AARHUS UNIVERSITET

Software Engineering and Architecture

Test Stubs and Doubles

… getting the world under test control

GammaTown's RateStrategy

```
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);
```

etc. etc.

But how to test? How do I TDD it?

Read system clock to determine if weekend

AARHUS UNIVERSITET

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

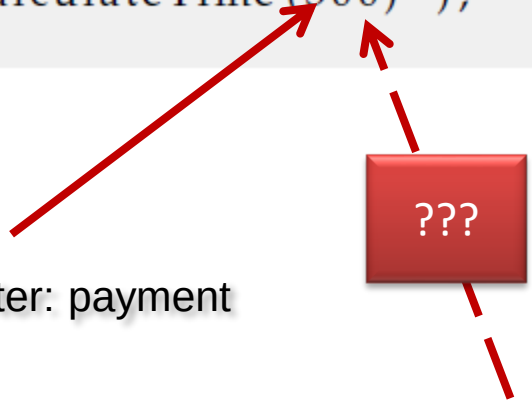
This parameter is not accessible from the testing code!

AARHUS UNIVERSITET

Code view

chapter/state/compositional/iteration-2/test/paystation/domain/TestGammaWeekdayRate.java

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



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 `test-weekday` target, a TestGammaWeekdayRate test case class that tests 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 the `test-weekend` target, I set 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 …

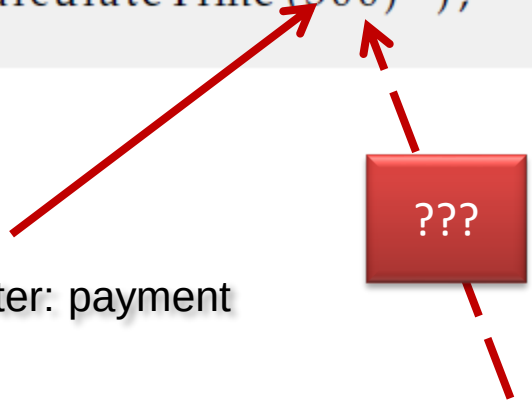
- After introducing Gammatown I no longer have automated tests because I have to run some of the tests during the weekend.
  - I have a ‘manual run on weekend and another run on weekdays targets’

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

Analysis of Parameters

chapter/state/compositional/iteration-2/test/paystation/domain/TestGammaWeekdayRate.java

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



Direct input parameter: payment

Indirect input parameter: day of week

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?

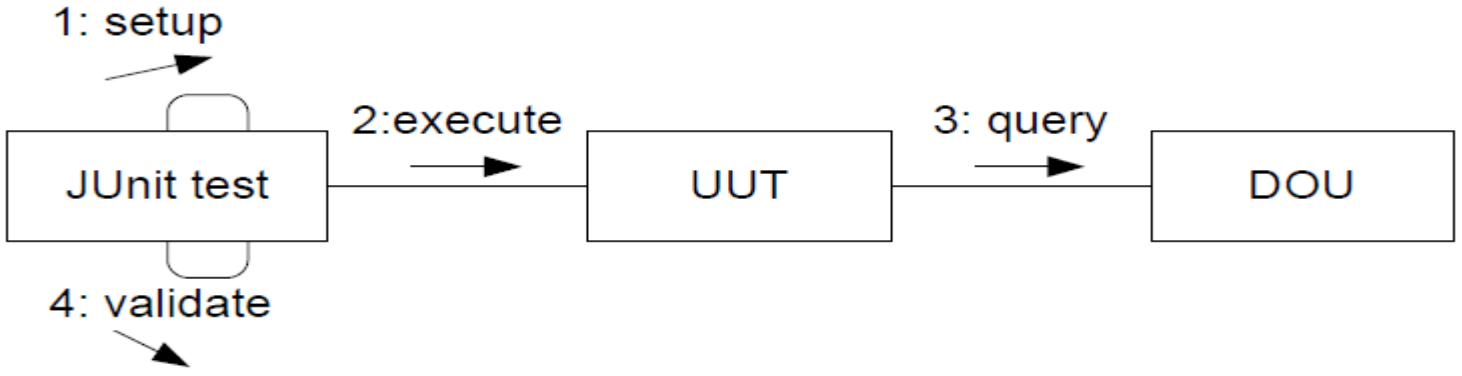
- So the 1000$ question is: where does the indirect input parameter come from?

