 7 you will use Gherkin with Behave and Python instead.
Structure of Lecture 6

• Test Levels
• Lab 6 = Bonus Lab
• BDD and Behavior Testing
• GUI Testing
• Visual Testing
GUITAR

- Source: https://sourceforge.net/projects/guitar/

A Simple Testing Workflow
GUITAR – GUI Ripping

• During ripping, the GUI application is executed automatically; the application’s windows are opened in a depth first manner.

• The Ripper extracts all the widgets and their properties from the GUI.
  – Properties of widgets include basic attributes such as position, color, size, and enabled status.
  – Properties also include information about widgets' events, such as: whether a widget opens a modal or modeless window or a menu, whether a widget closes a window, and whether the widget is a button or an editable text-field.

• The Ripper extracts properties for widgets as well as their containing GUI windows and stores the information in the GUI Tree.
  – For each GUI window, the Ripper first extracts structural information of that window and then executes widgets that invoke other GUI windows.
  – The depth-first traversal terminates when all GUI windows are covered.
GUITAR – GUI Ripping

GUI of an app with two windows

GUI tree with 2 nodes and subset of attributes
GUITAR – Model Conversion

- EFG = Event Flow Graph
GUITAR – Test Case Generation

Applies traversal algorithms for the EFG:

Defined length $L$ of event sequences)

Example ($L=5$):

$e_1 \rightarrow e_2 \rightarrow e_3 \rightarrow e_4 \rightarrow e_5$
# GUITAR – Test Case Execution and Evaluation

<table>
<thead>
<tr>
<th>Fault ID</th>
<th>Summary</th>
<th>Test case</th>
</tr>
</thead>
<tbody>
<tr>
<td>AU₁</td>
<td>FileNotFoundException with invalid input file name for Export Graphic</td>
<td>Expand ‘File’ menu → Click ‘Export Graphic’ submenu → Enter an invalid file name → Click ‘Save’</td>
</tr>
<tr>
<td>AU₂</td>
<td>FileNotFoundException with invalid input file name for Export All Graphic</td>
<td>Expand ‘File’ menu → Click ‘Export All Graphics’ submenu → Enter an invalid file name → Click ‘Save’</td>
</tr>
<tr>
<td>AU₃</td>
<td>An inappropriate exception trace printed out when deleting object with a blank document</td>
<td>Expand ‘Edit’ menu → Click ‘Delete from Model’ submenu</td>
</tr>
<tr>
<td>Jr₁</td>
<td>FileNotFoundException with a non-existing Journal abbreviation file</td>
<td>Expand ‘Option’ menu → Click ‘Manage journal abbreviation’ submenu → Enter an invalid New file name → Click ‘OK’</td>
</tr>
<tr>
<td>Jr₂</td>
<td>MalformedURLException with an invalid Journal abbreviation download URL</td>
<td>Expand the ‘Option’ menu → Open ‘Journal abbreviation’ windows → Click ‘Download’ button → Enter an invalid URL → Click ‘OK’</td>
</tr>
<tr>
<td>Jr₃</td>
<td>NullPointerException with invalid import folder name</td>
<td>Expand ‘Option’ menu → Click ‘Manage custom imports’ submenu → Click ‘Add from folder’ → Enter a non-existing folder path → Click ‘Cancel’</td>
</tr>
<tr>
<td>Jr₄</td>
<td>ZipException with invalid zip file name</td>
<td>Expand ‘Option’ menu → Click ‘Manage custom imports’ submenu → Click ‘Add from jar’ → Enter a non-existing zip file name → Click ‘Select a Zip-archive’</td>
</tr>
</tbody>
</table>

MalformedURLException with an invalid Journal abbreviation download: “CRASH ME”
Testar

- Script-less GUI testing ➔ Lab 8
- More in Lecture 8
Structure of Lecture 6

• Test Levels
• Lab 6 = Bonus Lab
• BDD and Behavior Testing
• GUI Testing
• Visual Testing
Sikuli X

Capture images of the GUI

Then write tests using those images

... e.g., to test email forwarding with Gmail ...
How Sikuli X mimics the user’s behavior:

- Find all the check boxes for emails
- Shift coordinate
- Click on the subject line
- Wait for everything to load
- Select the option Forward in the list of options
- ...
Sikuli X

Sources:


Alexander V. Leonov’s blog:

Recommended Textbook Exercises

• Chapter 6
  – 1, 3, 6, 7, 8, 9, 12
Next Week

• Quiz 6
• Lab 6: Bonus Lab – not mandatory / no homework
  – Automated Integration Testing (Mockito)
  – Possibility of getting 2 bonus points

• Lecture 7:
  – Black-Box Testing (advanced): State-Transition Testing & Exploratory Testing