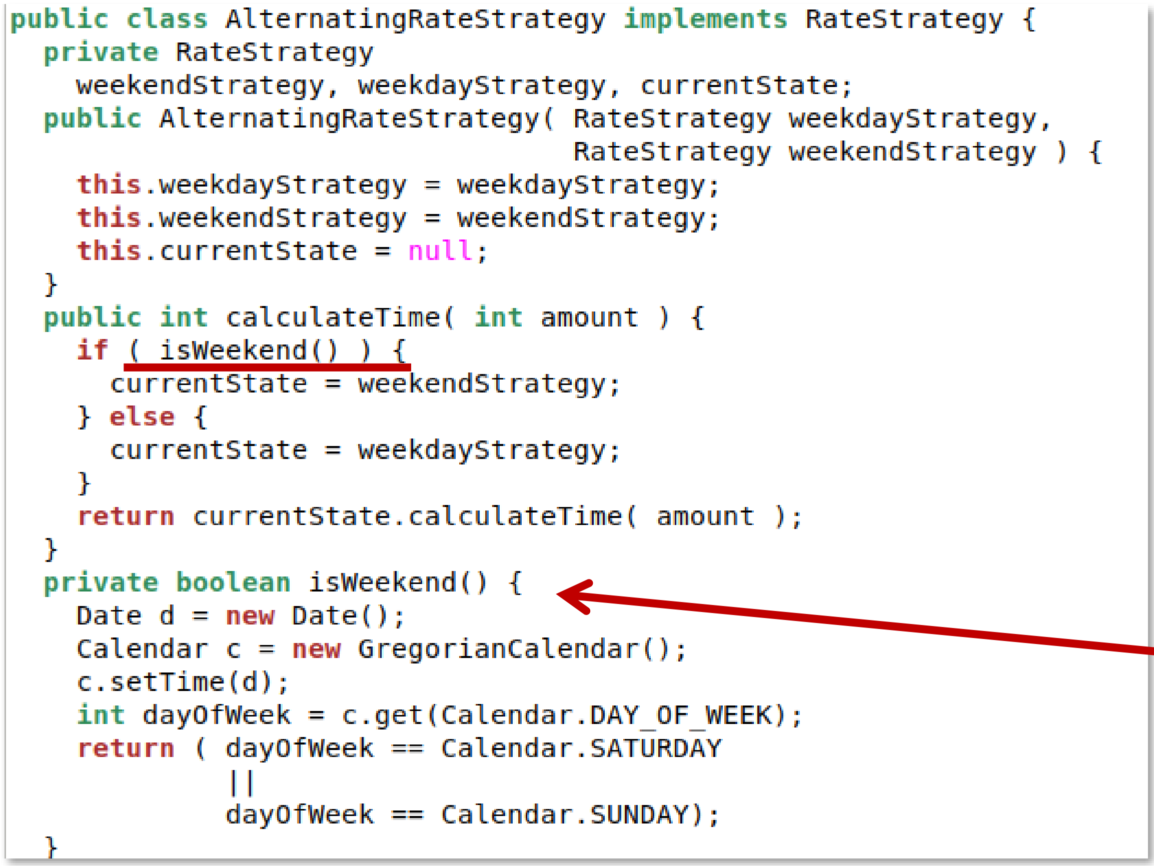
AARHUS UNIVERSITET

Software Engineering and Architecture

Test Doubles

… getting the world under test control

GammaTown's RateStrategy



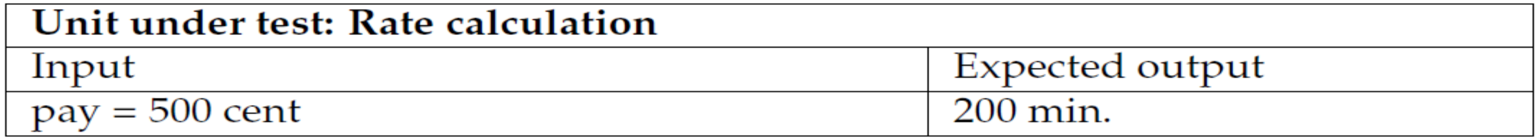
```java
public class AlternatingRateStrategy implements RateStrategy {
  private RateStrategy
    weekendStrategy, weekdayStrategy, currentState;
  public AlternatingRateStrategy( RateStrategy weekdayStrategy,
                                  RateStrategy weekendStrategy ) {
    this.weekdayStrategy = weekdayStrategy;
    this.weekendStrategy = weekendStrategy;
    this.currentState = null;
  }
  public int calculateTime( int amount ) {
    if ( isWeekend() ) {
      currentState = weekendStrategy;
    } else {
      currentState = weekdayStrategy;
    }
    return currentState.calculateTime( amount );
  }
  private boolean isWeekend() {
    Date d = new Date();
    Calendar c = new GregorianCalendar();
    c.setTime(d);
    int dayOfWeek = c.get(Calendar.DAY_OF_WEEK);
    return ( dayOfWeek == Calendar.SATURDAY
             ||
             dayOfWeek == Calendar.SUNDAY);
  }
```

But how to test? How do I TDD it?

Read system clock to determine if it is weekend

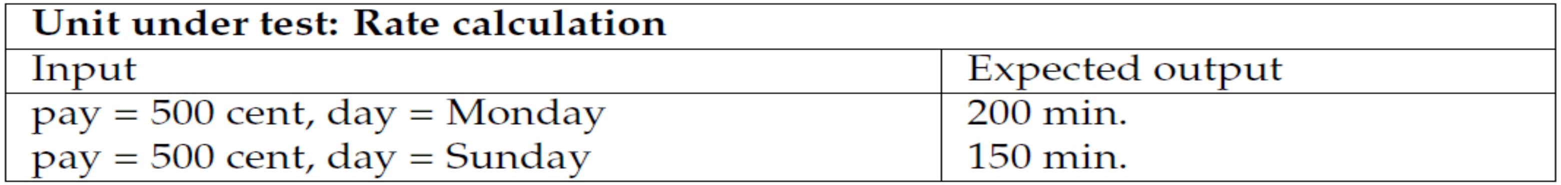
Tricky Requirement

- The test case for AlphaTown:



Unit under test: Rate calculation	
Input	Expected output
pay = 500 cent	200 min.

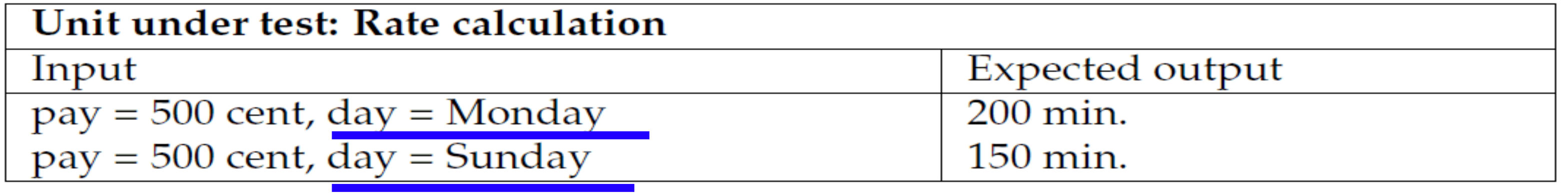
- … problematic for GammaTown…



Unit under test: Rate calculation	
Input	Expected output
pay = 500 cent, day = Monday	200 min.
pay = 500 cent, day = Sunday	150 min.

Analysis

- Gammatown, however, has one more parameter in the rate policy test case



Unit under test: Rate calculation	
Input	Expected output
pay = 500 cent, day = Monday	200 min.
pay = 500 cent, day = Sunday	150 min.

- The problem is

This parameter is not accessible from the testing code!

Code view

Fragment: chapter/state/compositional/iteration-2/src/test/java/paystation/manual/TestGammaWeekdayRate.java

```
    System.out.println( "Manual test of GammaTown Rate for Weekdays");
    RateStrategy rs =
      new AlternatingRateStrategy(new LinearRateStrategy(),
                                  new ProgressiveRateStrategy());
    // Should show 200 minutes for 500 cents
    assertThat(rs.calculateTime(500), is(500/5 * 2));
```

???

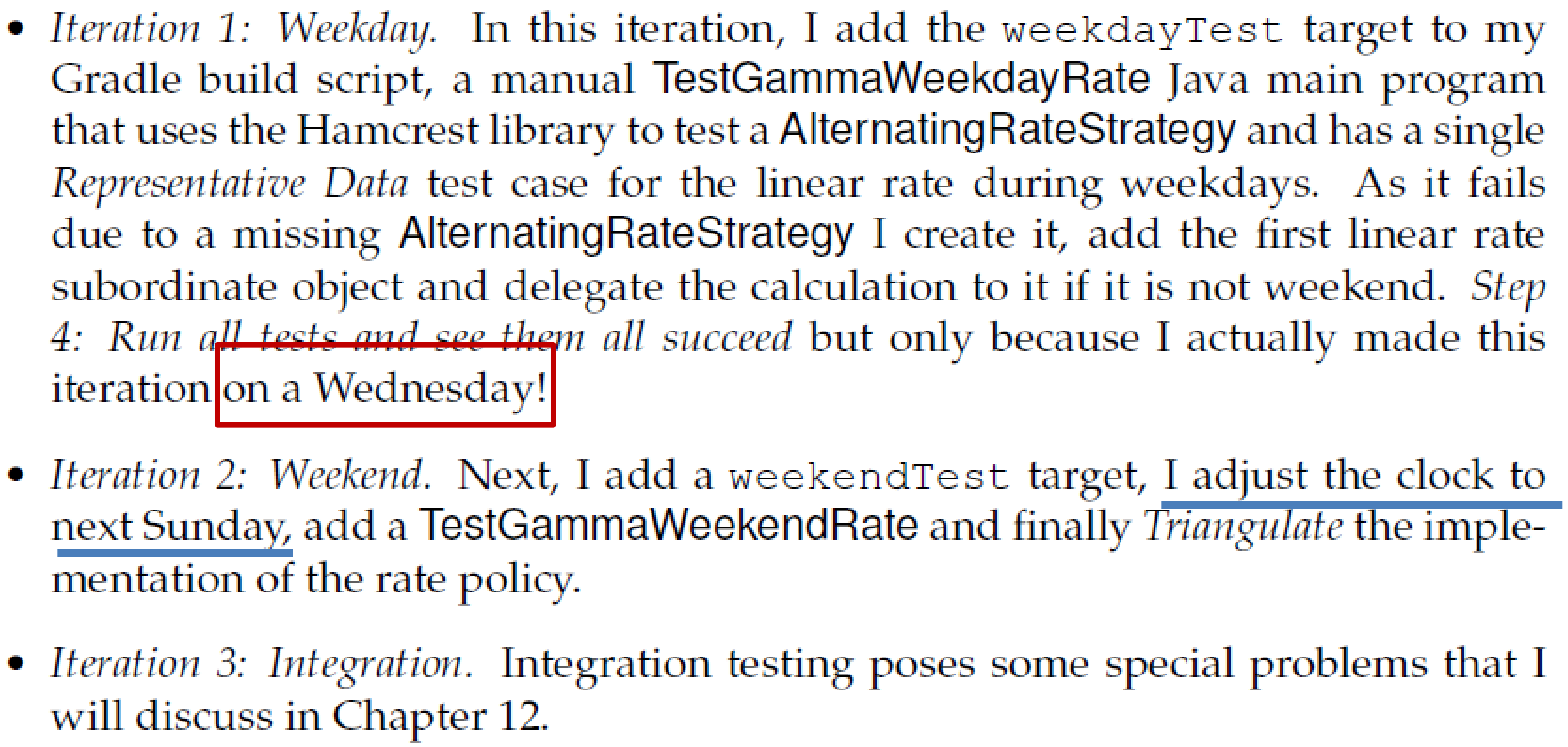
Direct input parameter: payment

Indirect input parameter: day of week

AARHUS UNIVERSITET

TDD of State Pattern

- To implement GammaTown requirements I do it *manually*



- *Iteration 1: Weekday.* In this iteration, I add the `weekdayTest` target to my Gradle build script, a manual TestGammaWeekdayRate Java main program that uses the Hamcrest library to test a AlternatingRateStrategy and has a single *Representative Data* test case for the linear rate during weekdays. As it fails due to a missing AlternatingRateStrategy I create it, add the first linear rate subordinate object and delegate the calculation to it if it is not weekend. *Step 4: Run all tests and see them all succeed* but only because I actually made this iteration on a Wednesday!
- *Iteration 2: Weekend.* Next, I add a `weekendTest` target, I adjust the clock to next Sunday, add a TestGammaWeekendRate and finally *Triangulate* the implementation of the rate policy.
- *Iteration 3: Integration.* Integration testing poses some special problems that I will discuss in Chapter 12.

