Software Engineering 2 – Advanced Testing with Java 2

Hochschule für Technik Stuttgart

Marcus Deininger

SS 2021

Topics

Part 1 [last week]

- Introduction
  - Definitions
  - Test Types: Black Box-/Glass Box-Testing
  - Test Workflow / Test Management
- Tools
  - Overview
  - Junit: Quick Introduction / Retake
  - eclEmma: Measuring Coverage
  - Java-Reflection: Access the Inaccessible

Topics

Part 2 [today's part]

- Mockito
  - Test-Doubles
  - Using Mockito
- AspectJ
  - Aspect Oriented Programming
  - AspectJ for Testing
- Selenium
  - Testing Web-Applications

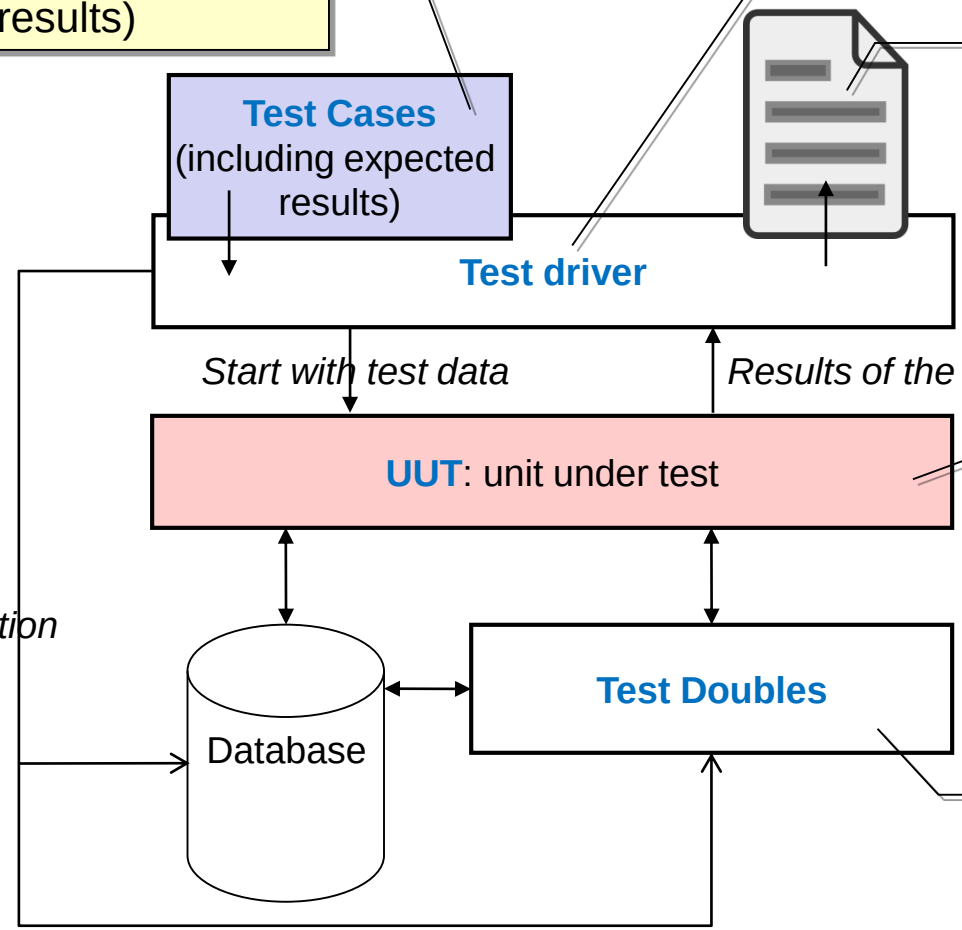

mockito

Test Workflow

The test department supplies the test cases (+ expected results)

Test driver calls the UUT with the test cases. (for example, JUnit)

