CHALMERS
UNIVERSITY OF TECHNOLOGY

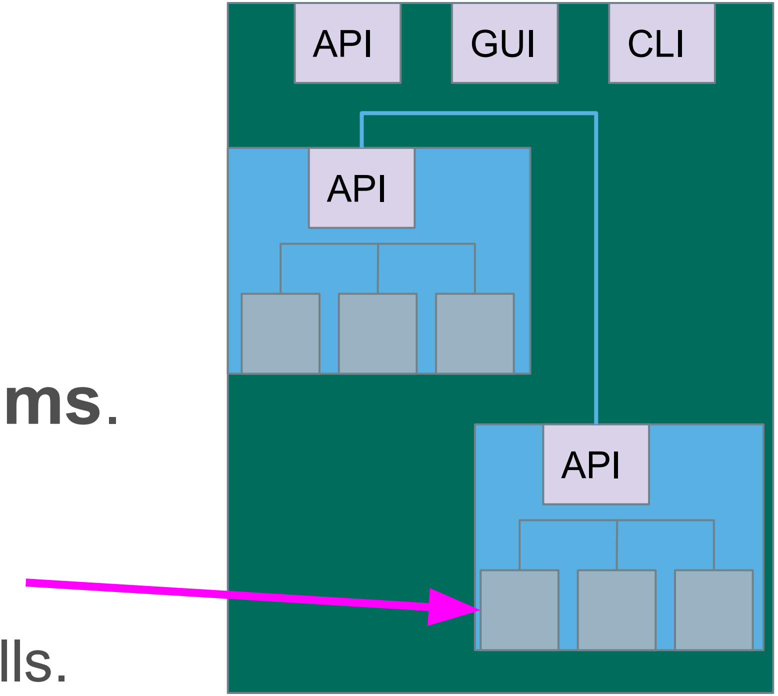
UNIVERSITY OF GOTHENBURG

Lecture 8: Unit Testing and Test Automation

Gregory Gay
DIT636/DAT560 - February 8, 2023

Testing Stages

- We interact with **systems** through **interfaces**.
  - APIs, GUIs, CLIs
- Systems built from **subsystems**.
  - With their own interfaces.
- Subsystems built from **units**.
  - Communication via method calls.



Today's Goals

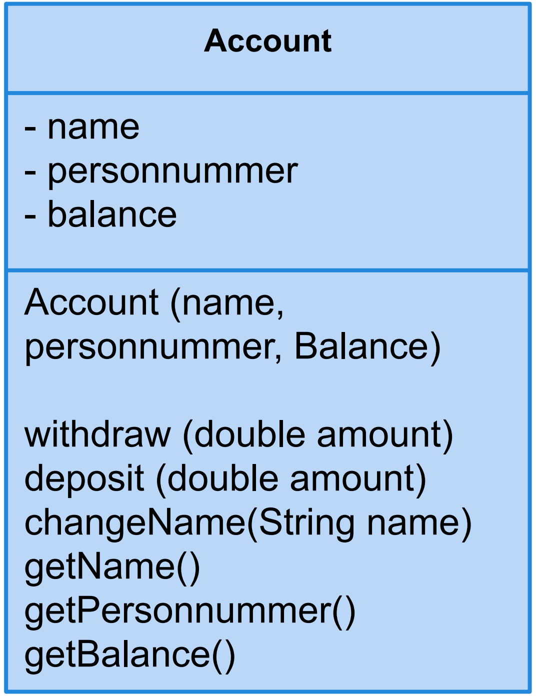
- Unit Testing
  - Testing of individual classes
- Writing and executing test cases
  - How to write unit tests in JUnit.
  - Executing tests as part of a build script.

Unit Testing

- Testing the smallest “unit” that can be tested.
  - Often, a class and its methods.
- Tested in **isolation** from all other units.
  - **Mock** the results from other classes.
- Test input = method calls.
- Test oracle = assertions on output/class variables.

Unit Testing

- For a unit, tests should:
  - Test all “jobs” associated with the unit.
    - Individual methods belonging to a class.
    - Sequences of methods that can interact.
  - Set and check class variables.
    - Examine how variables change after method calls.
    - Put the variables into all possible states (types of values).



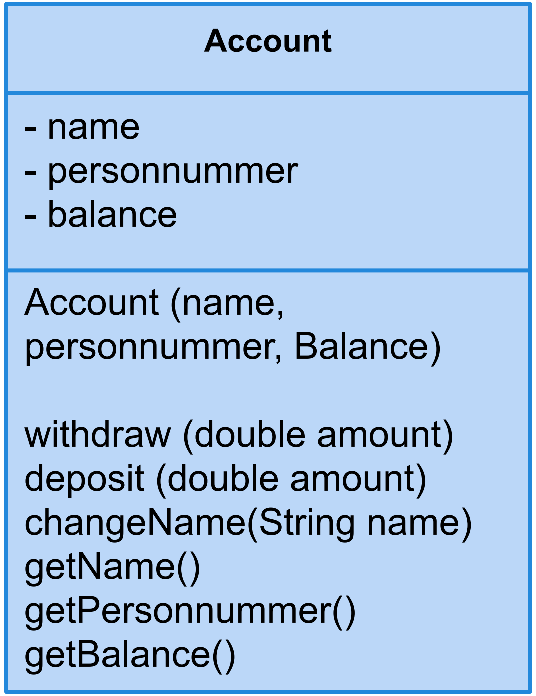
Account

- name
- personnummer
- balance

Account (name, personnummer, Balance)

withdraw (double amount)
deposit (double amount)
changeName(String name)
getName()
getPersonnummer()
getBalance()