But it is bad …

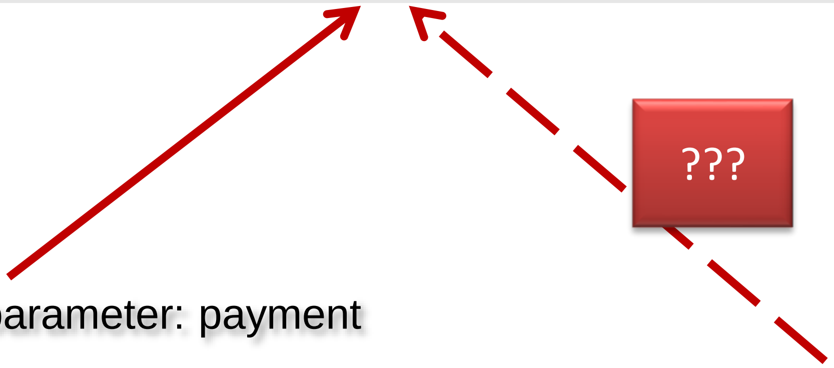
- After introducing Gammatown I no longer have *automated tests* because I have to run some of the tests during the weekend.
  - I have a *‘manual run on weekend and another run on weekdays targets’*

- I want to get back to as much automated testing as possible.

Analysis of Parameters

Fragment: chapter/state/compositional/iteration-2/src/test/java/paystation/manual/TestGammaWeekdayRate.java

```java
    System.out.println( "Manual test of GammaTown Rate for Weekdays");
    RateStrategy rs =
      new AlternatingRateStrategy(new LinearRateStrategy(),
                                  new ProgressiveRateStrategy());
    // Should show 200 minutes for 500 cents
    assertThat(rs.calculateTime(500), is(500/5 * 2));
```



Direct input parameter: payment

Indirect input parameter: day of week

AARHUS UNIVERSITET

Definitions

- This reflection allows me to classify parameters:

Definition: Direct input

Direct input is values or data, provided directly by the testing code, that affect the behavior of the unit under test (UUT).

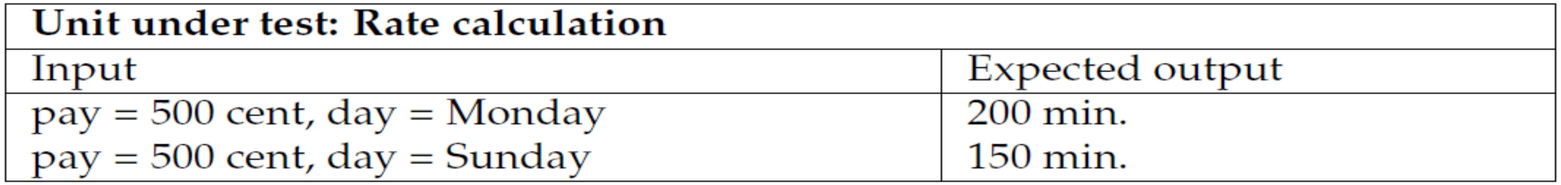
Definition: Indirect input

Indirect input is values or data, that cannot be provided directly by the testing code, that affect the behavior of the unit under test (UUT).

- UUT = Unit Under Test.
  - here it is the AlternatingRateStrategy instance...

Where does indirect input come from?

- So the 1000$ question is: where does the indirect input parameter come from?



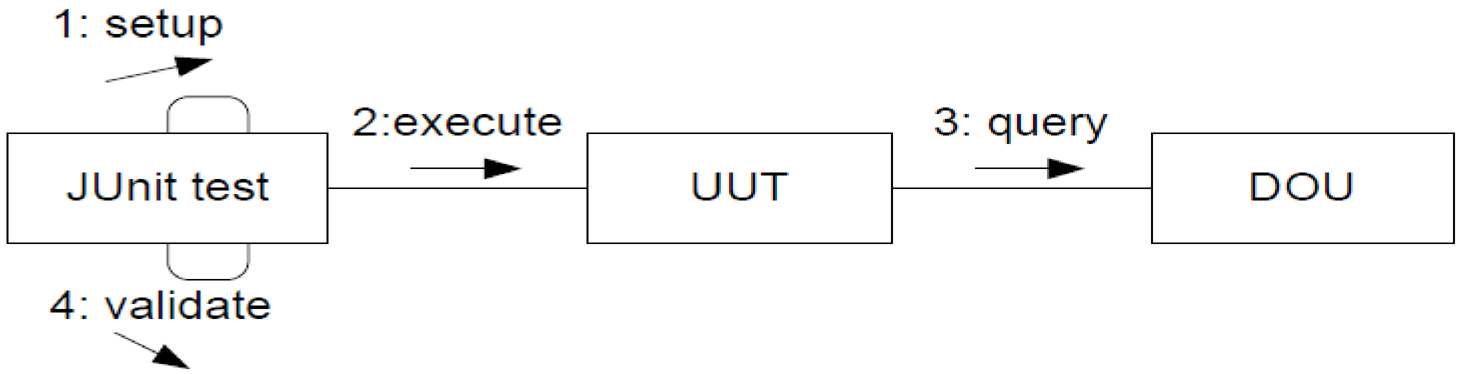
Unit under test: Rate calculation

Input	Expected output
pay = 500 cent, day = Monday	200 min.
pay = 500 cent, day = Sunday	150 min.

- Exercise: Name other types of indirect input?

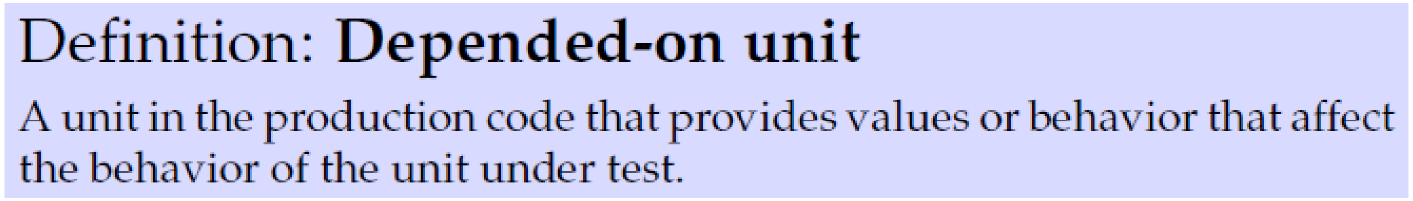
Analysis: Code view

- Structure of xUnit test cases



  - Collaboration diagram: interaction between objects

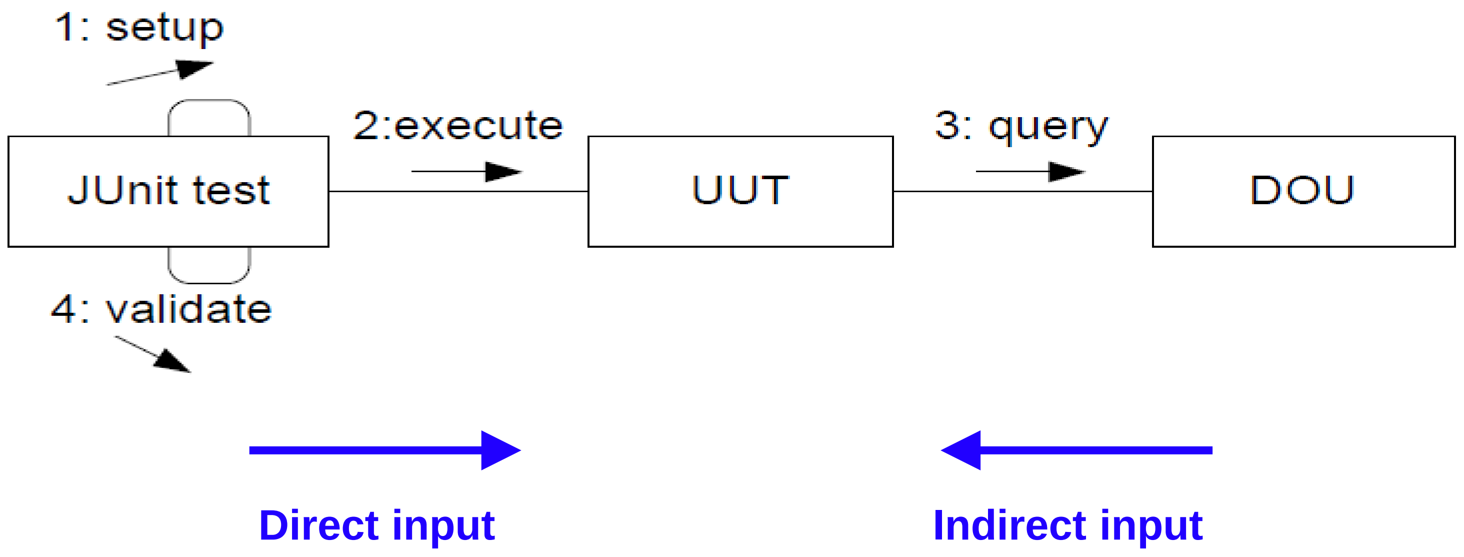
- DOU = Depended On Unit



Definition: **Depended-on unit**

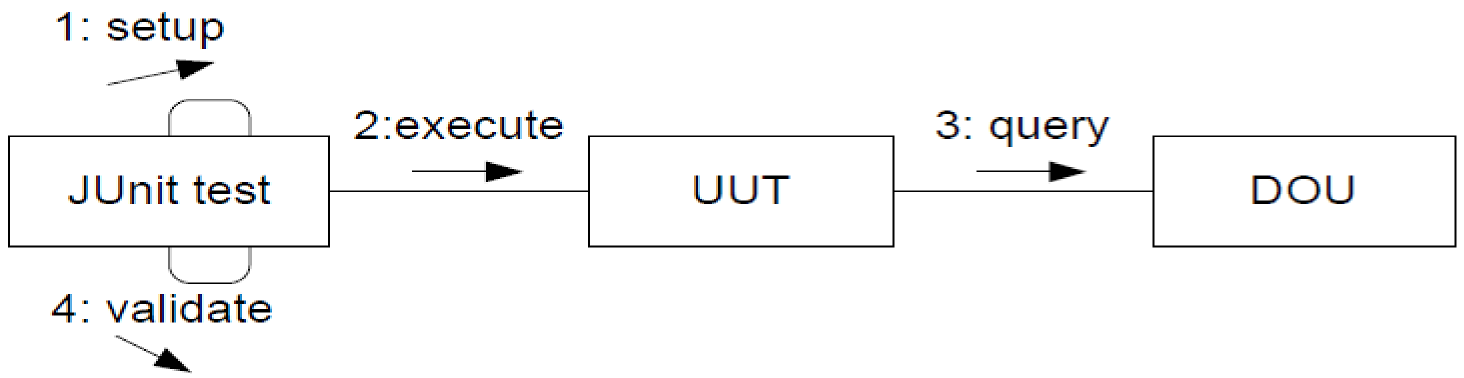
A unit in the production code that provides values or behavior that affect the behavior of the unit under test.