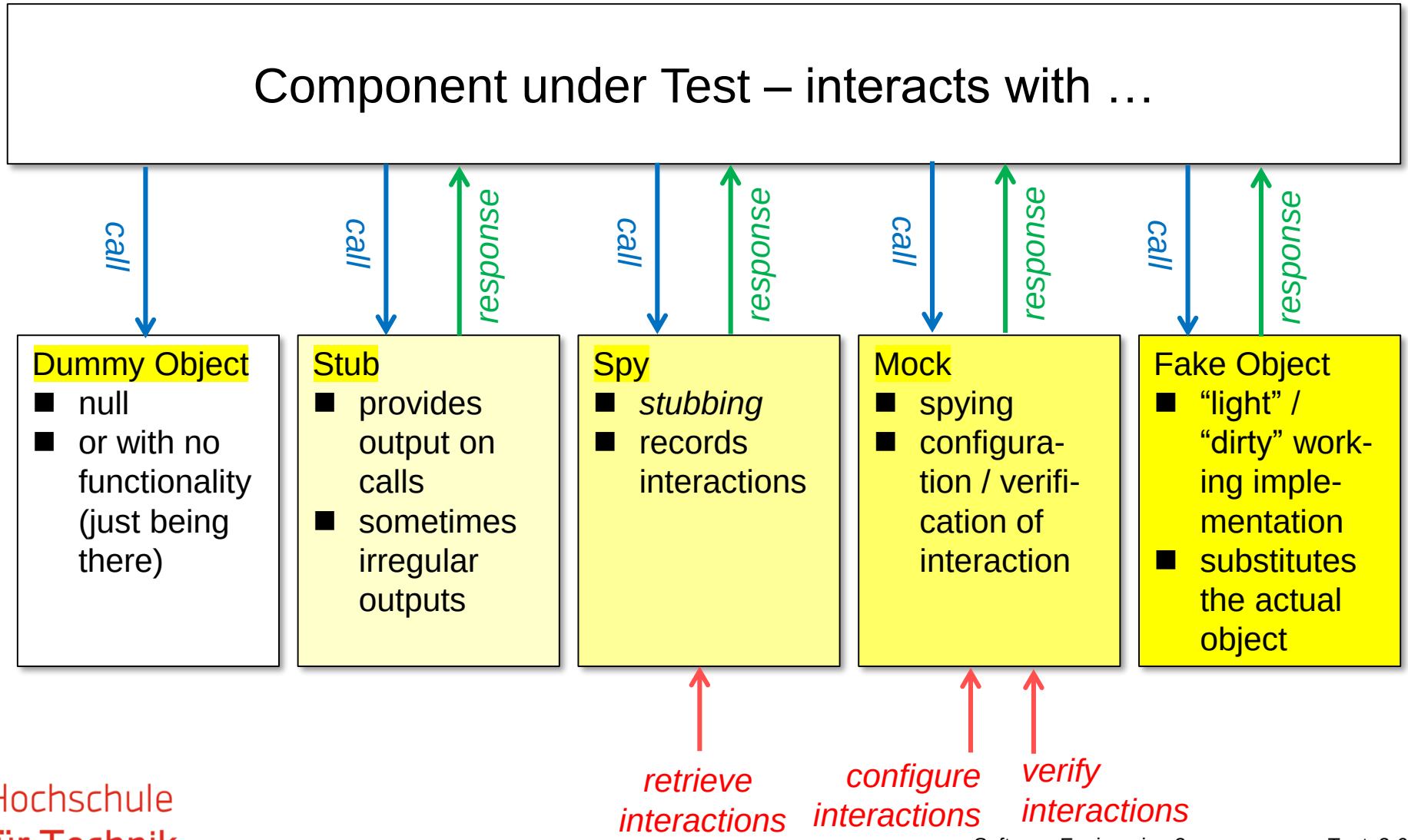
Test progress **report**: conformity / deviation of actual results against expected results.