Unit Testing - Account



Account

- name
- personnummer
- balance

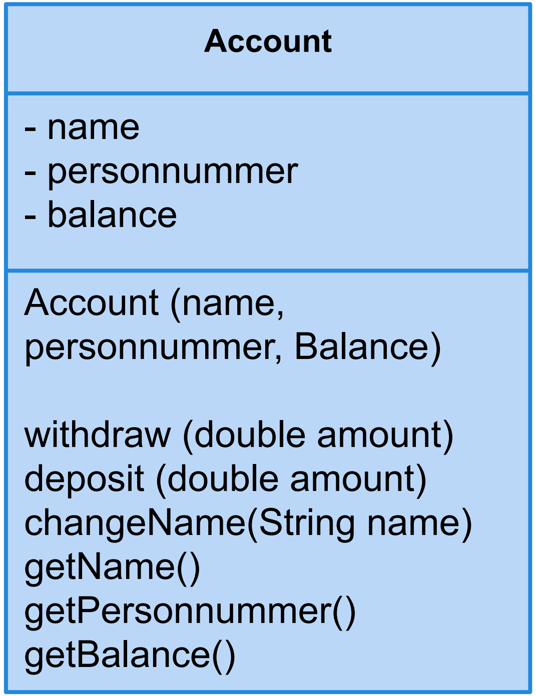
Account (name, personnummer, Balance)

withdraw (double amount)
deposit (double amount)
changeName(String name)
getName()
getPersonnummer()
getBalance()

Unit tests should cover:

- Set and check class variables.
  - Can any methods change name, personnummer, balance?
  - Does changing those create problems?
- Each “job” performed by the class.
  - Single methods or method sequences.
    - Vary the order methods are called.
  - Each outcome of each “job” (error handling, return conditions).

Unit Testing - Account


Account

- name
- personnummer
- balance

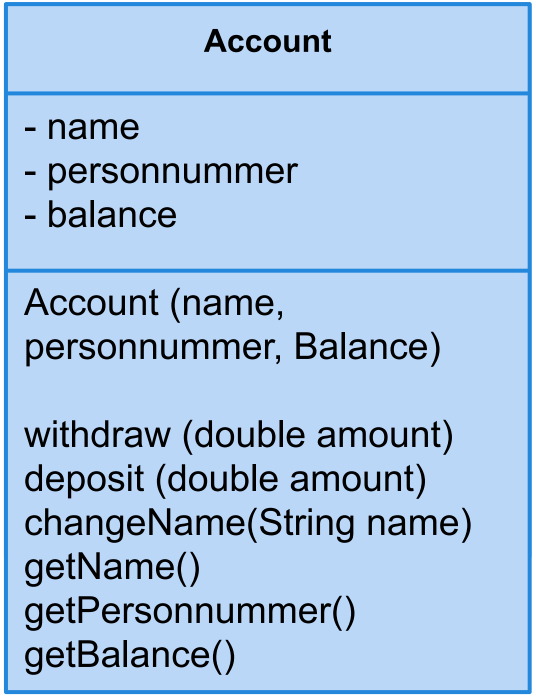
Account (name, personnummer, Balance)

withdraw (double amount)
deposit (double amount)
changeName(String name)
getName()
getPersonnummer()
getBalance()

Some tests we might want to write:

- Execute constructor, verify fields.
- Check the name, change the name, make sure changed name is in place.
- Check that personnummer is correct.
- Check the balance, withdraw money, verify that new balance is correct.
- Check the balance, deposit money, verify that new balance is correct.

Unit Testing - Account



Account

- name
- personnummer
- balance

Account (name, personnummer, Balance)

withdraw (double amount)
deposit (double amount)
changeName(String name)
getName()
getPersonnummer()
getBalance()

Some potential error cases:

- Withdraw more than is in balance.
- Withdraw a negative amount.
- Deposit a negative amount.
- Withdraw/Deposit a small amount (potential rounding error)
- Change name to a null reference.
- Can we set an “malformed” name?
  - (i.e., are there any rules on a valid name?)

Unit Testing and Test Automation

Executing Tests

- How do you run test cases on the program?
  - System level: *could* run code and check results by hand.
  - **Please don’t do this.**
    - Humans are slow, expensive, and error-prone.
    - **Exception - exploratory and acceptance testing.**
  - Test design requires effort and creativity.
  - Test execution should not.

Test Automation

- Development of software to separate repetitive tasks from creative aspects of testing.
- Control over *how* and *when* tests are executed.
  - Control environment and preconditions/setup.
  - Automatic comparison of predicted and actual output.
  - Automatic hands-free re-execution of tests.

Testing Requires Writing Code

- The component to be tested must be isolated and *driven* using method or interface calls.
- Untested dependencies must be *mocked* with reliable substitutions.
- The deployment environment must be simulated by a controllable *harness*.