Direct versus Indirect



The Gammatown Rate Policy

- My DOU is the Java system clock:



Test code

AlternatingRateStrategy

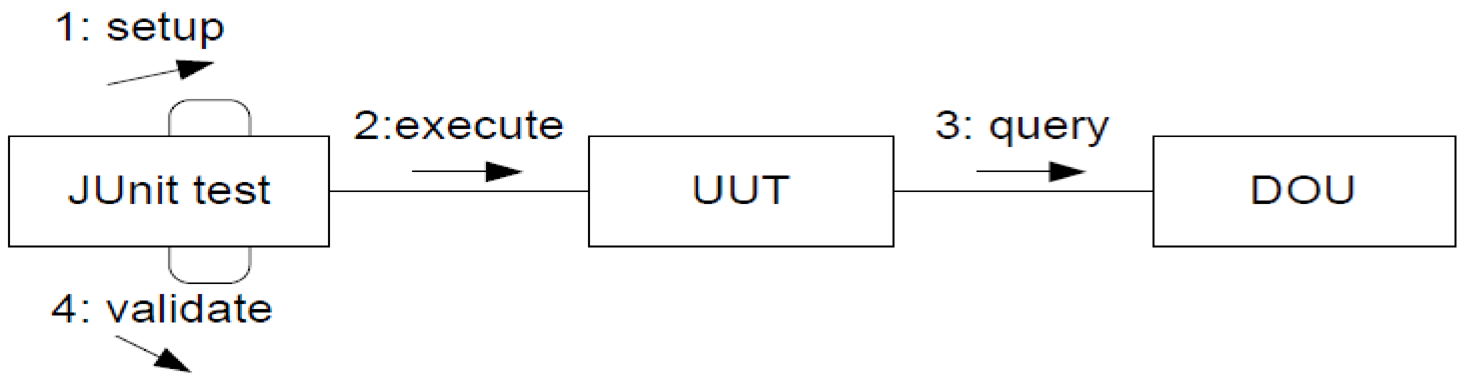
System Clock:
java.time package

Method parameters

Calling library methods

The Challenge

- This analysis allows me to state the challenge:



Test code

AlternatingRateStrategy

System Clock
System.DateTime
java.util.Calendar

- *How can I make the DOU return values that are defined by the testing code?*

AARHUS UNIVERSITET

Analysis

- Basically it is a *variability problem*
  - *During testing*, use data given by test code
  - *During normal operations*, use data given by system

- So I can reuse my previous analysis
  - parametric proposal
  - polymorphic proposal
  - compositional proposal

Scientists like to do this all the time! If we can rephrase a new question into an old one, whose answer is known – then we are done ☺

Parametric

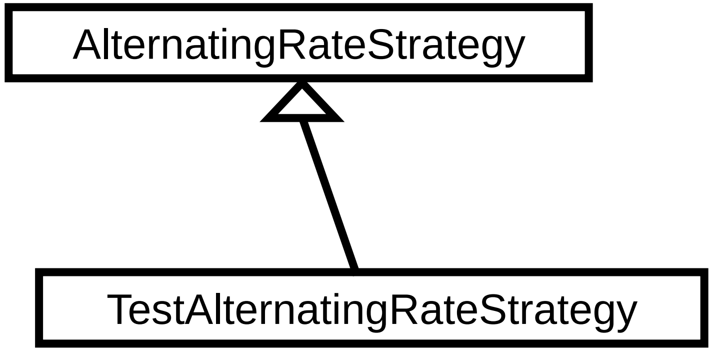
- This is perhaps the oldest solution in the C world

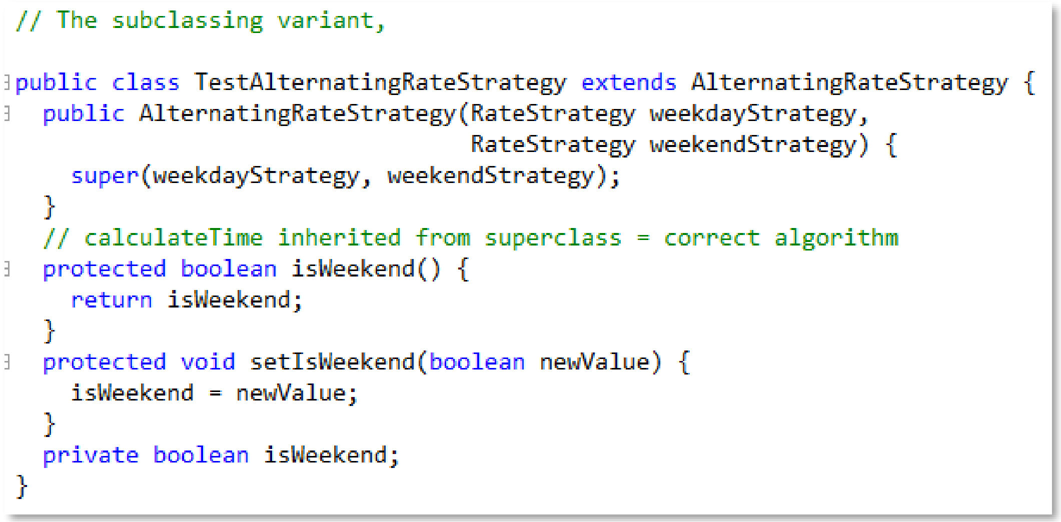
- #ifdef DEBUG
-   today = PRESET_VALUE;
- #else
-   today = (get date from clock);
- #
- return today == Saturday || today == Sunday;

AARHUS UNIVERSITET

Polymorphic

- Subclass or die...



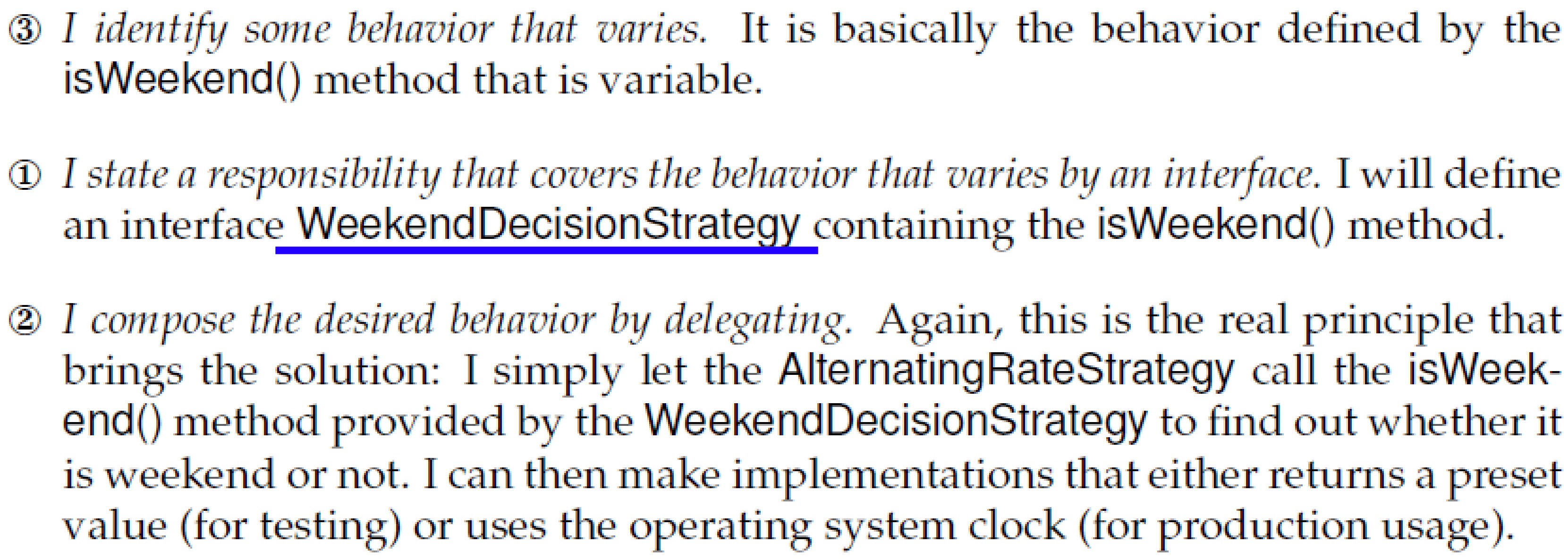


```
// The subclassing variant,

public class TestAlternatingRateStrategy extends AlternatingRateStrategy {
  public AlternatingRateStrategy(RateStrategy weekdayStrategy,
                                 RateStrategy weekendStrategy) {
    super(weekdayStrategy, weekendStrategy);
  }
  // calculateTime inherited from superclass = correct algorithm
  protected boolean isWeekend() {
    return isWeekend;
  }
  protected void setIsWeekend(boolean newValue) {
    isWeekend = newValue;
  }
  private boolean isWeekend;
}
```

- Actually a quite reasonable approach...
  - If you locate the TestAlterna… in the /test tree in the codebase
- Argue why!!!
- Hm, liability: *Have to make isWeekend() ‘non private’*

Compositional

- 3-1-2 leads to yet another Strategy Pattern:

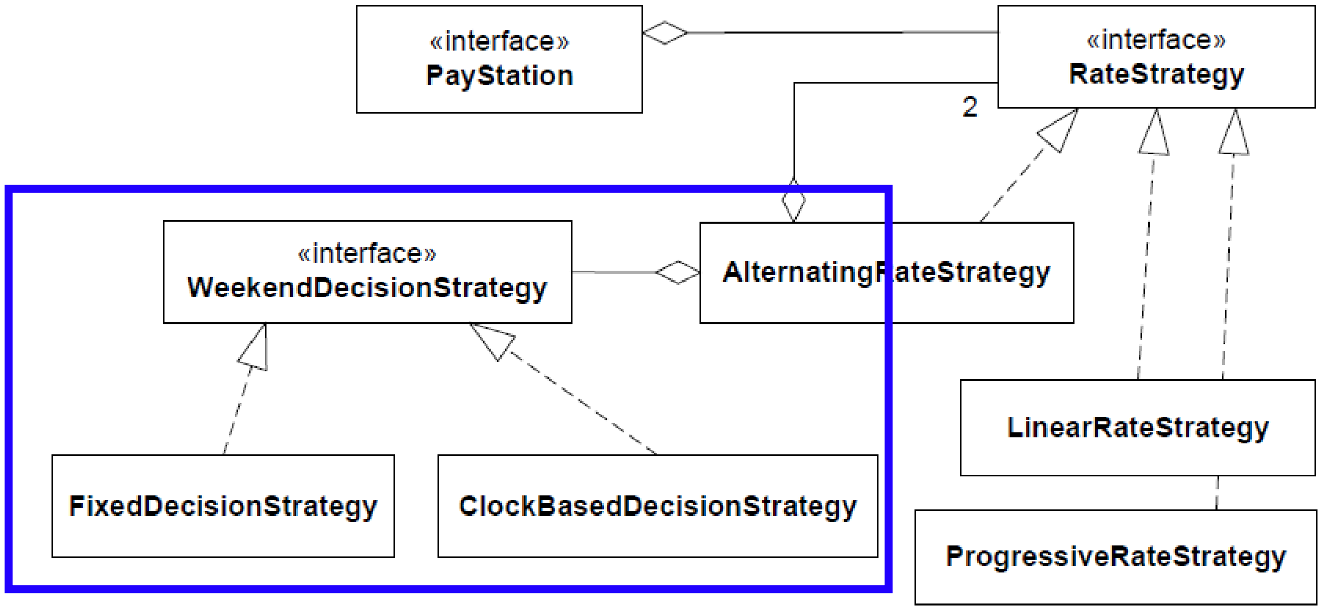


③ *I identify some behavior that varies.* It is basically the behavior defined by the isWeekend() method that is variable.

① *I state a responsibility that covers the behavior that varies by an interface.* I will define an interface WeekendDecisionStrategy containing the isWeekend() method.

② *I compose the desired behavior by delegating.* Again, this is the real principle that brings the solution: I simply let the AlternatingRateStrategy call the isWeekend() method provided by the WeekendDecisionStrategy to find out whether it is weekend or not. I can then make implementations that either returns a preset value (for testing) or uses the operating system clock (for production usage).