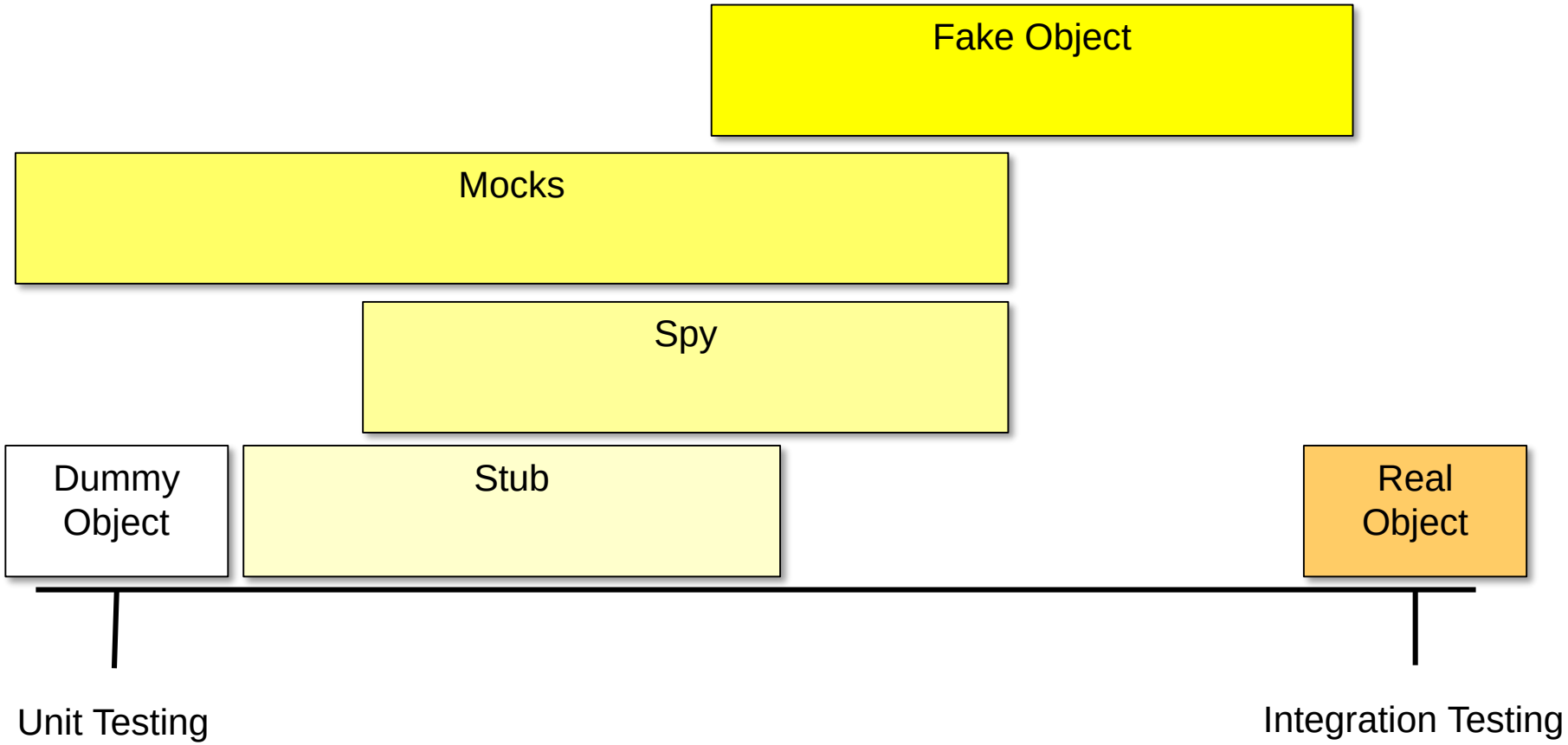
The developer provides the UUT

Developers/testers provide stubs that replace the real (too complex/not yet available) components

Test Doubles – Categories



Test-Doubles – Usage



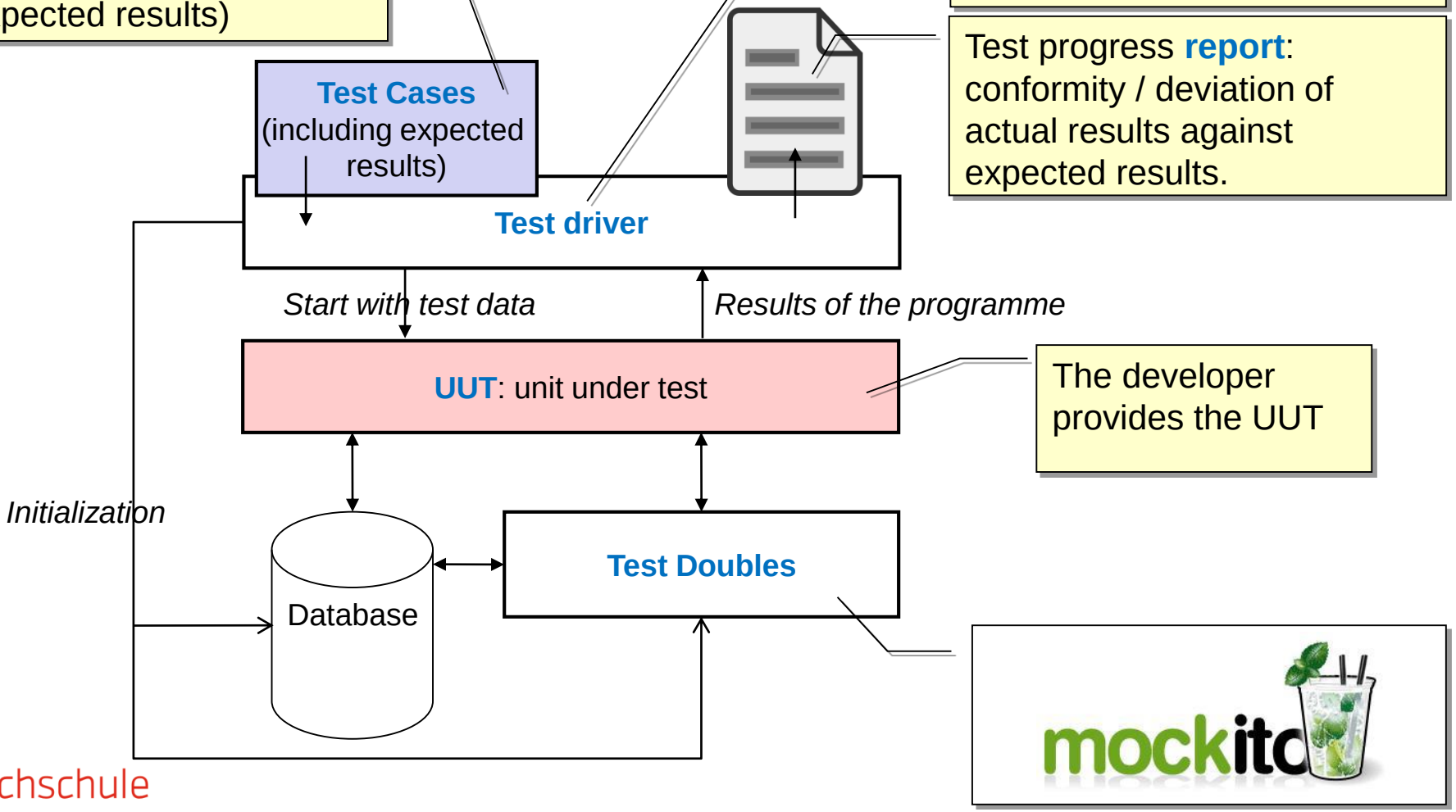
source: G. Bala, Master Thesis, HFT, 2012

Test Environment with Mockito

The test department supplies the test cases (+ expected results)

Test driver calls the UUT with the test cases. (for example, JUnit)

Test progress **report**: conformity / deviation of actual results against expected results.