Test Scaffolding

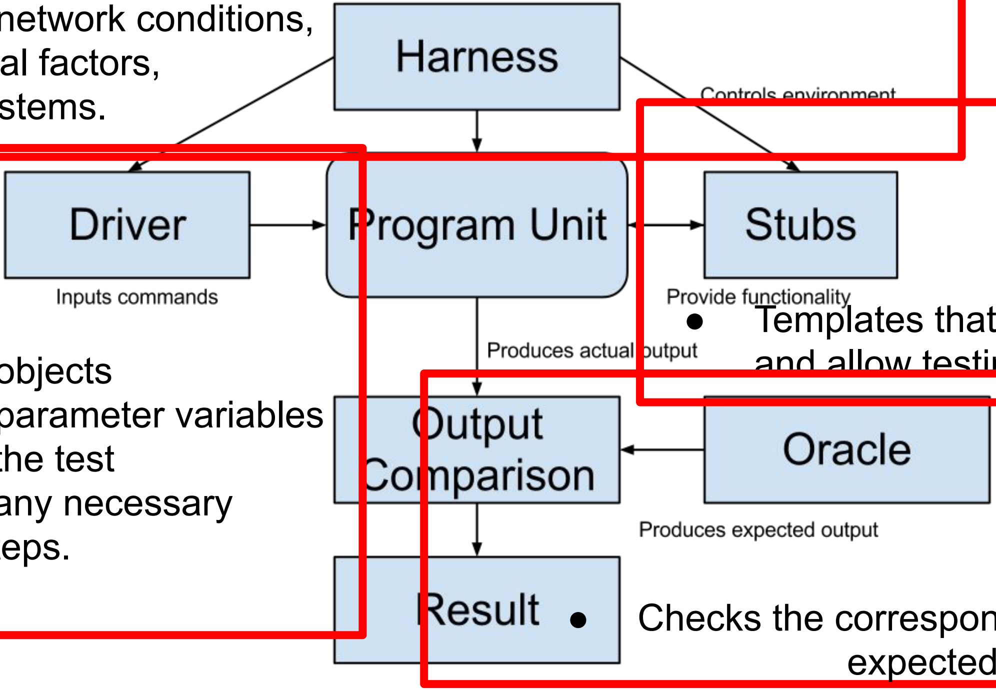
- Test scaffolding is a set of programs written to support test automation.
  - Not part of the product, often temporary
- Allows for:
  - Testing before all components complete.
  - Testing independent components.
  - Control over testing environment.

Test Scaffolding

- A **driver** substitutes for a main or calling program.
    - Test cases are drivers.
- A **harness** substitutes for part of the deployment environment.
- A **stub** (or **mock object**) substitutes for system functionality that has not been tested.
- Support for recording and managing test execution.

Test Scaffolding

- Simulates the execution environment.
- Can control network conditions, environmental factors, operating systems.



- Initializes objects
- Initializes parameter variables
- Performs the test
- Performs any necessary cleanup steps.

- Templates that provide functionality and allow testing in isolation.

- Checks the correspondence between the produced and expected output and renders a test verdict.

Writing an Executable Test Case

- Test Input
  - Any required input data.
- Expected Output (Test Oracle)
  - What *should* happen, i.e., values or exceptions.
- Initialization
  - Any steps that must be taken before test execution.
- Test Steps
  - Interactions (e.g., method calls), and output comparisons.
- Tear Down
  - Steps that must be taken after execution to prepare for the next test.

Writing a Unit Test

JUnit is a Java-based toolkit for writing executable tests.

- Choose a target from the code base.
- Write a “testing class” containing a series of unit tests centered around testing that target.

```
public class Calculator {
  public int evaluate (String
              expression) {
    int sum = 0;
    for (String summand:
              expression.split("\\+"))
      sum += Integer.valueOf(summand);
    return sum;
  }
}
```

JUnit Test Skeleton

@Test annotation defines a single test:

Type of scenario, and expectation on outcome.
I.e., `testEvaluate_GoodInput()` or `testEvaluate_NullInput()`

```
@Test
public void test<Feature or Method Name>_<Testing Context>() {
    //Define Inputs
    try{ //Try to get output.
    }catch(Exception error){
        fail("Why did it fail?");
    }
    //Compare expected and actual values through assertions or through
    //if-statements/fail commands
}
```

Writing JUnit Tests

Convention - name the test class after the class it is testing.

```
public class Calculator {
  public
    int sum = 0;
    for (String summand:
               expression.split(
      sum += Integer.valueOf(summand);
    return sum;
  }
}
```

Each test is denoted with keyword **@test**.

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class CalculatorTest {
  @Test
  void testEvaluate_Valid_ShouldPass(){
    Calculator calculator = new Calculator();
    int sum = calculator.evaluate("1+2+3");
    assertEquals(6, sum);
  }
}
```

Initialization

Test Steps

Input

Oracle

Test Fixtures - Shared Initialization

@BeforeEach annotation defines a common test initialization method:

```
@BeforeEach
public void setUp() throws Exception
{
   this.registration = new Registration();
   this.registration.setUser(“ggay”);
}
```

Test Fixtures - Teardown Method

@AfterEach annotation defines a common test tear down method:

```
@AfterEach
public void tearDown() throws Exception
{
    this.registration.logout();
    this.registration = null;
}
```

More Test Fixtures

- **@BeforeAll** defines initialization to take place before any tests are run.
- **@AfterAll** defines tear down after all tests are done.