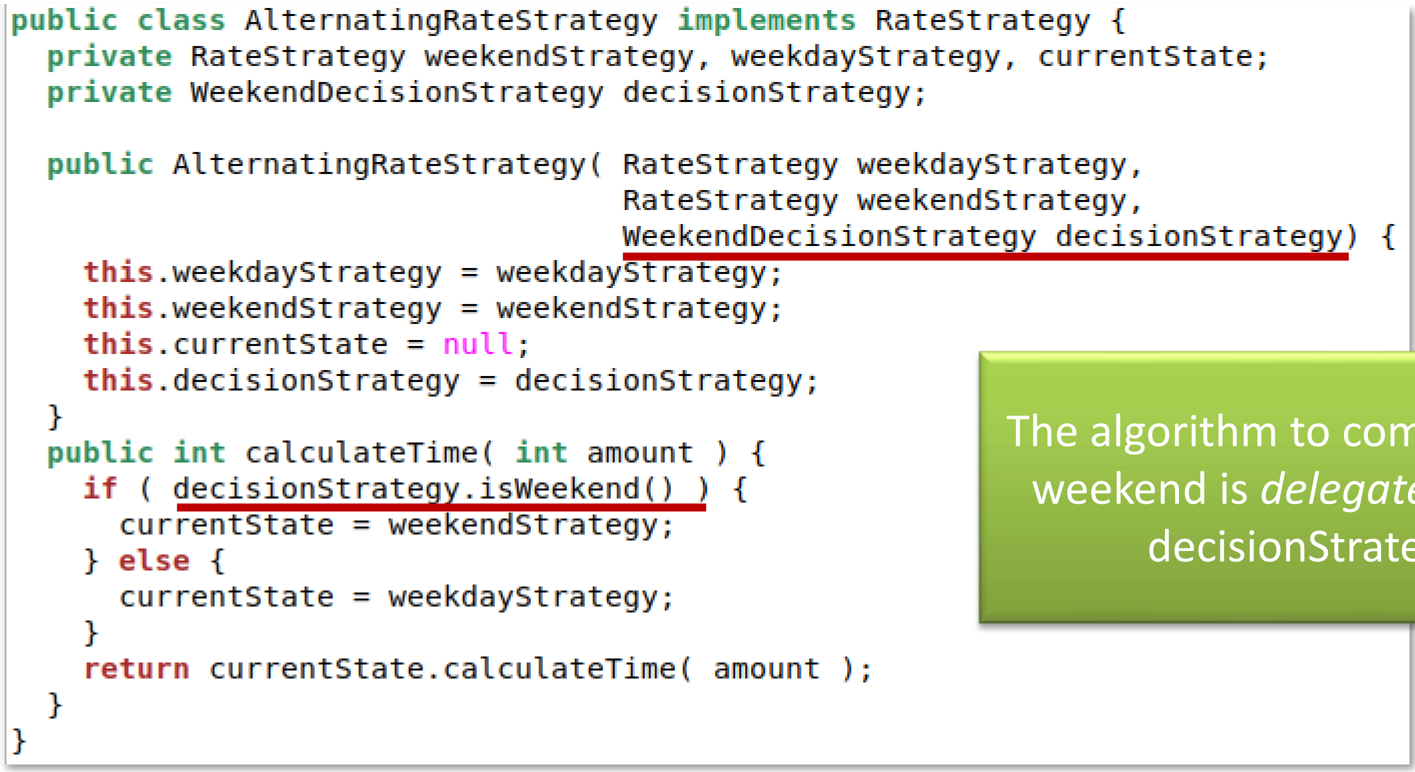
Static Architecture View



- Exercise: Why is this Strategy and not State?

AARHUS UNIVERSITET

Production Code

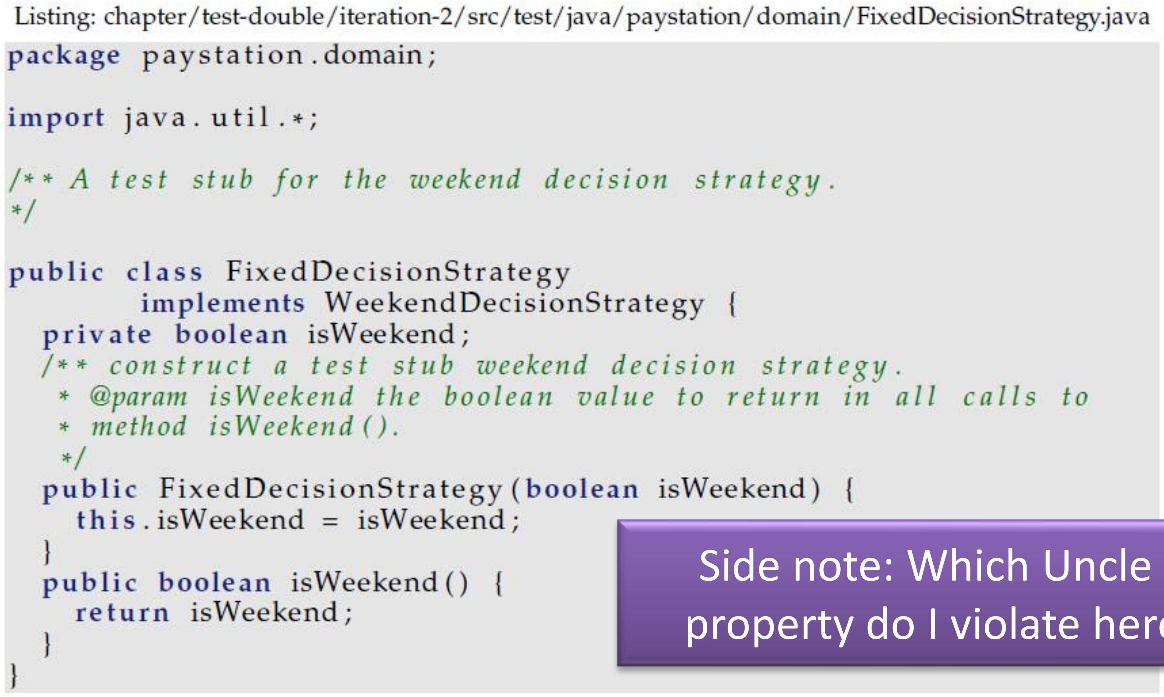


```java
public class AlternatingRateStrategy implements RateStrategy {
  private RateStrategy weekendStrategy, weekdayStrategy, currentState;
  private WeekendDecisionStrategy decisionStrategy;

  public AlternatingRateStrategy( RateStrategy weekdayStrategy,
                                  RateStrategy weekendStrategy,
                                  WeekendDecisionStrategy decisionStrategy) {
    this.weekdayStrategy = weekdayStrategy;
    this.weekendStrategy = weekendStrategy;
    this.currentState = null;
    this.decisionStrategy = decisionStrategy;
  }
  public int calculateTime( int amount ) {
    if ( decisionStrategy.isWeekend() ) {
      currentState = weekendStrategy;
    } else {
      currentState = weekdayStrategy;
    }
    return currentState.calculateTime( amount );
  }
}
```

The algorithm to compute if its weekend is *delegated* to our decisionStrategy

The Stub

- To make a deterministic test; we write an implementation which makes the 'indirect input' into 'direct input'
  - That is, we get the 'is-it-weekend' under direct control of our test code

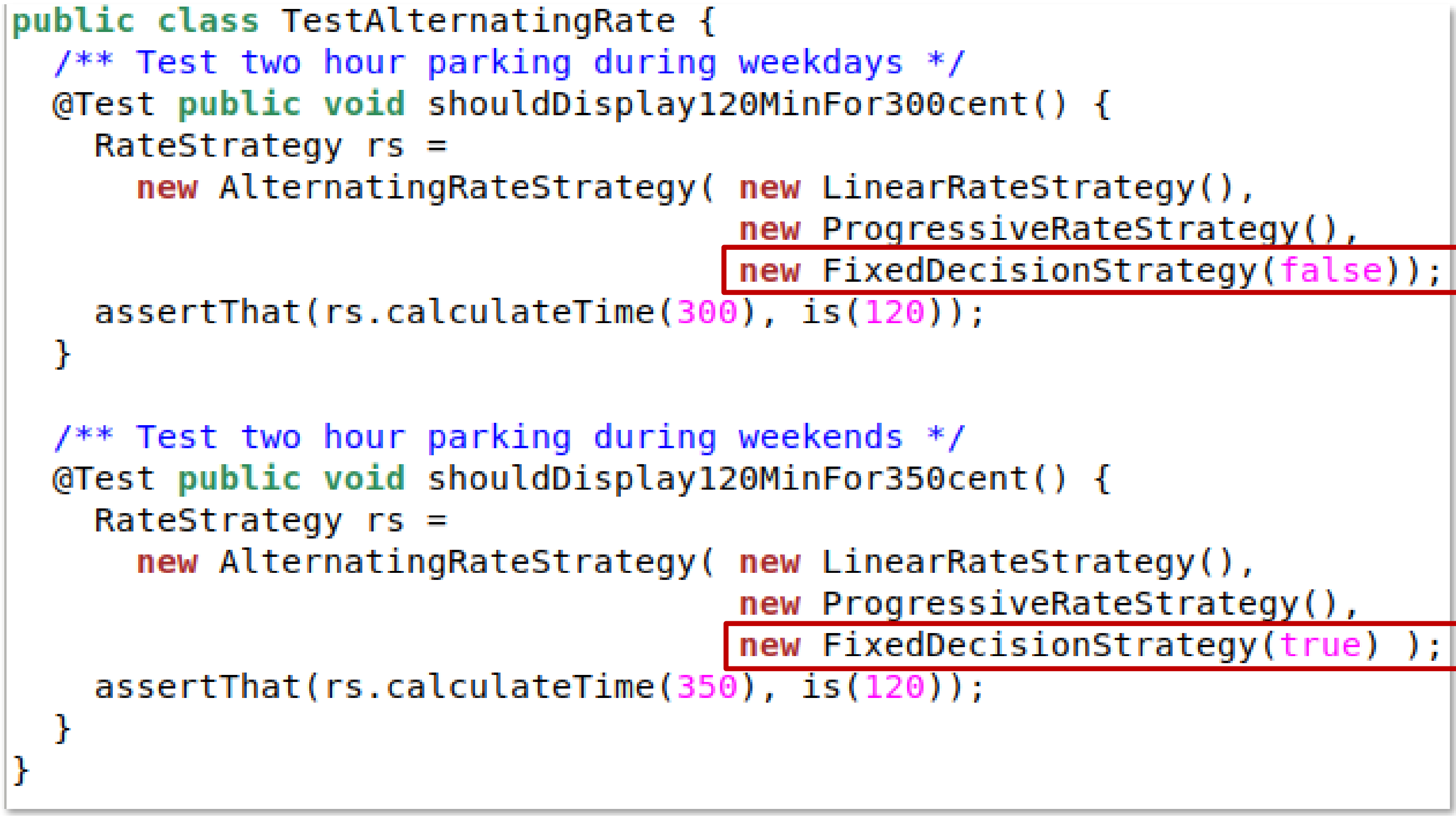

Listing: chapter/test-double/iteration-2/src/test/java/paystation/domain/FixedDecisionStrategy.java

```java
package paystation.domain;

import java.util.*;

/** A test stub for the weekend decision strategy.
*/

public class FixedDecisionStrategy
        implements WeekendDecisionStrategy {
  private boolean isWeekend;
  /** construct a test stub weekend decision strategy.
   * @param isWeekend the boolean value to return in all calls to
   * method isWeekend().
   */
  public FixedDecisionStrategy(boolean isWeekend) {
    this.isWeekend = isWeekend;
  }
  public boolean isWeekend() {
    return isWeekend;
  }
}
```

Side note: Which Uncle Bob property do I violate here ☹?

AARHUS UNIVERSITET

Now the Test Code is:



```
public class TestAlternatingRate {
  /** Test two hour parking during weekdays */
  @Test public void shouldDisplay120MinFor300cent() {
    RateStrategy rs =
      new AlternatingRateStrategy( new LinearRateStrategy(),
                                   new ProgressiveRateStrategy(),
                                   new FixedDecisionStrategy(false));
    assertThat(rs.calculateTime(300), is(120));
  }

  /** Test two hour parking during weekends */
  @Test public void shouldDisplay120MinFor350cent() {
    RateStrategy rs =
      new AlternatingRateStrategy( new LinearRateStrategy(),
                                   new ProgressiveRateStrategy(),
                                   new FixedDecisionStrategy(true) );
    assertThat(rs.calculateTime(350), is(120));
  }
}
```

Rephrasing as Test Case

Input	Expected output
pay = 300 cent, day = Wednesday	120 min.

can be rephrased

Input	Expected output
pay = 300 cent, day-type = weekday	120 min.