The developer provides the UUT

Mockito – 1

Mockito allows

- mocking interfaces → on-the-fly creation of interface-objects
- simple creation of mock-objects
- injection of mock-objects in components under test

Mock-Objects

- can be stubbed → method-results can be defined
- can be verified → check that certain interactions have happened

Mockito – 2

- defined 2007 by Szczepan Faber
- Mockito is free and open source; fully usable within JUnit
- installation
  - maven → add to dependencies in `pom.xml`

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>3.9.0</version>
</dependency>
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>3.9.0</version>
    <scope>test</scope>
</dependency>
```

or a more recent version `3.x.xx`

Mockito – Mock Creation/Verification

```
import static org.mockito.Mockito.*;

import java.util.List;

public class MockitoSample1 {

	public static void main(String[] args) {
		//mock creation
		List<String> mockedList = mock(List.class);

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

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

This is the interface to be mocked.

Typically in the test setup; the mock has to be injected in the CuT

This will happen in the CuT

Verification will be done at the end of the test. Throws RTE on fail.

Mockito – Stubbing

```
public class MockitoSample2 {

	public static void main(String[] args) {
		//mock creation
		List<String> mockedList = mock(List.class);

		//stubbing
		when(mockedList.get(0)).thenReturn("zero");
		when(mockedList.get(1)).thenReturn("one", "eins", "uno");
		when(mockedList.get(2)).
		    thenReturn("two").thenThrow(new RuntimeException());

		//using mock object
		System.out.println(mockedList.get(0));
		System.out.println(mockedList.get(1));
		System.out.println(mockedList.get(1));
		System.out.println(mockedList.get(1));
		System.out.println(mockedList.get(1));
		System.out.println(mockedList.get(2));
		try { System.out.println(mockedList.get(2));
		}catch(Exception e){ System.out.println(e); }
		System.out.println(mockedList.get(3));

		//verification
		verify(mockedList, times(4)).get(1);
	}
}
```

Define results on given calls

Define results on consecutive calls

throw runtime-exception on second call

prints `zero`, `one`, `eins`, `uno`, `uno`, `two`

throws RuntimeEx

prints `null`

verifies `ok`

Mockito – Advanced Verification

```
public class Data {
    // …
}

public interface Service {
    public void doSomething(Data arg);
}

public class MockitoSamples3 {

    public static void main(String[] args) {
        //mock creation
        Service mock = mock(Service.class);

        //using the mock object
        Data c1 = new Data(), c2 = new Data();
        mock.doSomething(c1);
        mock.doSomething(c2);

        ArgumentMatcher<Data> any = arg -> arg instanceof Data;

        // verification
        verify(mock, times(2)).doSomething(argThat(any)); }
}
```

Some class to be used

The interface to be mocked

Execute the mock twice.

An argument matcher for `Data` – only checks if the called argument was an `Data`.

Verify the usage for any instance of `Data`.

Mockito – Usage within Junit – 1

```
public class Client {

        private Service service;

        public void callService (){
                service.doSomething(new
                service.doSomething(new Data ());
        }
}

public interface Service {
        public void doSomething(Data arg);
}

public class Data {
        // …
}
```

This is the Unit under Test

The object is not set here – probably will be through some injection in the real application (e.g. JEE EntityManager)

Calls to the interface method.

This is the interface used by the Unit under Test

This is the Data used by the interface

Mockito – Usage within Junit – 2

```
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.ArgumentMatcher;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

@ExtendWith(MockitoExtension.class)
public class ClientTest {

	private ArgumentMatcher<Data>
		any = arg -> arg instanceof Data;


	@Mock
	private Service mock;

	@InjectMocks
	private Client client;

	@Test
	public void test1() throws Exception{
		client.callService();
		verify(mock, times(2)).action(argThat(any));
	}
}
```

Do not forget – otherwise no injection will happen.

Same as
`Service mock = mock(Service.class)`

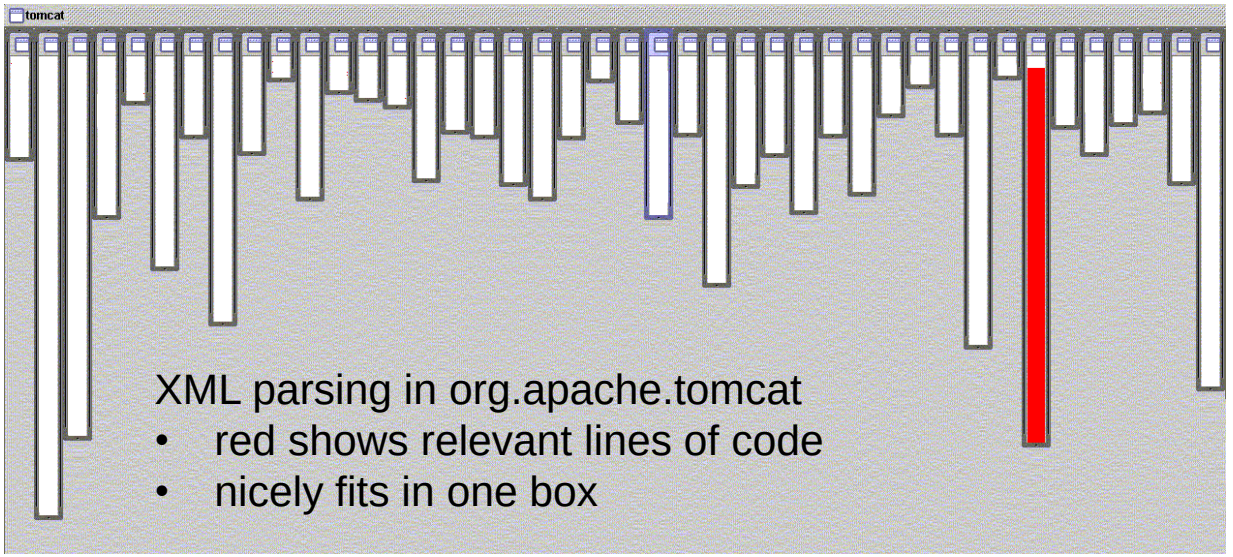
Unit under Test; creates an object and injects the created mocks, through constructor, setter, direct injection