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

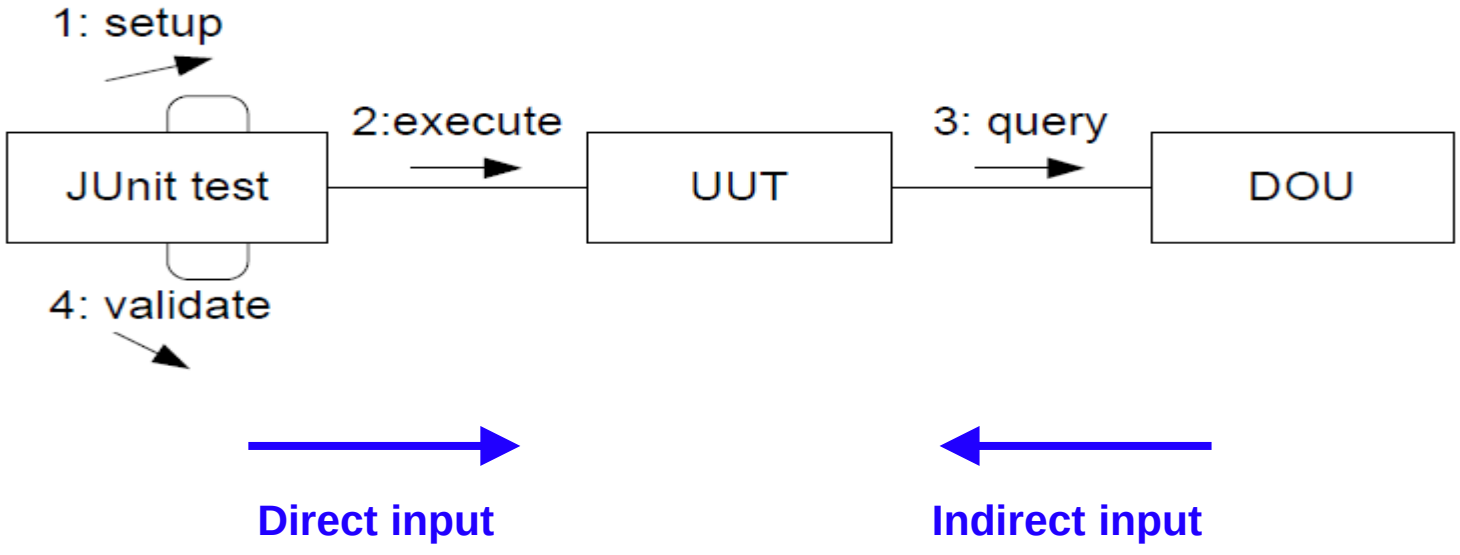
- Structure of xUnit test cases



  - Collaboration diagram: interaction between objects
- DOU = Depended On Unit

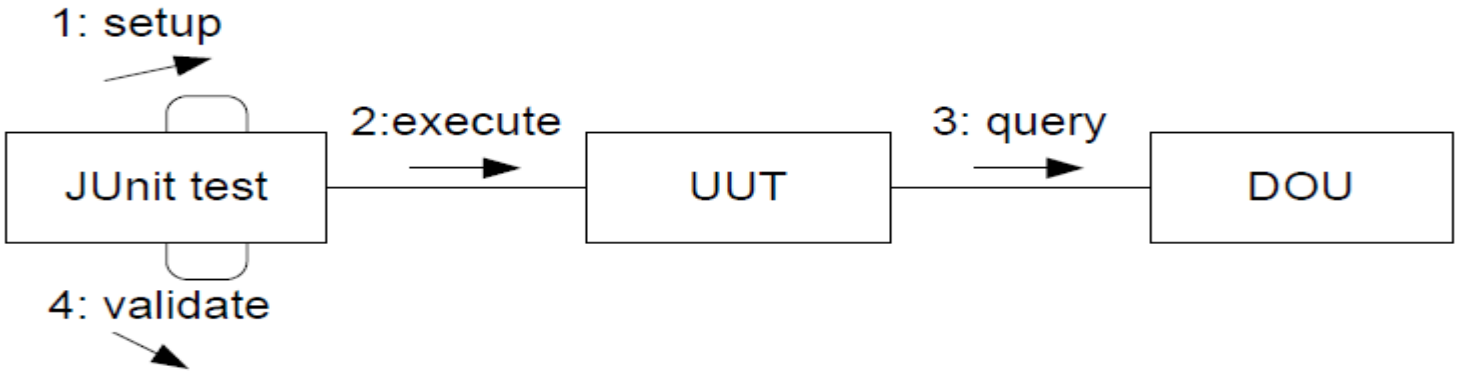
> Definition: **Depended-on unit**
> A unit in the production code that provides values or behavior that affect the behavior of the unit under test.