Fragment: chapter/test-double/iteration-2/src/test/java/paystation/domain/TestGammaWeekdayRate.java

```
@Test public void shouldDisplay120MinFor300cent() {
  RateStrategy rs =
    new AlternatingRateStrategy( new LinearRateStrategy(),
                                 new ProgressiveRateStrategy(),
                                 new FixedDecisionStrategy(false));
  assertThat(rs.calculateTime(300), is(300 / 5 * 2));
}
```

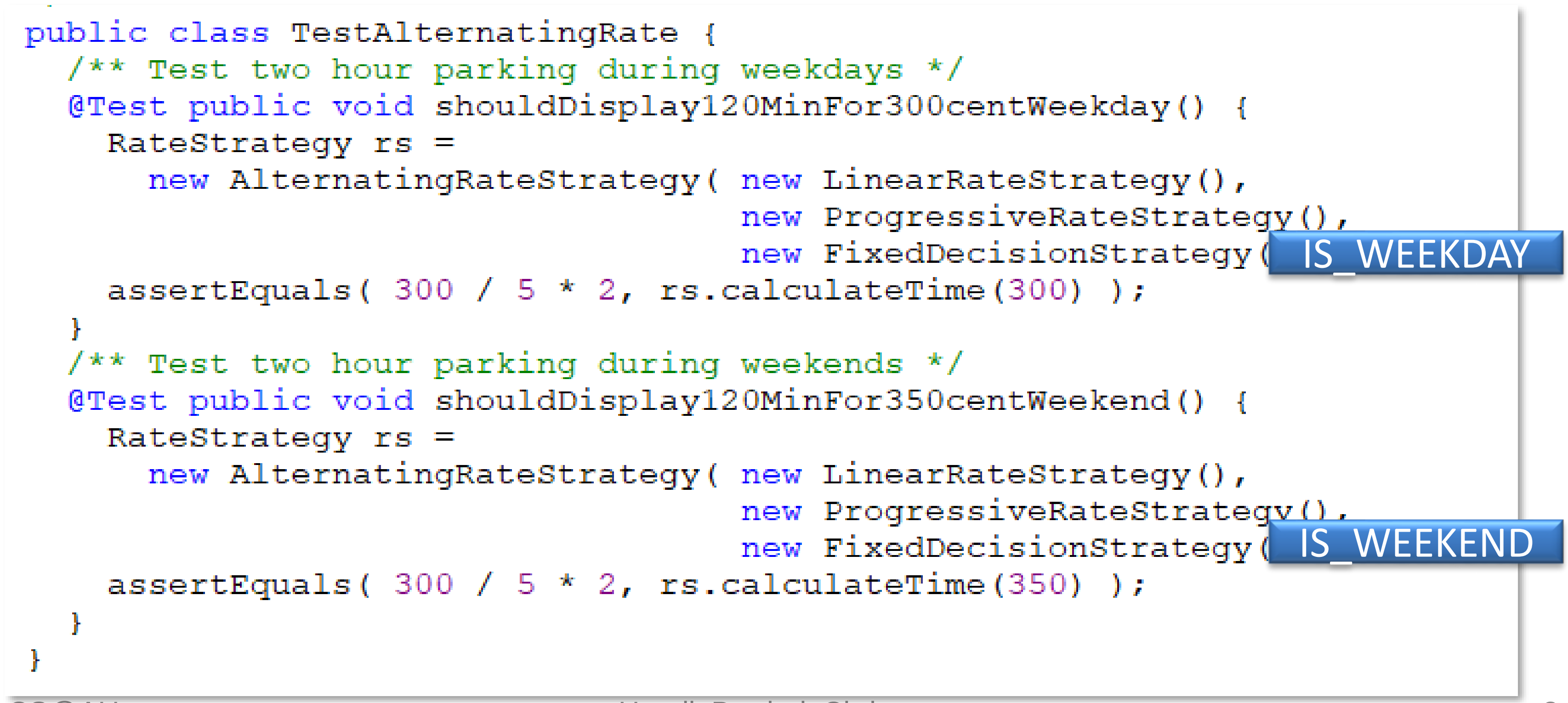
Direct input parameter: payment

Now: **Direct input** parameter: weekend or not

AARHUS UNIVERSITET

Side note: Sorry Bob ☺

- On my ToDo … introduce an Enum type
  - No flag argument, replaced by *descriptive names*



```
public class TestAlternatingRate {
   /** Test two hour parking during weekdays */
   @Test public void shouldDisplay120MinFor300centWeekday() {
     RateStrategy rs =
       new AlternatingRateStrategy( new LinearRateStrategy(),
                                    new ProgressiveRateStrategy(),
                                    new FixedDecisionStrategy( IS_WEEKDAY
     assertEquals( 300 / 5 * 2, rs.calculateTime(300) );
   }
   /** Test two hour parking during weekends */
   @Test public void shouldDisplay120MinFor350centWeekend() {
     RateStrategy rs =
       new AlternatingRateStrategy( new LinearRateStrategy(),
                                    new ProgressiveRateStrategy(),
                                    new FixedDecisionStrategy( IS_WEEKEND
     assertEquals( 300 / 5 * 2, rs.calculateTime(350) );
   }
}
```

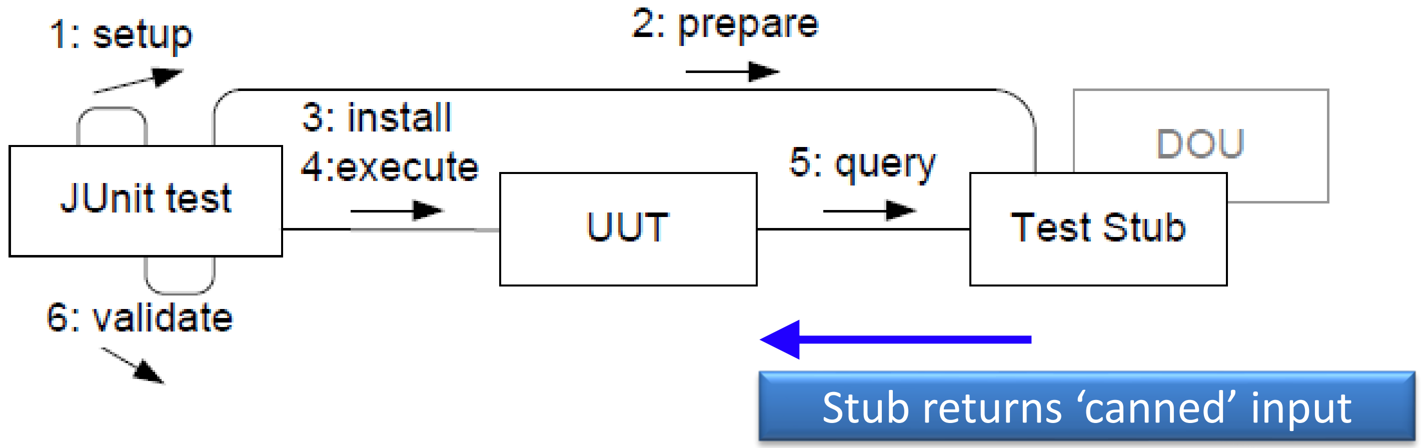
AARHUS UNIVERSITET

Test Stub

- The new delegate is an example of a **test stub**

Definition: Test stub

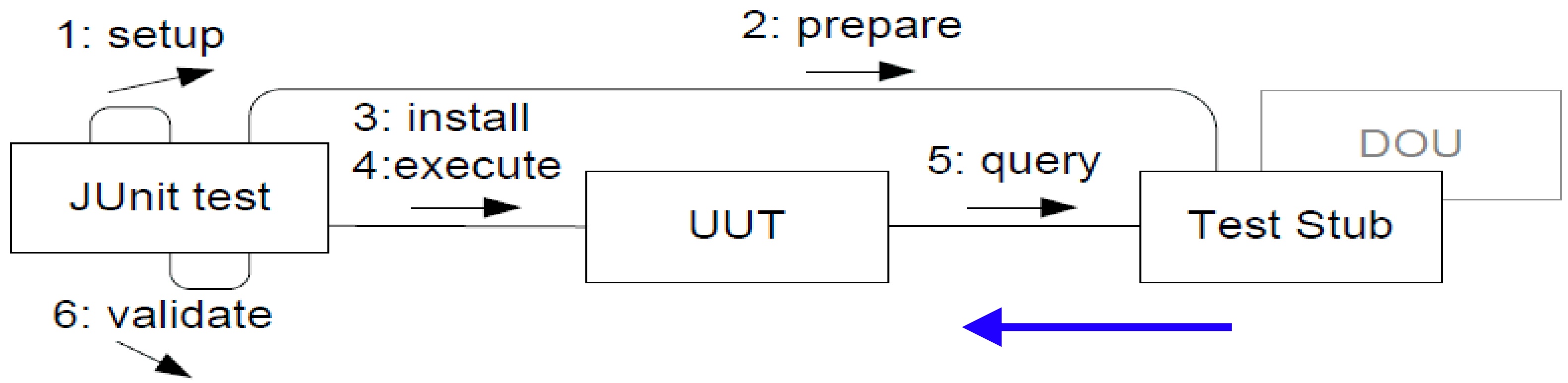
A test stub is a replacement of a real *depended-on unit* that feeds indirect input, defined by the test code, into the *unit under test*.



AARHUS UNIVERSITET

UUT Queries served by Stubs

- Test Stubs serve ***queries (accessors)*** by the UUT



- Stubs are *simple implementations* (‘Evident Tests’)
- Stubs return *canned or configured input* to UUT
  - ‘setNextValueToReturn(3);’ return nextValue;
  - return 3;

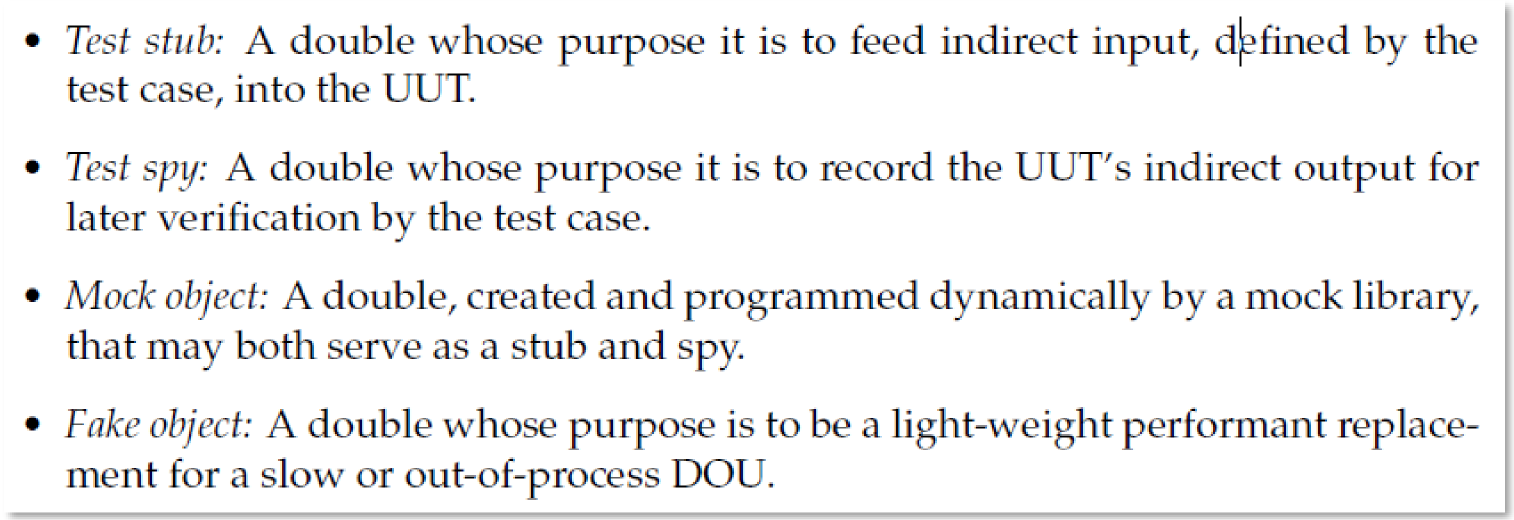
Test Doubles

The Stub is just one type of ‘replacement delegate’