The actual test case: checks, if expected interactions have happened – if not, the test will fail.


aspectj
crosscutting objects for better modularity

Aspect Oriented Programming (AOP) – 1

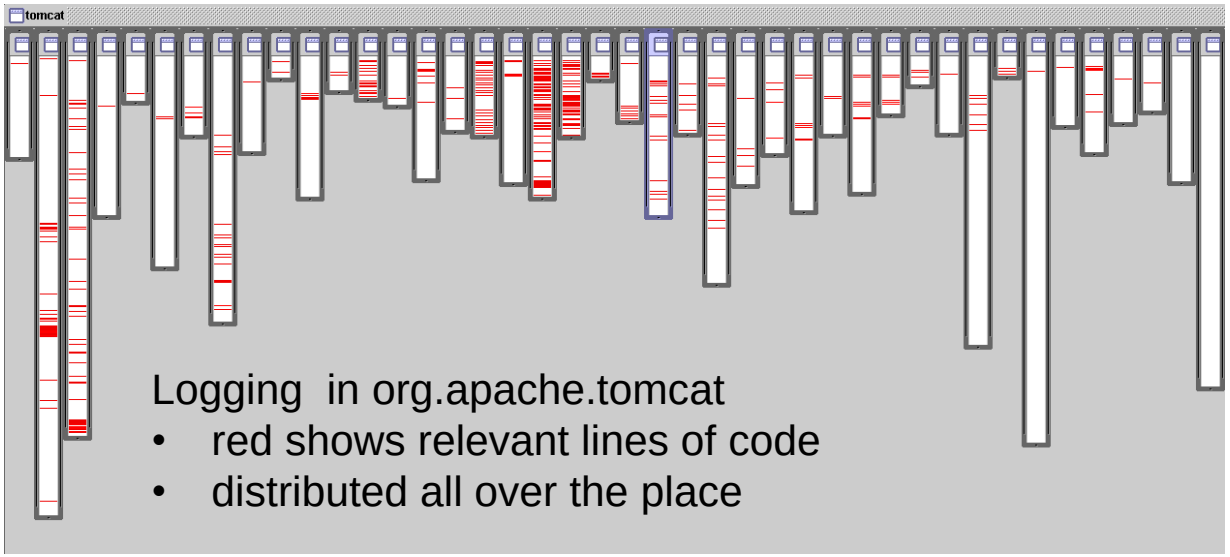
- Regular programming
  - typically concentrates on one use case and handles this within one module



source: K. Havelund, http://www.runtime-verification.org/course09/, 2009

Aspect Oriented Programming (AOP) – 2

- Sometimes a use case is “scattered” across several other modules
  - typical examples: logging, enforcing security policies, monitoring – and advanced testing



source: K. Havelund, http://www.runtime-verification.org/course09/, 2009

Aspect Oriented Programming (AOP) – 3

- Enter AOP
  - merge of distributed code in one special module (“aspect”)
  - definition of points of interest within other modules (“pointcuts”)
  - definition of actions to be executed, when such a point is reached (“advices”)

Aspect Oriented Programming (AOP) – 4

This is our “system”:

```
public class Hello {

   public static void sayHello() {
      System.out.print("Hello World");
    }

   public static void sayGoodbye() {
      System.out.print("Goodbye, cruel World");
   }
}

public class HelloToo {

   public static void saySomething() {
      System.out.print("Hello World, too");
   }

   public static void makeRemark() {
      System.out.println("Hi World");
   }
}

public class Main {

   public static void main(String[] args) {
      Hello.sayHello();
      HelloToo.saySomething();
      HelloToo.makeRemark();
      Hello. sayGoodbye();
   }
}
```

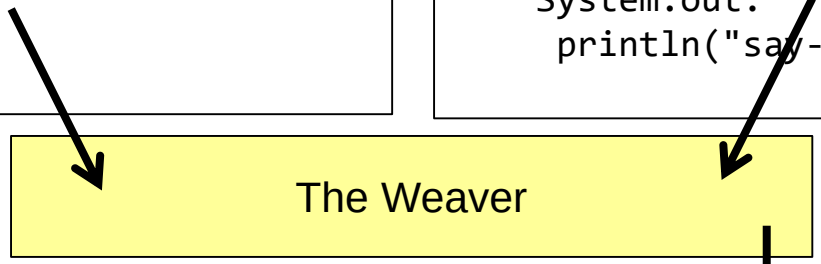
We would like to log all calls to “say…”-Methods (Pseudo-Language)

```
Monitor:

  when(method say*() is called)
  then
    System.out.
     println("say-Method executed");
```