```
@BeforeAll
  public static void setUpClass() {
     myManagedResource = new
          ManagedResource();
  }

  @AfterAll
  public static void tearDownClass()
throws IOException {
    myManagedResource.close();
    myManagedResource = null;
  }
```

Assertions

Assertions are a "language" of testing - constraints that you place on the output.

- `assertEquals, assertArrayEquals`
- `assertFalse, assertTrue`
- `assertNull, assertNotNull`
- `assertSame,assertNotSame`

assertEquals

```
@Test
public void testAssertEquals() {
   assertEquals("failure - strings are not
equal", "text", "text");
}

@Test
public void testAssertArrayEquals() {
    byte[] expected = "trial".getBytes();
    byte[] actual = "trial".getBytes();
    assertArrayEquals("failure - byte arrays
not same", expected, actual);
}
```

- Compares two items for equality.
- For user-defined classes, relies on `.equals` method.
  - Compare field-by-field
  - `assertEquals(studentA.getName(), studentB.getName())` rather than `assertEquals(studentA, studentB)`
- assertArrayEquals compares arrays of items.

assertFalse, assertTrue

```
@Test
public void testAssertFalse() {
   assertFalse("failure - should be false",
(getGrade(studentA, “DIT635”).equals(“A”));
}

@Test
public void testAssertTrue() {
     assertTrue("failure - should be true",
(getOwed(studentA) > 0));
}
```

- Take in a string and a boolean expression.
- Evaluates the expression and issues pass/fail based on outcome.
- Used to check conformance of solution to expected properties.

assertSame, assertNotSame

```
@Test
public void testAssertNotSame() {
   assertNotSame("should not be same Object",
studentA, new Object());
}

@Test
public void testAssertSame() {
    Student studentB = studentA;
   assertSame("should be same", studentA,
studentB);
}
```

- Checks whether two objects are clones.
- Are these variables aliases for the same object?
  - assertEquals uses .equals().
  - assertSame uses ==

assertNull, assertNotNull

```
@Test
public void testAssertNotNull() {
    assertNotNull("should not be null",
    new Object());
}

@Test
public void testAssertNull() {
    assertNull("should be null", null);
}
```

- Take in an object and checks whether it is null/not null.
- Can be used to help diagnose and void null pointer exceptions.

Grouping Assertions

```
@Test
void groupedAssertions() {
  Person person = Account.getHolder();
  assertAll("person",
    () -> assertEquals("John",
person.getFirstName()),
    () -> assertEquals("Doe",
person.getLastName()));
}
```

- Grouped assertions are executed.
  - Failures are reported together.
  - Preferred way to compare fields of two data structures.

assertThat

ha tems,
bu

either - pass if one of these properties is true.

```
@Test
public void testAssertThat{
  assertThat("albumen", both(containsString("a")).and(containsString("b")));
  assertThat(Arrays.asList("one", "two", "three"), hasItems("one", "three"));
  assertThat(Arrays.asList(new String[] { "fun", "ban", "net" }),
                everyItem(containsString("n")));
  assertThat("good", allOf(equalTo("good"), startsWith("good")));
  assertThat("good", not(allOf(equalTo("bad"), equalTo("good"))));
  assertThat("good", anyOf(equalTo("bad"), equalTo("good")));
  assertThat(7, not(CombinableMatcher.<Integer>
                either(equalTo(3)).or(equalTo(4))));
}
```

Testing Exceptions

```
@Test
void exceptionTesting() {
  Throwable exception =
    assertThrows(
      IndexOutOfBoundsException.class,
      () -> { new ArrayList<Object>().get(0);}
    );
    assertEquals("Index:0, Size:0",
      exception.getMessage());
}
```

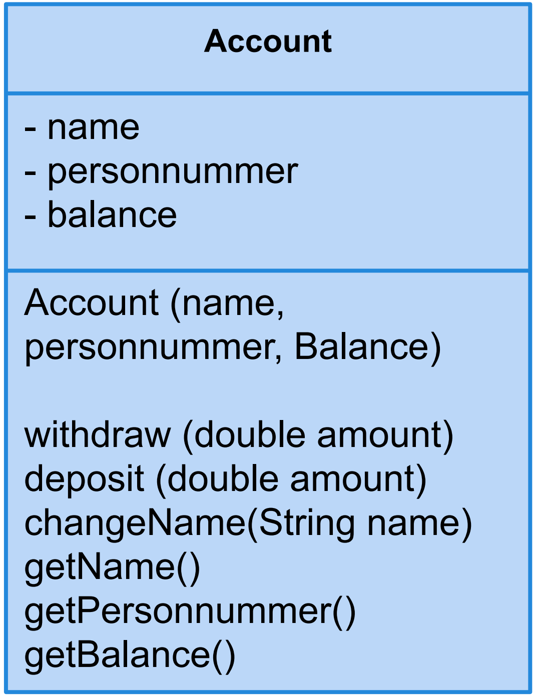
- When testing error handling, we expect exceptions to be thrown.
    - **assertThrows** checks whether the code block throws the expected exception.
    - **assertEquals** can be used to check the contents of the stack trace.

Testing Performance