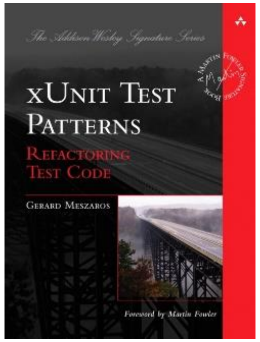
The superclass: Double

Meszaros (2007)

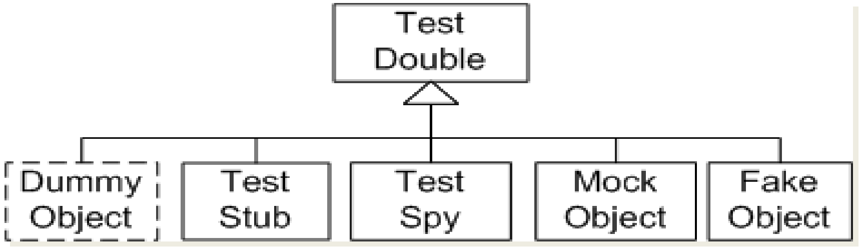
- There are actually several types of 'replacements'…



- *Test stub:* A double whose purpose it is to feed indirect input, defined by the test case, into the UUT.
- *Test spy:* A double whose purpose it is to record the UUT's indirect output for later verification by the test case.
- *Mock object:* A double, created and programmed dynamically by a mock library, that may both serve as a stub and spy.
- *Fake object:* A double whose purpose is to be a light-weight performant replacement for a slow or out-of-process DOU.

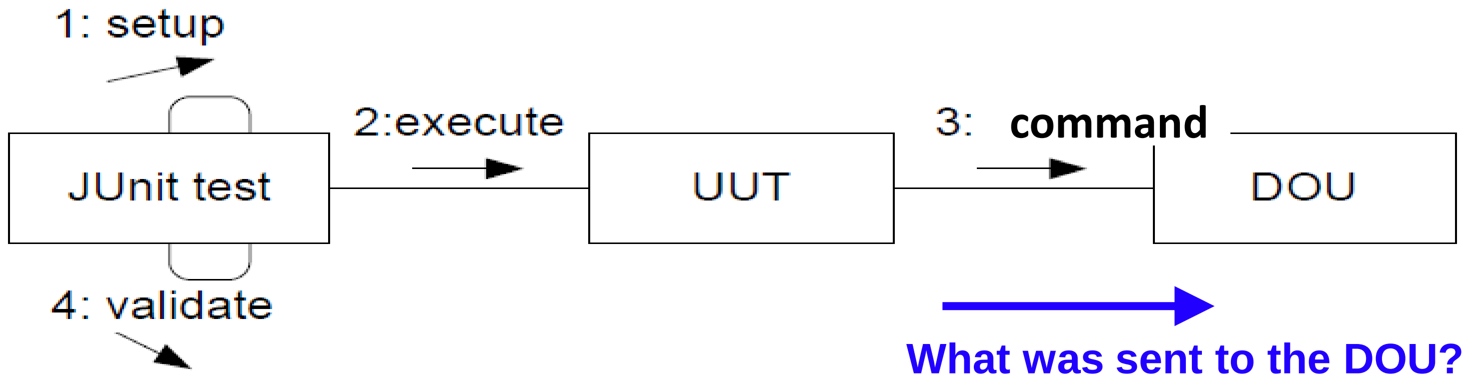

The Addison Wesley Signature Series
xUnit Test Patterns
Refactoring Test Code
Gerard Meszaros
Foreword by Martin Fowler

Double???
From the term 'stunt double' in movie making



Test Spy

- Spies serve ***commands (mutators)*** by the UUT



- Spies are *recorders* of interaction
  - So JUnit test can *later query the spy about "what happened?"*

- Again, simple implementations ('Evident Tests')

Example

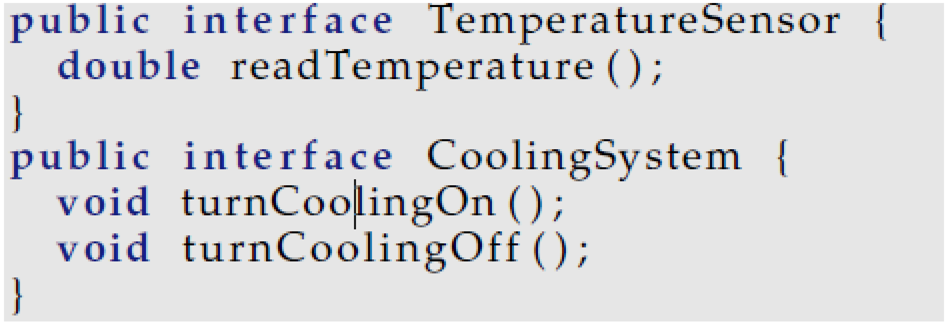
- Chemical plant
  - Control temperature in chemical process
- Algorithm
  - Measure the temperature
    - *Query the temperature sensor*
  - Compute a response
    - If (T > 67) then cool the process; if (T < 62) then stop cooling;
  - Activate the cooling system
    - *Command the cooler to turn On*
- **Manual testing:**
  - Let the process run; if it explodes then the test has failed ☺

AARHUS UNIVERSITET

The UUT

- The UUT (Unit-Under-Test) is of course the algorithm, the monitoring of the chemical process
  - Compute a response
    - Measure T
    - If (T > 67) then Turn on Cooling

- But there are *two DOUs* involved
  - TemperaturSensor
  - CoolingSystem

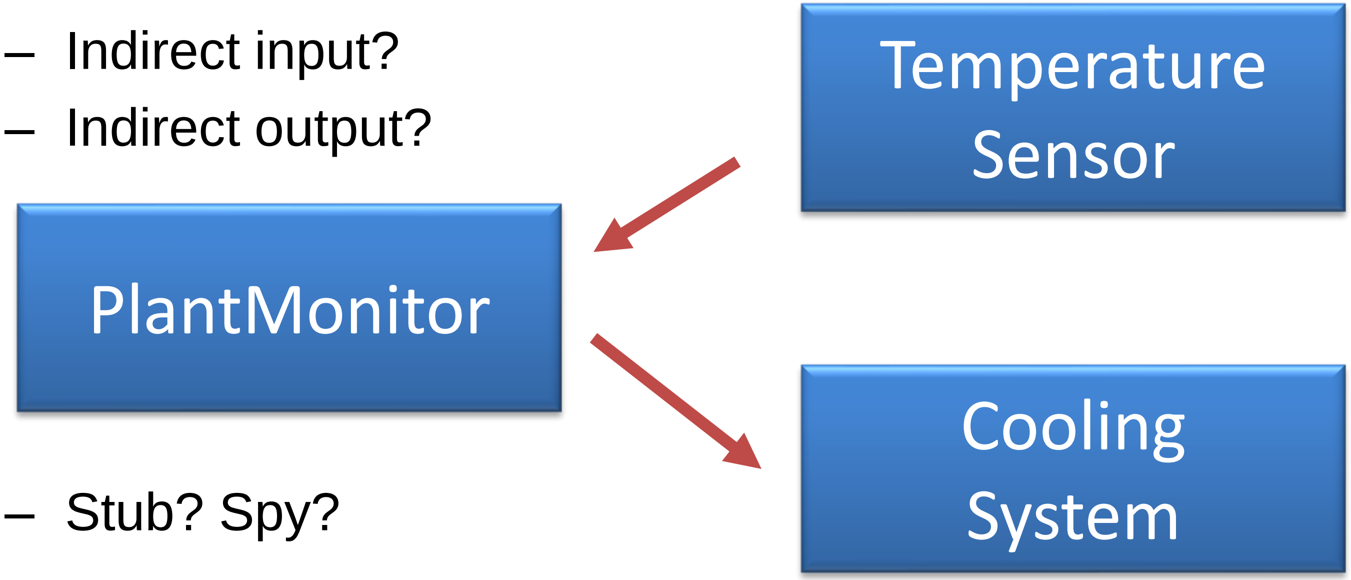


```
public interface TemperatureSensor {
  double readTemperature();
}
public interface CoolingSystem {
  void turnCoolingOn();
  void turnCoolingOff();
}
```

AARHUS UNIVERSITET

The Test Doubles

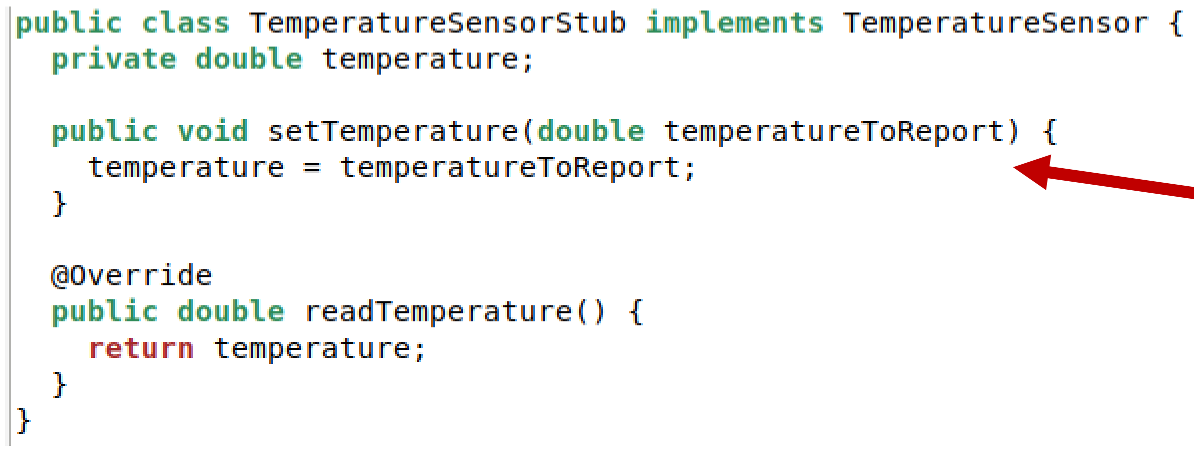
- So we need two test doubles
- ***Exercise:***
  - Indirect input?
  - Indirect output?



  - Stub? Spy?