Aspect Oriented Programming (AOP) – 5

```
public class Hello {
      …
}

public class HelloToo {
      …

public class Main {
      …
}
```

```
Monitor:

  when(method say*() is called)
  then
    System.out.
     println("say-Method executed");
```



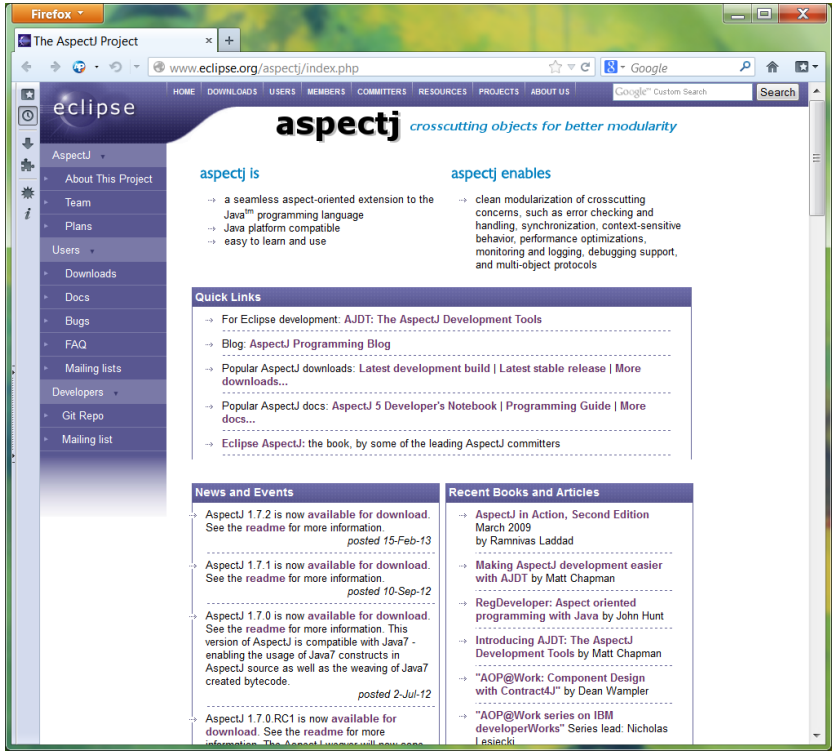

```
public class Hello {

      public static void sayHello() {
            System.out.print("say-Method executed");
            System.out.print("Hello World");
       }

      public static void sayGoodbye() {
            System.out.print("say-Method executed");
            System.out.print("Goodbye, cruel World");
       }
}

public class HelloToo {

      public static void saySomething() {
            System.out.print("say-Method executed");
            System.out.print("Hello World, too");
       }

      public static void makeRemark() {
            System.out.println("Hi World");
       }
}
```

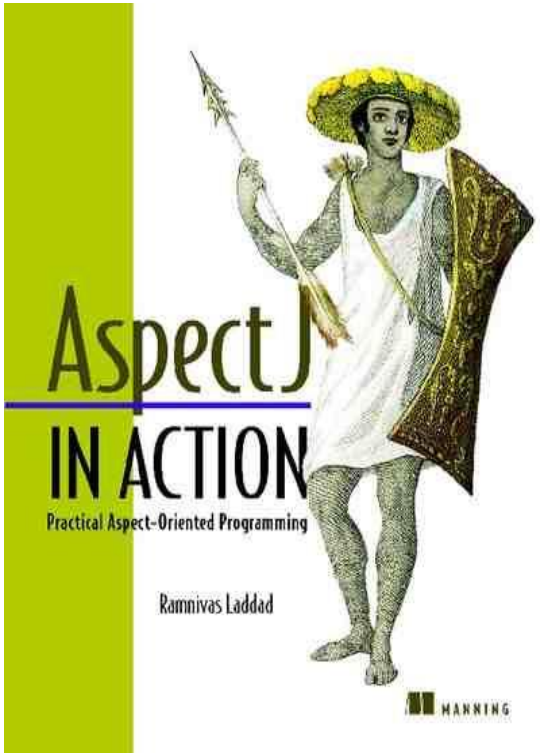
Aspect Oriented Programming with AspectJ – 1

- defined 1998 at Xerox PARC, extension of Java
- adds an additional compiler ("weaver"), which creates class files compatible with any JVM
- the AspectJ compiler is free and open source, and very mature
- current AspectJ-Eclipse 4.10 integration available at http://download.eclipse.org/tools/ajdt/410/dev/update

source: K. Havelund, http://www.runtime-verification.org/course09/, 2009

Aspect Oriented Programming with AspectJ – 2





http://www.eclipse.org/aspectj/index.php
http://www.eclipse.org/aspectj/doc/released/progguide/index.html

Laddad, R.: AspectJ in Action, 2nd ed.Manning 2009

Aspect Oriented Programming with AspectJ – 3

Define a pointcut

Define an aspect

This is the actual syntax.