```
@Test
void timeoutExceeded() {
  assertTimeout( ofMillis(10),
  () -> { Order.process(); });
}
@Test
void timeoutNotExceededWithMethod() {
  String greeting =
    assertTimeout(ofMinutes(2),
      AssertionsDemo::greeting);
  assertEquals("Hello, World!", greeting);
}
```

- **assertTimeout** can be used to impose a time limit on an action.
  - Time limit stated using ofMilis(..), ofSeconds(..), ofMinutes(..)
  - Result of action can be captured as well, allowing checking of result correctness.

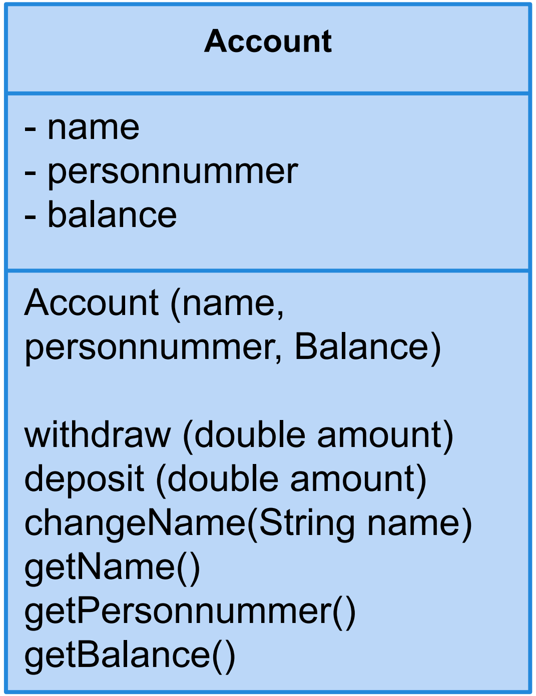
Unit Testing - Account



- Withdraw money, verify balance.

```
@Test
public void testWithdraw_normal() {
    // Setup
    Account account = new Account(“Test McTest”, “19850101-1001”, 48.5);
    // Test Steps
    double toWithdraw = 16.0; //Input
    account.withdraw(toWithdraw);
    double actual = account.getBalance();
    double expectedBalance = 32.5; // Oracle
    assertEquals(expected, actual); // Oracle
}
```

Unit Testing - Account



Account

- name
- personnummer
- balance

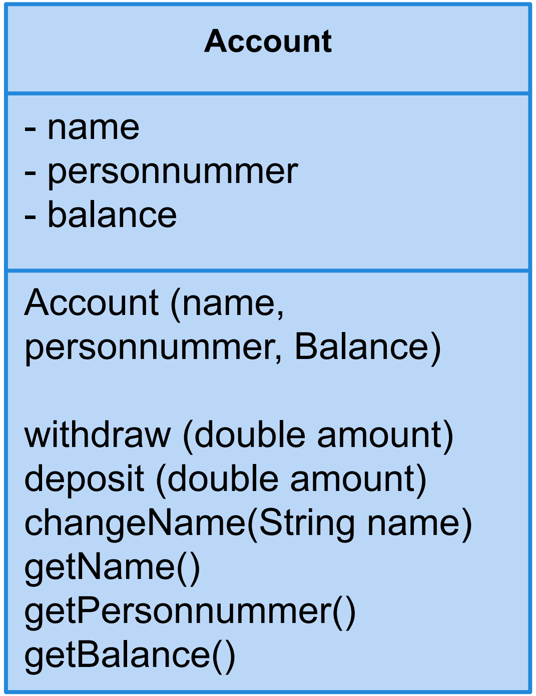
Account (name, personnummer, Balance)

withdraw (double amount)
deposit (double amount)
changeName(String name)
getName()
getPersonnummer()
getBalance()

- Withdraw more than is in balance.
    - (should throw an exception with appropriate error message)

```
@Test
public void testWithdraw_moreThanBalance() {
    // Setup
    Account account = new Account(“Test McTest”, “19850101-1001”, 48.5);
    // Test Steps
    double toWithdraw = 100.0; //Input
    Throwable exception = assertThrows(
        () -> { account.withdraw(toWithdraw); } );
    assertEquals(“Amount 100.00 is greater than balance 48.50”,
                 exception.getMessage()); // Oracle
}
```

Unit Testing - Account



Account

- name
- personnummer
- balance

Account (name, personnummer, Balance)

withdraw (double amount)
deposit (double amount)
changeName(String name)
getName()
getPersonnummer()
getBalance()

- Withdraw a negative amount.
  - (should throw an exception with appropriate error message)

```
@Test
public void testWithdraw_negative() {
    // Setup
    Account account = new Account("Test McTest", "19850101-1001", 48.5);
    // Test Steps
    double toWithdraw = -2.5; //Input
    Throwable exception = assertThrows(
        () -> { account.withdraw(toWithdraw); } );
    assertEquals("Cannot withdraw a negative amount: -2.50",
                 exception.getMessage()); // Oracle
}
```

Let’s take a break.

Best Practices

- Use assertions instead of print statements

```
@Test

public void testStringUtil_Bad() {

    String result = stringUtil.concat("Hello ", "World");
    System.out.println("Result is "+result);
}
```

🚫

```
@Test
public void testStringUtil_Good() {
    String result = stringUtil.concat("Hello ", "World");
    assertEquals("Hello World", result);
}
```

○

- The first will always pass (no assertions)

Best Practices

- If code is non-deterministic, tests should give deterministic results.