The Stub, you all know now

- Stub: *Simple implementation, returning indirect output that is either canned or configured.*
  - We want to control the indirect output, so we just provide a method to configure it



```java
public class TemperatureSensorStub implements TemperatureSensor {
  private double temperature;

  public void setTemperature(double temperatureToReport) {
    temperature = temperatureToReport;
  }

  @Override
  public double readTemperature() {
    return temperature;
  }
}
```

Note: No '@Override'. It is a method *just implemented in the stub!*

So: A TestCase

- **Given** T > 67 Celcius, **When** asked to monitor, **Then** cooling is turned on

Fragment: chapter/test-double/spy/src/test/java/chemicalplant/TestTemperatureRegulation.java

```
@Test
public void shouldTurnOnCoolingAbove67degrees() {
  // Given a temperature above 67
  temperatureSensor.setTemperature(67.2);
  // When the monitor needs to regulate the temperature
  plantMonitor.regulateTemperature();
  // Then cooling is commanded to turn on cooling
  assertThat(coolingSystem.lastMethodCalled(), is("turnCoolingOn"));
}
```

AARHUS UNIVERSITET

Spy

- Spies are *recorders* of interaction
  - So JUnit test can *later query the spy about "what happened"*

Fragment: chapter/test-double/spy/src/test/java/chemicalplant/CoolingSystemSpy.java

```
public class CoolingSystemSpy implements CoolingSystem {
  private String lastCalledMethod = "none";
  @Override
  public void turnCoolingOn() {
    lastCalledMethod = "turnCoolingOn";
  }

  @Override
  public void turnCoolingOff() {
    lastCalledMethod = "turnCoolingOff";
  }

  public String lastMethodCalled() {
    return lastCalledMethod;
  }
}
```

Note: No '@Override'. It is a method *just implemented in the spy!*

As used in…

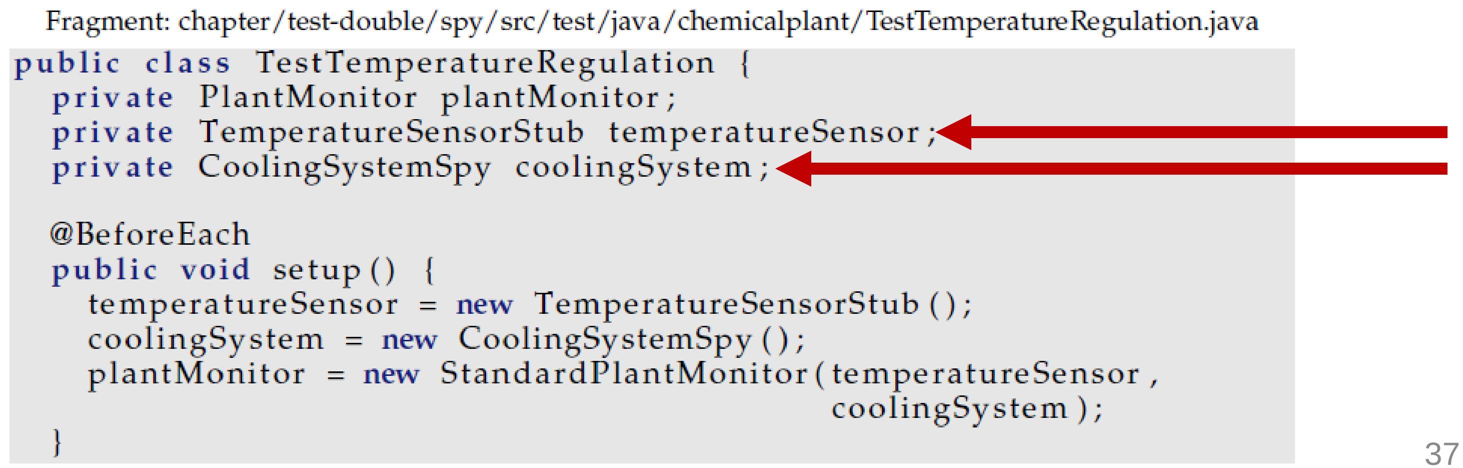
- Validate that the cooling was turned on…

Fragment: chapter/test-double/spy/src/test/java/chemicalplant/TestTemperatureRegulation.java

```
@Test
public void shouldTurnOnCoolingAbove67degrees() {
  // Given a temperature above 67
  temperatureSensor.setTemperature(67.2);
  // When the monitor needs to regulate the temperature
  plantMonitor.regulateTemperature();
  // Then cooling is commanded to turn on cooling
  assertThat(coolingSystem.lastMethodCalled(), is("turnCoolingOn"));
}
```

Retrieval Interfaces

- **Retrieval Interfaces:** *Special methods for setting and inspecting state in doubles, only defined in the test double classes themselves!*
  - I.e. the real temperature sensor should of course not have a method to set the temperature, right?
- Thus *doubles* are often declared by class, not interface



Fragment: chapter/test-double/spy/src/test/java/chemicalplant/TestTemperatureRegulation.java

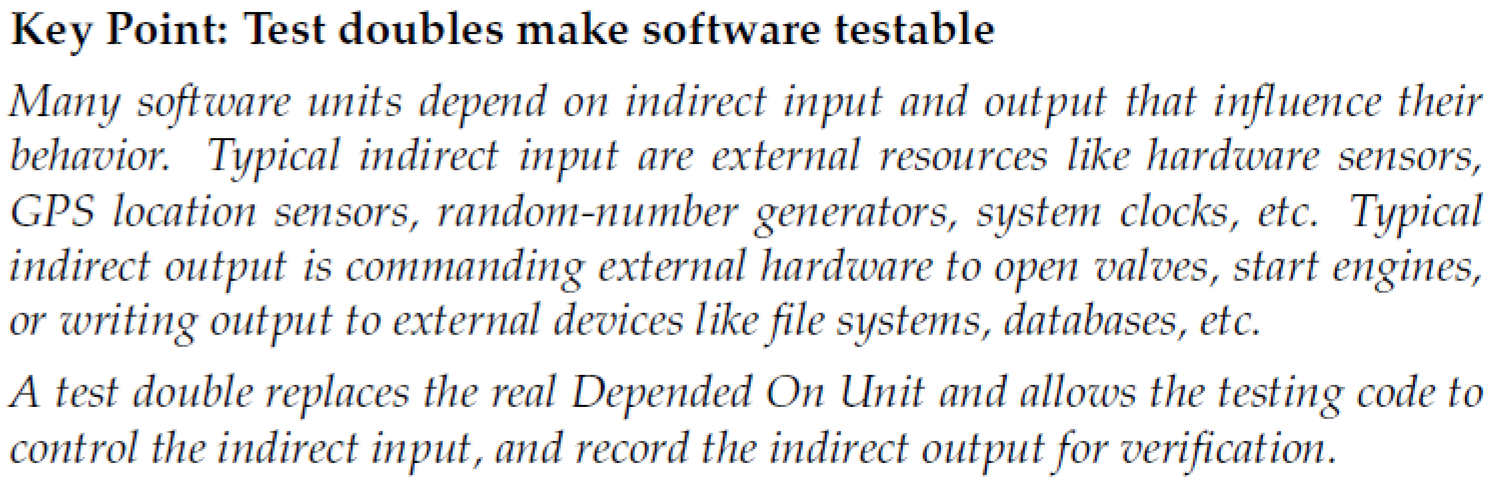
```
public class TestTemperatureRegulation {
  private PlantMonitor plantMonitor;
  private TemperatureSensorStub temperatureSensor;
  private CoolingSystemSpy coolingSystem;

  @BeforeEach
  public void setup() {
    temperatureSensor = new TemperatureSensorStub();
    coolingSystem = new CoolingSystemSpy();
    plantMonitor = new StandardPlantMonitor(temperatureSensor,
                                            coolingSystem);
  }
```

Side Note

- Retrieval interface are
  - *"The role that the object must play, as seen from the test perspective"*
  - It is a specific role that is only related to testing
- As such it could be designed by a
- **Role Interface / Private Interface**
- … as introduced later in the course ☺

AARHUS UNIVERSITET

Key point



Key Point: Test doubles make software testable

Many software units depend on indirect input and output that influence their behavior. Typical indirect input are external resources like hardware sensors, GPS location sensors, random-number generators, system clocks, etc. Typical indirect output is commanding external hardware to open valves, start engines, or writing output to external devices like file systems, databases, etc.

A test double replaces the real Depended On Unit and allows the testing code to control the indirect input, and record the indirect output for verification.

- Allow us to test the nuclear reactor core control software without doing the ‘Tjernobyl test’…

Note

- Please note that once again the 3-1-2 is the underlying and powerful engine for *Test Doubles*.
  - *Encapsulate the temperature sensor that (3) varies, by defining an interface (1), and then use delegation (2) to let ‘someone else read the temperature’*

- I use the 3-1-2 to *derive* a solution that “accidentally” has a name and is a well known concept; just as I previously derived several design patterns.

AARHUS UNIVERSITET

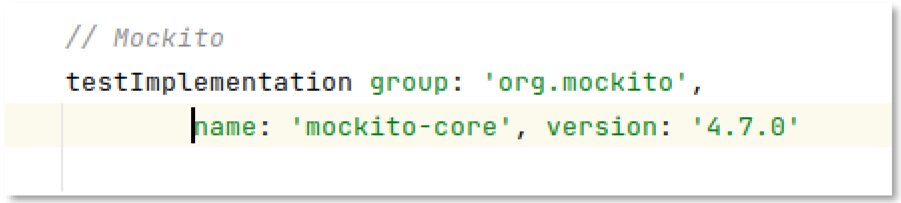
Fake Object

- ... Is not needed in SWEA
- They are *light-weight, performant, replacements for slow or out-of-process DOUs*
- Examples
  - Replacing a database with a in-memory hashmap
  - Replacing a REST service with a simple in-memory impl.

- Both are *out-of-process* – that is you have to start an external service (a DB, a web server) which is difficult from within JUnit

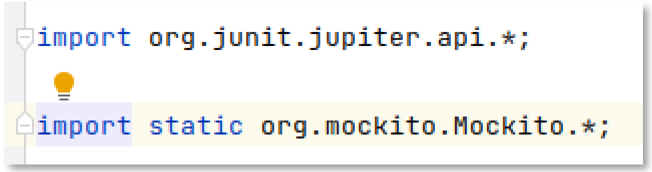
Mocks

- … are “even more not used” in SWEA (yet)
- Mocks are *auto-generated doubles*, made by libraries.
- Example: Mockito
  - You need to tell Gradle to pull the library, of course…



```
// Mockito
testImplementation group: 'org.mockito',
        name: 'mockito-core', version: '4.7.0'
```

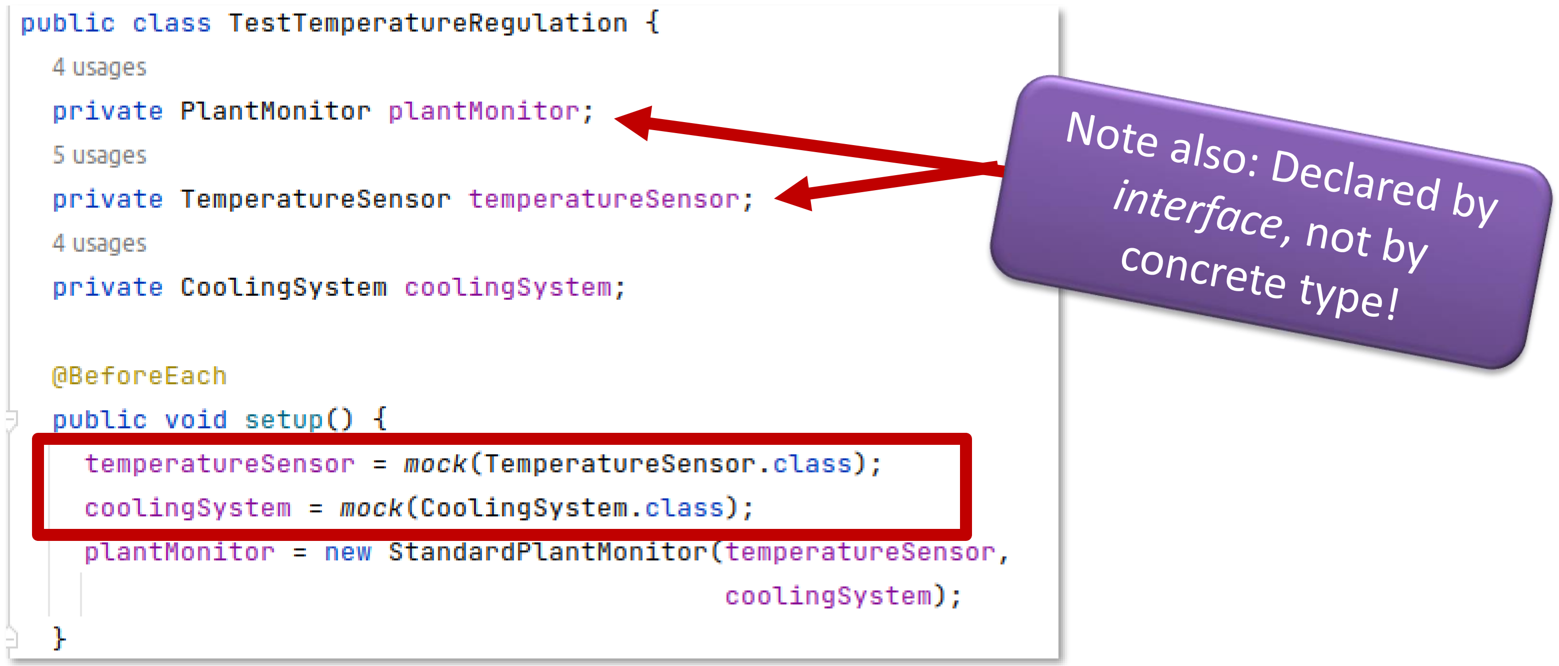
  - … which allows you to



```
import org.junit.jupiter.api.*;

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

Mocks

- Creating your stub/spy is easy, just tell Mockito to do it!

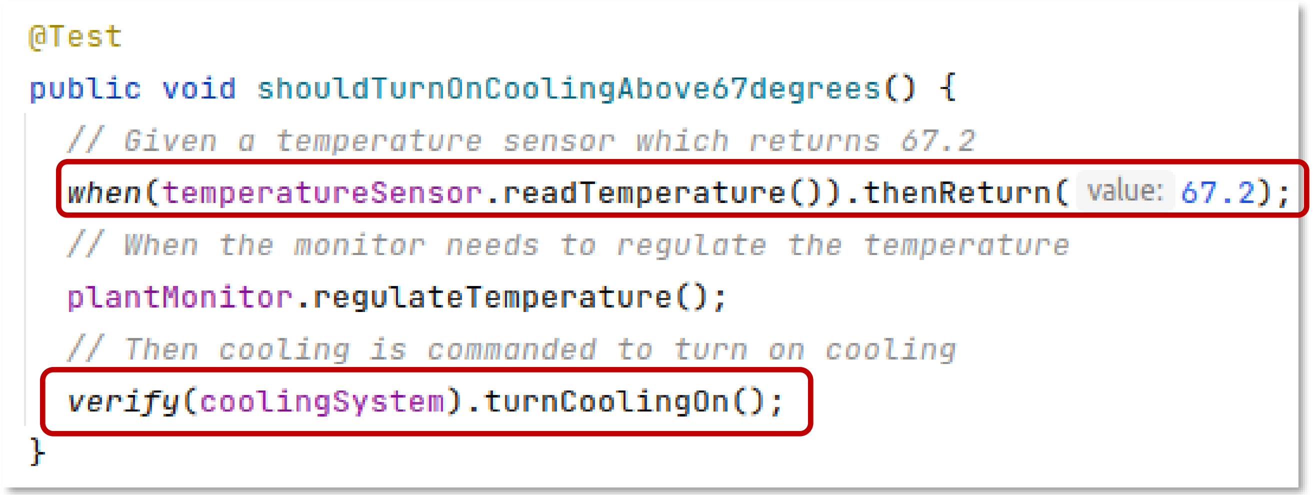