Means: when executing any method, which starts with say

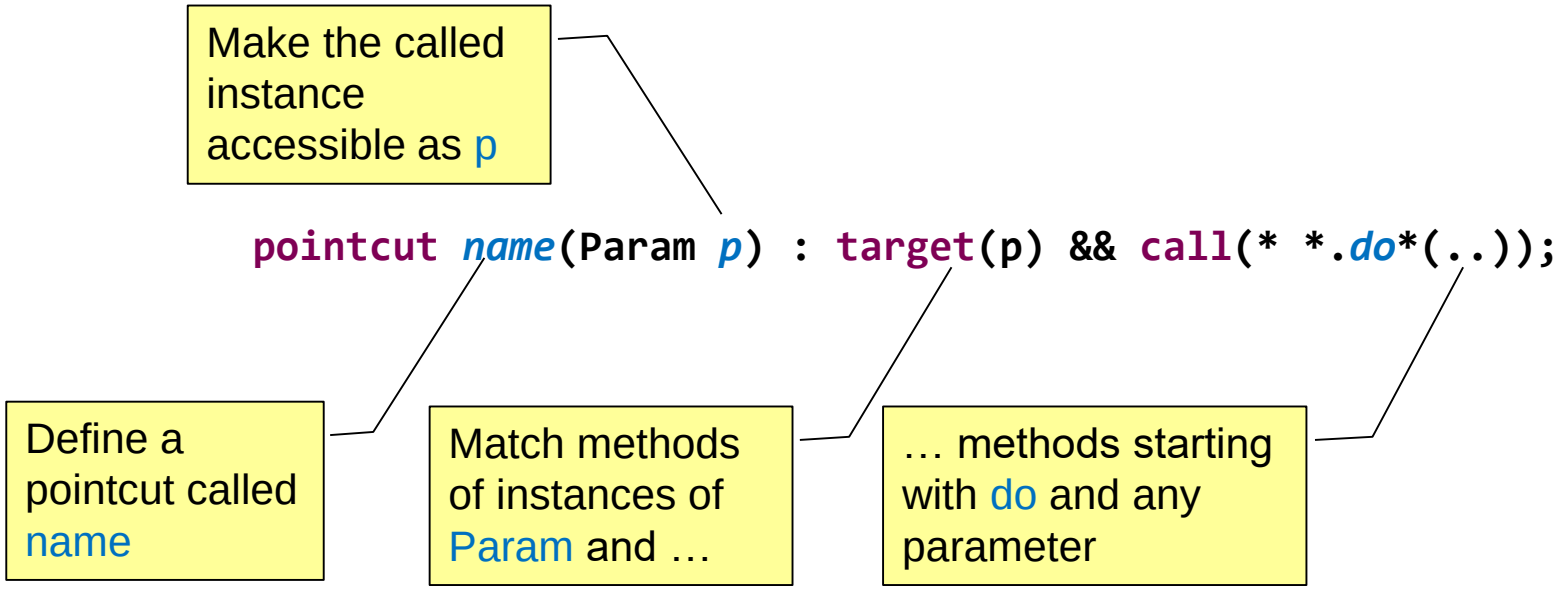
```
public aspect Monitor {

    pointcut interesting() : execution(* *.say*(..));

    before() : interesting() {
        System.out.print("execution started >>");
    }

    after() : interesting() {
        System.out.println("<< execution finished");
    }
}
```

Define an advice on the pointcut, which is triggered before the actual execution

Define an advice on the pointcut, which is triggered after the actual execution

Aspect Oriented Programming with AspectJ – 4

The difficult thing is the definition of the pointcuts



Aspect Oriented Programming with AspectJ – 5

For testing mainly relevant are the following pointcut definitions:

`within(AClass)`	→ only matched if within this class
`call(a pattern)`	→ call of a method (outside); i.e. you can do something before or instead the actual call happens
`execution(a pattern)`	→ execution of a method (inside); i.e. you can do something while the actual method is executed

Aspect Oriented Programming with AspectJ – 6

Typical patterns are:

`* *.do*(..)` → any result (first *), from any Class (second *), begins with **do** (do*), any parameters (..)

`void AClass.doThat(int)` → no result, from **AClass**, the method **doThat**, with one **int** parameter

`AClass.new(..)` → a constructor from **AClass**, any parameters

Aspect Oriented Programming with AspectJ – 7

possible advices are:

`before()` → executed before the pointcut is executed

`after()` → executed after the pointcut is executed (in any case)

`after() returning(Param p)` → executed after the pointcut is executed (when returning regularly)

`after() throwing(Exception e)` → executed after the pointcut is executed (when an exception has been thrown)

`around()` → executed instead of the pointcut-code (you can resume the original code with **`proceed()`**)

Aspect Oriented Programming with AspectJ – 8

An aspect

- may add additional (private) fields to an object,
- which (of course) can only be accessed by the aspect
- typically within an advice
- e.g.

```
aspect MyAspect{
  private int TargetClass.field = 0;
  before(TargetClass t) :
      call(* TargetClass.*(..)) && target(t) {
    t.field = …; // do something
  }
}
```

Aspect Oriented Programming with AspectJ – 9

The aspect class itself:

- is actually a singleton, with the instance created automatically
- the instance can be accessed in another class e.g.

```
private MyAspect aspect =
                    MyAspect.aspectOf();
```

- otherwise an aspect class can be implemented like any other Java-class (i.e. methods, fields, etc.)

AspectJ for Testing

Mainly replacing actual behavior by self-defined behavior

- forcing certain results (unobtainable otherwise)
- working around unreachable resources, etc.