```
public long calculateTime(){
    long time = 0;
    long before = System.currentTimeMillis();
    veryComplexFunction();
    long after = System.currentTimeMillis();
    time = after - before;
    return time;
}
```

- Tests for this method should not specify exact time, but properties of a “good” execution.
    - The time should be positive, not negative or 0.
    - A range on the allowed times.

Best Practices

- Test negative scenarios and boundary cases, in addition to positive scenarios.
    - Can the system handle invalid data?
    - Method expects a string of length 8, with A-Z,a-z,0-9.
        - Try non-alphanumeric characters. Try a blank value. Try strings with length < 8, > 8
- Boundary cases test extreme values.
    - If method expects numeric value 1 to 100, try 1 and 100.
        - Also, 0, negative, 100+ (negative scenarios).

Best Practices

- Test only one unit at a time.
  - Each scenario in a separate test case.
  - Helps in isolating and fixing faults.
- Don’t use unnecessary assertions.
  - Specify how code should work, not a list of observations.
  - Generally, each unit test performs one assertion
    - Or all assertions are related.

Best Practices

- Make each test independent of all others.
  - Use @BeforeEach and @AfterEach to set up state and clear state before the next test case.
- Create unit tests to target exceptions.
  - If an exception should be thrown based on certain input, make sure the exception is thrown.

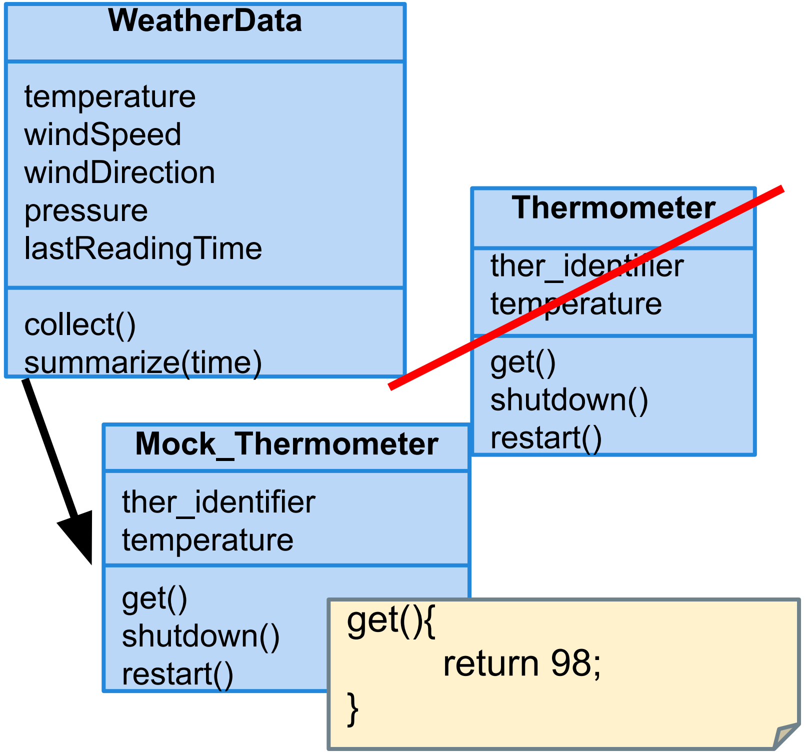
Scaffolding

- Mock objects and drivers are written as replacements for other parts of the system.
  - May be required if pieces of the system do not exist.
- Scaffolding allows control over test execution and greater observability to judge test results.
  - Simulate dependencies and test components in isolation.
  - Ability to set up specialized testing scenarios.
  - Ability to replace part of the program with a version more suited to testing.

Unit Testing - Object Mocking

Unit may depend on unfinished (or untested) components. Can **mock** those components.

- Same interface as real component, but hand-created simulation.
- Can be used to simulate abnormal operation or rare events.
    - Ex. Place exact data in database needed to hit special outcome.



Mocking Example

mockito

- Declare a mock object:

```
LinkedList mList = mock(LinkedList.class);
```

- Specify method behavior:

```
when(mList.get(0)).thenReturn(“first”);
```

  - Returns “first”: `mList.get(0);`
  - Returns null: `mList.get(99);`
    - Because behavior for “99” is not specified.