Direct versus Indirect



The Gammatown Rate Policy

- My DOU is the Java system clock:



Test code

AlternatingRateStrategy

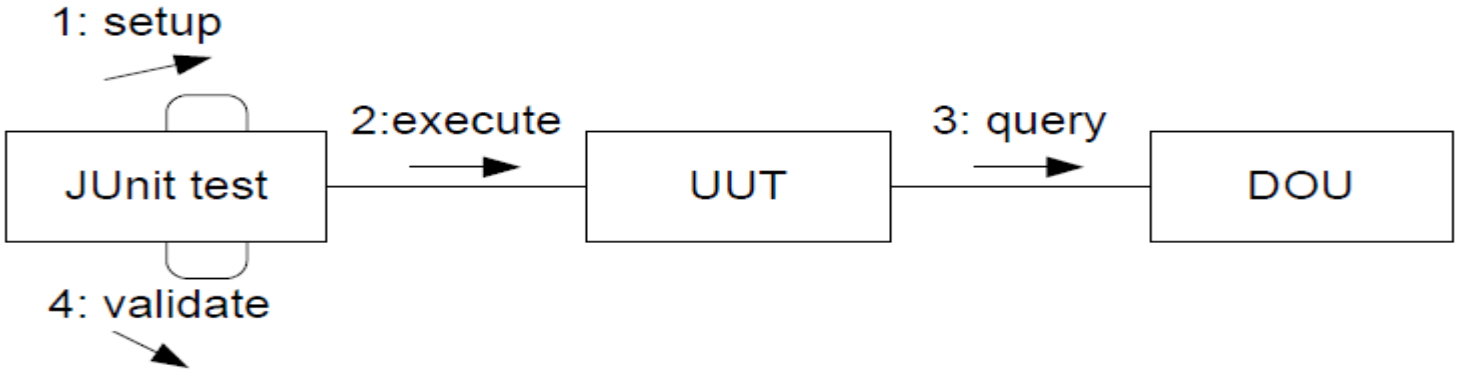
System Clock
System.DateTime
java.util.Calendar

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 ops, 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 ☺

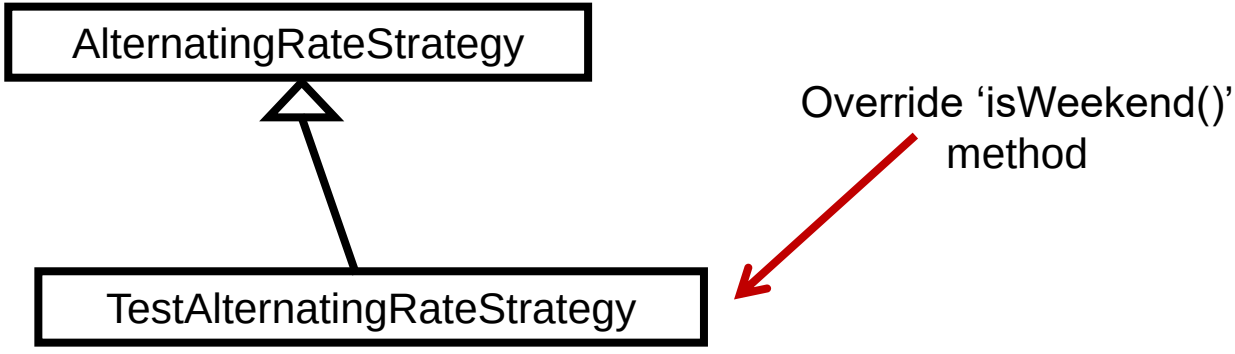
Parametric

- 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;

Polymorphic

- Subclass or die...



- Actually a quite reasonable approach...

- Argue why!!!