Typical structure:

```
public aspect MyInjector {

   pointcut replace() :
      within(ClassUnderTest) && call(a certain method);


  ResultType around() : replace() {

   if(should replace)
     do something instead of a certain method and return result;
   else
     return proceed();
  }
}
```

Attention: No `.class`, just the (qualified) name. If the class is not found, the compiler gives only a warning – no error.

AspectJ for Testing – Example 1

```
…
public class StringReader {

   public static void main(String[] args)
                                    throws IOException {

      Reader reader = new InputStreamReader(System.in);
      BufferedReader buffer = new BufferedReader(reader);
      String line = buffer.readLine();
      System.out.println("You said: " + line);
   }
}
```

Ever wondered, how an `IOException` could happen during `readLine()`?

AspectJ for Testing – Example 2

```
public aspect InjectException {
  private boolean throwException = false;
  public void throwException(){
    this.throwException = true;
  }
  pointcut onReadLine() : within(StringReader) && call(* *.readLine());
  String around() throws IOException : onReadLine() {
    if(throwException){
      this.throwException = false;
      throw new IOException("thrown by AspectJ");
    } else
      return proceed();
  }
}
```

By default, do nothing.

Trigger exception to be thrown once.

Observe all calls to `readLine` within the class under test.

On a call throw the `IOException`.

If no exception should be thrown, proceed normally.

```
public class TestReadFromConsole {
  private InjectException injector = InjectException.aspectOf();
  @Test public void testMain() {
    injector.throwException();
    assertThrows(IOException.class, () -> StringReader.main(null));
  }
}
```

Trigger the exception.

Attention: lambdas and Aspects don‘t mix too well – sometimes you may have to force a recompile.

Selenium Browser Automation

Selenium – Browser Automation

- 2004 developed as an internal tool for ThoughtWorks
- 2004 becoming an open source project
- Main components
  - Selenium IDE: recording of browser interaction (by Mozilla-plugin) → records can be edited afterwards
  - Selenium WebDriver: controls a browser directly through API *[focus here]* or by replaying the records
  - Selenium Grid: Distributed execution of several tests.
- available at: www.seleniumhq.org/download/

Selenium – Installation

- maven → add to dependencies in `pom.xml`

```
<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>3.141.59</version>
</dependency>
```

- for each browser an additional platform-specific browser-driver is required (be aware of 32/64-bit versions)
  - Firefox (geckodriver): https://github.com/mozilla/geckodriver/releases
  - Chrome (chromedriver): https://sites.google.com/a/chromium.org/chromedriver/downloads
- Selenium is quite sensitive to different / new browser versions and may not always run perfectly!

Selenium – Example

```
public class BrowseSearchWithChrome {

        private static final String CHROME_DRIVER
      = "driver/chromedriver 90.0.4430.24/chromedriver.exe";

   public static void main(String[] args) {

      System.setProperty("webdriver.chrome.driver",
                                 getResource(CHROME_DRIVER));

      ChromeOptions options = new ChromeOptions();
      options.addArguments("--no-sandbox");
      options.setPageLoadStrategy(PageLoadStrategy.EAGER);
      WebDriver driver = new ChromeDriver(options);

      String query = "HFT";
      String expected = "https://www.hft-stuttgart.de/";
      boolean found = false;

      try {
         driver.get("https://www.startpage.com//");
         driver.findElement(By.name("query")).sendKeys(query);

         driver.findElement(By.className(
         "search-form-home__button")).searchIcon.click();

         WebElement first = driver.findElement(
               By.className("w-gl__result-title"));

         String result = first.getAttribute("href");
         found = result.equals(expected);

      } catch (Exception e) {
         e.printStackTrace();
      } finally {
         driver.quit();
      }

      System.out.println("Test " + (found ? "passed." : "failed."));
   }
}
```

Set up the driver (with some specific settings for the current version 90.0.4430.24)

Open the browser on this page. Access a local page with `file://` prefix

Type the query

Press the search button.

Get the first result

Get the link for this.

Close the page again.

Selenium – Usage in JUnit-Tests

- Set up the driver in a “BeforeAll”-Fixture
- Close the driver in a “AfterAll”-Fixture
- The actual test cases call a page and assert, that they contain the desired content.

Selenium – Within JUnit

```
public class ChromeTest {

   private static final String CHROME_DRIVER =
      "driver/chromedriver 90.0.4430.24/chromedriver.exe";

   private static WebDriver driver;

   @BeforeAll
   static void setUp() {

      System.setProperty("webdriver.chrome.driver",
            getResource(CHROME_DRIVER));

      ChromeOptions options = new ChromeOptions();
      options.addArguments("--no-sandbox");
      options.setPageLoadStrategy(PageLoadStrategy.EAGER);

      driver = new ChromeDriver(options);
   }

   @Test
   void testSearchHFT() {

      String query = "HFT";
      String expected = "https://www.hft-stuttgart.de/";

      driver.get("https://www.startpage.com//");
      driver.findElement(By.name("query")).sendKeys(query);
      driver.findElement(By.className(
            "search-form-home__button")).click();
      WebElement first = driver.findElement(
            By.className("w-gl__result-title"));

      String result = first.getAttribute("href");
      assertEquals(expected, result);
   }

   @AfterAll
   static void tearDown() {
         driver.quit();
   }
}
```