```
when(mList.get(anyInt()).thenReturn(“element”);
```

  - `mList.get(0)`, `mList.get(99)` both return “element”, as all input are specified.

Mocking Within a Test

```
@test
public void temperatureTest(){
    Thermometer mockTherm = mock(Thermometer.class);
    when(mockTherm.get()).thenReturn(98);
    WeatherData wData = new WeatherData();
    wData.collect(mockTherm);
    assertEquals(98,wData.temperature);
}
```

Build Systems

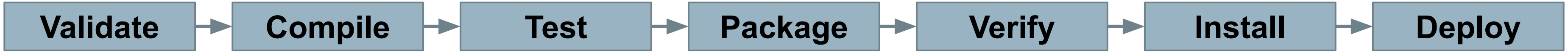
Build Systems

- Building, running tests, packaging and distributing are very common, effort-intensive tasks.
  - Building and deploying should be as easy as possible.
- **Build systems** ease process by automating as much as possible.
  - Repetitive tasks can be automated and run at-will.

Build Systems

- Allow control over code compilation, test execution, executable packaging, and deployment.
- Script defines actions that can be automatically invoked at any time.
- Many frameworks for build scripting.
  - Most popular for Java include Ant, Maven, Gradle.
  - Gradle is very common for Android projects.

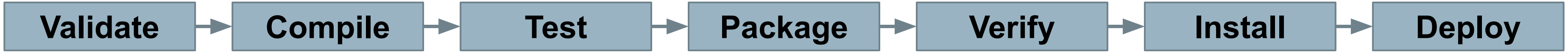
Build Lifecycle



Validate → Compile → Test → Package → Verify → Install → Deploy

- **Validate** the project is correct and all necessary information is available
- **Compile** the source code of the project.
- **Test** the source code using a suitable unit testing framework.
    - Run **unit tests** against classes and **subsystem integration tests** against groups of classes.
- Take the compiled code and **package** it in its distributable format, such as a JAR.

Build Lifecycle



- **Verify** - run system tests for quality/correctness.
  - System tests require a packaged executable.
  - This is also when tests of non-functional criteria like performance are executed.
- **Install** the package for use as a dependency in other projects locally.
- **Deploy** the package to the installation environment.

Apache Ant

- Build system for Java.
- Build scripts define **targets** that can be executed on command.
  - Correspond to lifecycle phases or other automated tasks.
  - Targets can trigger other targets.
  - Build scripts written in XML.
    - Platform neutral, But can invoke platform-specific commands.
    - Human and machine readable.
    - Created automatically by many IDEs (Eclipse).

A Basic Build Script

```
<?xml version = "1.0"?>
<project name = "Hello World Project" default = "info">
   <target name = "info">
      <echo>Hello World - Welcome to Apache Ant!</echo>
   </target>
</project>
```

- File typically named **build.xml**, and placed in the base directory of the project.
- Build script requires **project** element and at least one **target**.
  - Project defines a **name** and a default **target**.
  - This target prints project information.
    - **Echo** prints information to the terminal.

Targets

```
<target name = "deploy" depends = "package"> .... </target>
<target name = "package" depends = "clean,compile"> .... </target>
<target name = "clean" > .... </target>
<target name = "compile" > .... </target>
```

- A target is a collection of tasks you want to run in a single unit.
  - Targets can depend on other targets.
  - **deploy** command will call **package** target, which will call **clean** and **compile** first.
  - Dependencies denoted using the **depends** attribute.

Targets

```
<target name = "deploy" depends = "package"> .... </target>
<target name = "package" depends = "clean,compile"> .... </target>
<target name = "clean" > .... </target>
<target name = "compile" > .... </target>
```

- Target attributes:
  - **name** defines the name of the target (required)
  - **depends** lists dependencies of the target.
  - **description** is used to describe the target.
  - **if** and **unless** allow execution of the target to depend on a conditional attribute.
    - Execute target **if** attribute is true, or execute **unless** true.

Executing targets

```
<?xml version = "1.0"?>
<project name = "Hello World Project" default = "info">
   <target name = "info">
      <echo>Hello World - Welcome to Apache Ant!</echo>
   </target>
</project>
```

```
Buildfile: build.xml
info: [echo] Hello World - Welcome to Apache
Ant!
BUILD SUCCESSFUL
Total time: 0 seconds
```

- In the command line, invoke:
  - **ant <target name>**
- If no target is supplied, the default will be executed.
  - In this case, **ant** and **ant info** give same result because info is default target.

Properties

- XML does not natively allow variable declaration.
  - Instead, create **property** elements, which can be referred to by name.

```
<?xml version = "1.0"?>
<project name = "Hello World Project" default = "info">
   <property name = "sitename" value = "http://cse.sc.edu"/>
   <target name = "info">
      <echo>Apache Ant version is ${ant.version} - You are at ${sitename} </echo>
   </target>
</project>
```

Properties

```
<?xml version = "1.0"?>
<project name = "Hello World Project" default = "info">
   <property name = "sitename" value = "http://cse.sc.edu"/>
   <target name = "info">
      <echo>Apache Ant version is ${ant.version} - You are at ${sitename} </echo>
   </target>
</project>
```

- Properties have a name and a value.
  - Property value is referred to as **${property name}**.
  - Ant pre-defines **ant.version**, **ant.file** (location of the build file), **ant.project.name**, **ant.project.default-target**, and other properties.

Property Files

- Separate file can define static properties.
  - Allows reuse of build file in different environments (development, testing, production).
  - Allows easy lookup of property values.
- Called **build.properties** and stored in the same directory as build script.
  - Lists one property per line: `<name> = <value>`
  - Comments can be added using `# <comment>`

Property Files

- build.xml

```
<?xml version = "1.0"?>
<project name = "Hello World Project" default = "info">
   <property file = "build.properties"/>
   <target name = "info">
      <echo>You are at ${sitename}, version ${buildversion}.</echo>
   </target>
</project>
```

- build.properties