```
public class TestTemperatureRegulation {
  4 usages
  private PlantMonitor plantMonitor;
  5 usages
  private TemperatureSensor temperatureSensor;
  4 usages
  private CoolingSystem coolingSystem;

  @BeforeEach
  public void setup() {
    temperatureSensor = mock(TemperatureSensor.class);
    coolingSystem = mock(CoolingSystem.class);
    plantMonitor = new StandardPlantMonitor(temperatureSensor,
                                            coolingSystem);
  }
```

AARHUS UNIVERSITET

Mocks

- Using Mocks you “program” your stub and spy behavior using the Mockito API, not by coding Java.



```
@Test
public void shouldTurnOnCoolingAbove67degrees() {
    // Given a temperature sensor which returns 67.2
    when(temperatureSensor.readTemperature()).thenReturn( value: 67.2);
    // When the monitor needs to regulate the temperature
    plantMonitor.regulateTemperature();
    // Then cooling is commanded to turn on cooling
    verify(coolingSystem).turnCoolingOn();
}
```

Mocks or Not

- Personally, I am a bit torn on ‘to use or not?’
- The benefit
  - “Quickly” add a test – I just say ‘mock(Database.class)’ and I have a stub + spy for it…
  - Quite elaborate verifications possible
    - Ordering, never, 10 times, any…

```
verify(mockedList, never()).clear();
```

```
InOrder inOrder = Mockito.inOrder(mockedList);
inOrder.verify(mockedList).size();
inOrder.verify(mockedList).add("a parameter");
inOrder.verify(mockedList).clear();
```

```
verify(mockedList, atLeast(1)).clear();
verify(mockedList, atMost(10)).clear();
```

```
verify(mockedList).add("test");
```

```
verify(mockedList).add(anyString());
```

AARHUS UNIVERSITET

Mocks or Not

- The liabilities
  - I am **not programming in Java!!!**
    - I am coding in obscure when()/verify() syntax
      - No help from IntelliJ
      - No help from 25+ years of experience
      - I often find myself trial/error coding – **It is not 'evident test'**
  - Vendor Lock-in = I am stuck with a specific library
    - Changing to EasyMock or jMock? Bad luck, rewrite **all your tests!**
  - The Mockist approach slippery slope into ***white-box tests***
    - Tendency to test **How things are done**, not **What** was done…
- So – use it with care…
  - (I did my EtaStone tests using Mockito, though, and loved it ☺)

Reusing the variability points...

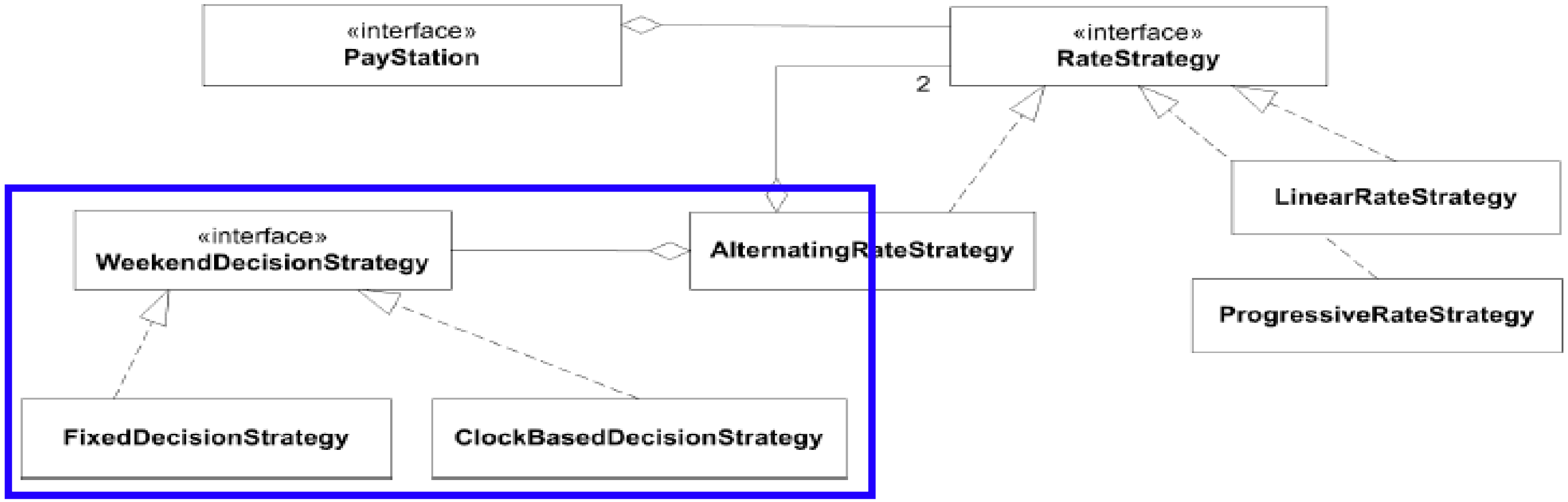
Aah – I could do this...

Variability points to the rescue

- The WeekendDecisionStrategy introduces yet another variability point...

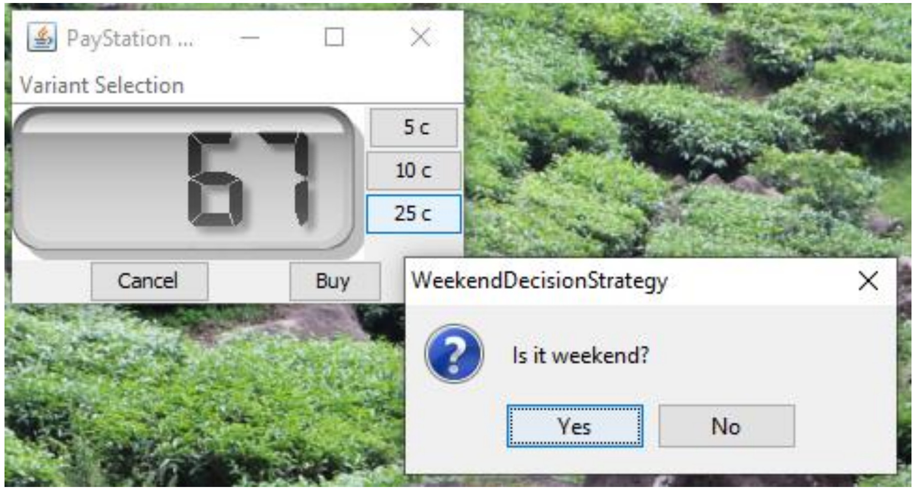
- Often they come in handy later **if**
  - **1) they encapsulate well-defined responsibilities**
  - **2) are defined by interfaces and**
  - **3) uses delegation ☺**

Static Architecture View



Manual testing

- Manual testing of GammaTown, for *demo* to end users!



```
public class DialogDecisionStrategy implements WeekendDecisionStrategy {
  public boolean isWeekend() {
    return
      JOptionPane.YES_OPTION
      ==
      JOptionPane.showConfirmDialog(null,
                                    "Is it weekend?",
                                    "WeekendDecisionStrategy",
                                    JOptionPane.YES_NO_OPTION );
  }
}
```

AARHUS UNIVERSITET

Discussion

Package/Namespace View

- Gradle dictate that we split the code into two trees
  - src/main/java: all production code rooted here
  - src/test/java: all test code rooted here
- Here
  - WeekendDecisionStrategy (interface)
  - ClockBasedDecisionStrategy (class)
  - FixedDecisionStrategy (class)
- Exercise: Where would you put these units?

C# Delegates / Java 8 Lambda

- The WeekendDecisionStrategy only contains a single method and having an interface may seem a bit of an overkill.
  - In Java 8, you can use a *Lambda*
  - In C# you may use *delegates* that is more or less a type safe *function pointer.*
  - In functional languages you may use higher order functions, closures

AARHUS UNIVERSITET

Summary

Key Points

- *Test Doubles make software testable.*

- 3-1-2 technique help isolating DOUs
  - because I isolated the responsibility by an interface I had the opportunity to delegate to a test stub

- My solution is overly complex to our weekend issue
  - Yes! Perhaps subclassing in test tree would be better here ☺
  - **But**
    - it scales well to complex DOUs
    - it is good at handling aspects that may vary across the entire system (see next slide)

This is a *PowerTool*

- **Test Doubles** usage are a key technique in modern, microservice, continuous deployment, development!!!
  – Build servers that automatically pull git repositories for newest releases, runs extensive tests, and finally pushes code into production on the production servers…
- **It would not be possible if stubs, spies, fake objects, mocks were not used to thoroughly test using automated testing!**
- Example:
  – NetFlix need to survive server crashes to continue streaming
    - Test stubs (‘saboteurs’) throw IOExceptions to simulate failures…

AARHUS UNIVERSITET

Still Untested Code

- Some code units are not automatically testable in a cost-efficient manner
  - Note that if I rely on the automatic tests only, then the ClockBasedDecisionStrategy instance **is never tested!**
    - (which it actually was when using the manual tests!)

- Thus:
  - DOUs handling external resources must still be manually tested (and/or formally reviewed by *software reviews*).
  - Keep ‘non-testable code’ in the smallest possible software unit, and **if it ain’t broke, then don’t fix it ☺**

Know When to Stop Testing

- Note also that I do not test that the return values from the system library methods are not tested.

- I expect Oracle / MicroSoft to test their software.
  - sometimes we are wrong but it is not cost efficient.

- *Do not test the random generator ☺*