```
# The Site Name
sitename = http://cse.sc.edu
buildversion = 3.3.2
```

Conditions

```
<target name = "myTarget" depends =
"myTarget.check" if =
"myTarget.run"> .... </target>
<target name = "myTarget.check">
    <condition property =
"myTarget.run">
        <and>
            <available file =
"foo.txt"/>
            <available file =
"bar.txt"/>
        </and>
    </condition>
</target>
```

- Properties whose value determined by **and** and **or** expressions.
  - **And** requires that each property is true.
    - Both foo.txt and bar.txt must exist.
      - (**available** is an Ant command that checks for file existence)
  - **Or** requires that 1+ properties true.
  - Calling **myTarget.check** creates property (**myTarget.run**) that is true if both files are present.
  - When **myTarget** is called, it will run only if myTarget.run is true.

Ant Utilities

- **Fileset** generates list of files matching criteria for inclusion or exclusion.
  - ** means that the file can be in any subdirectory.
  - * allows partial file name matches.

```
<fileset dir = "${src}" casesensitive = "yes">
   <include name = "**/*.java"/>
   <exclude name = "**/*Stub*"/>
</fileset>
```

Ant Utilities

- **Path** is used to represent a classpath.
  - **pathelement** is used to add items or other paths to the path.

```
<path id = "build.classpath.jar">
   <pathelement path = "${env.J2EE_HOME}/j2ee.jar"/>
   <fileset dir = "lib"> <include name = "**/*.jar"/> </fileset>
</path>
```

Building a Project

```
<project name = "Hello-World" basedir = "." default = "build">
   <property name = "src.dir" value = "src"/>
   <property name = "build.dir" value = "target"/>
   <path id = "master-classpath">
      <fileset dir = "${src.dir}/lib"> <include name = "*.jar"/> </fileset>
      <pathelement path = "${build.dir}"/>
   </path>
</project>
```

- Properties **src.dir** and **build.dir** define where the source files are stored and where the built classes are deployed.
- Path **master-classpath** includes all JAR files in the lib folder and all files in the build.dir folder.

Building a Project

```
<project name = "Hello-World" basedir = "." default = "build">

   <target name = "clean" description = "Clean output directories">
      <delete>
         <fileset dir = "${build.dir}">
            <include name = "**/*.class"/>
         </fileset>
      </delete>
   </target>
</project>
```

- The clean target is used to prepare for the build process by cleaning up any remnants of previous builds.
  - In this case, it deletes all compiled files (.class)
  - May also remove JAR files or other temporary artifacts that will be regenerated by the build.

Building a Project

```
<project name = "Hello-World" basedir = "." default = "build">

    <target name = "build" description = "Compile source tree java files">
      <mkdir dir = "${build.dir}"/>
      <javac destdir = "${build.dir}" source = "1.8" target = "1.8">
         <src path = "${src.dir}"/>
         <classpath refid = "master-classpath"/>
      </javac>
   </target>

</project>
```

- The build target will create the build directory, compile the source code (using javac), and place the class files in the build directory.
  - Can specify which java version to target (1.8).
  - Must reference the classpath to use during compilation.

Creating a JAR File

- The **jar** command creates executable from compiled classes.

```
<target name = "package">
    <jar destfile = "lib/util.jar" basedir = "${build.dir}/classes"
        includes = "app/util/**" excludes = "**/Test.class">
    <manifest><attribute name = "Main-Class" value = "com.util.Util"/></manifest>
</jar>
</target>
```

- **destfile** is the location to place the JAR file.
- **basedir** is the base directory of included files.
- **includes** defines the files to include in the JAR.
- **excludes** prevents certain files from being added.
- The **manifest** declares metadata about the JAR.
    - Attribute Main-Class makes the JAR executable.

Running Unit Tests

- JUnit tests run using the **junit** command.

```
<target name = "test">
   <junit haltonfailure = "true" haltonerror = "false"
          printsummary = "true" timeout = "5000">
      <test name = "com.utils.UtilsTest"/>
   </junit>
</target>
```

  - **test** entries list the test classes to execute.
  - **haltonfailure** will stop test execution if any tests fail, **haltonerror** if errors occur.
  - **printsummary** displays test statistics (number of tests run, number of failures/errors, time elapsed).
  - **timeout** will stop a test and issue an error if the specified time limit is exceeded.

We Have Learned

- Test automation can lower cost and improve the quality of testing.
- Automation involves creating drivers, harnesses, stubs, and oracles.
- Test cases are often written in unit testing frameworks as executable code.
    - Assertions allow examination of output for failures.

We Have Learned

- Testing is not all that can be automated.
  - Project compilation, installation, deployment, etc.
- **Project build automation:**
  - Automating the entire compilation, testing, and deployment process.
  - Ant is an XML-based tool for automating build process.

Next Time

- Exercise Session: Unit Testing Practice
- Next Tuesday: Structural Testing
  - Pezze and Young, Ch. 5.3 and 12
- Assignment 1 due Sunday.
- Assignment 2 out.
  - (Based on Lectures 7-10, but you can start)

UNIVERSITY OF GOTHENBURG

CHALMERS

UNIVERSITY OF TECHNOLOGY