Compositional

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

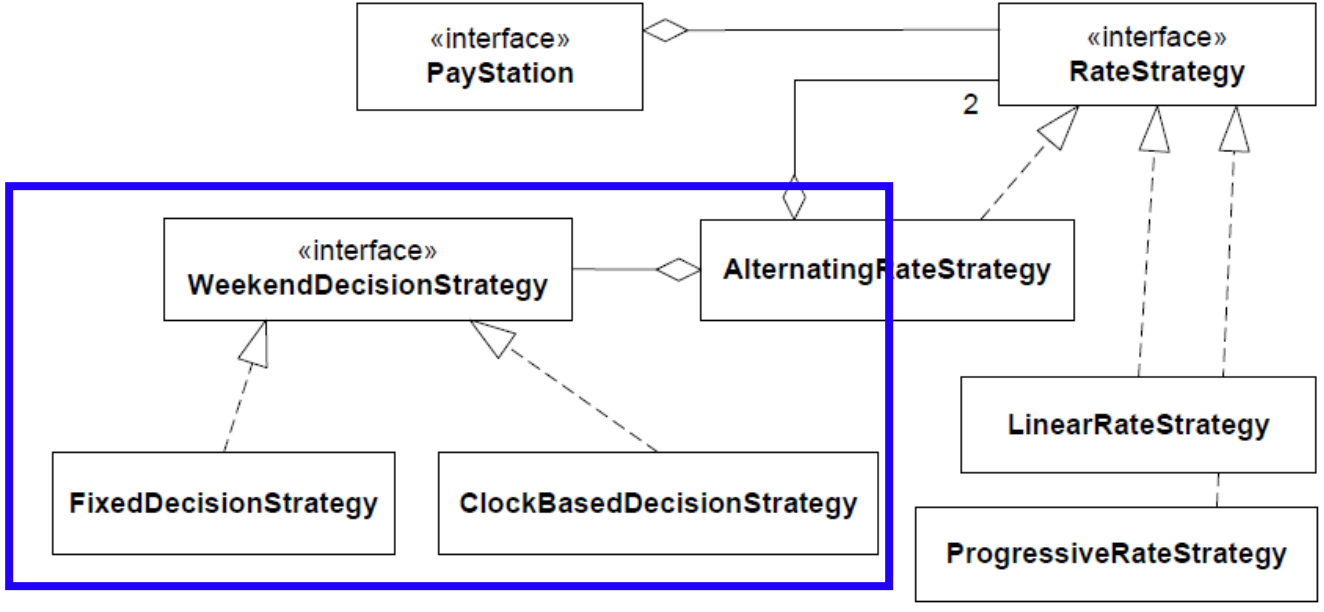
③ *I identify some behavior that varies.* It is basically the behavior defined by the isWeekend() method that is variable.

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).

AARHUS UNIVERSITET

Static Architecture View



- Exercise: Why is this Strategy and not State?

AARHUS UNIVERSITET

Production Code

```
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 Stub

Listing: chapter/test-stub/iteration-2/test/paystation/domain/FixedDecisionStrategy.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;
  }
}
```

Test Code

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

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-stub/iteration-2/test/paystation/domain/TestGammaWeekdayRate.java

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

Direct input parameter: payment

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

AARHUS UNIVERSITET

Sorry Bob ☹

- I had not read Uncle Bob when I wrote the book
  - What property is the present code missing?

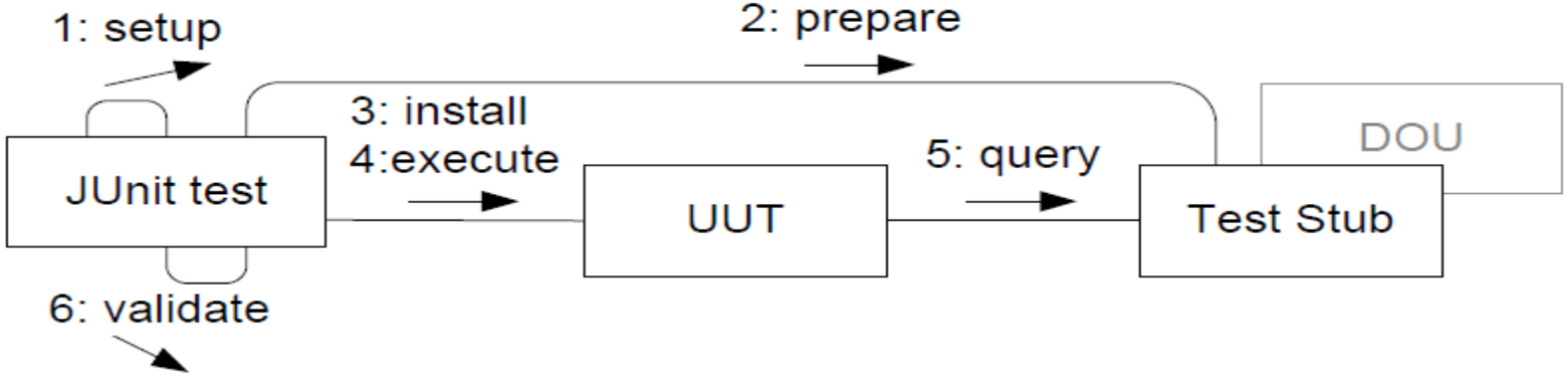
```
public class TestAlternatingRate {
  /** Test two hour parking during weekdays */
  @Test public void shouldDisplay120MinFor300centWeekday() {
    RateStrategy rs =
      new AlternatingRateStrategy( new LinearRateStrategy(),
                                   new ProgressiveRateStrategy(),
    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

Test Stub

- I have made 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*.



Key point

Key Point: Test stubs make software testable

Many software units depend on indirect input that influence their behavior. Typical indirect input are external resources like hardware sensors, random-number generators, system clocks, etc. Test stubs replace the real units and allow the testing code to control the indirect input.

Note

- Please note that once again the 3-1-2 is the underlying and powerful engine for *Test Stub*.

- 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.

Reusing the variability points...

Aah – I could do this...

AARHUS UNIVERSITET

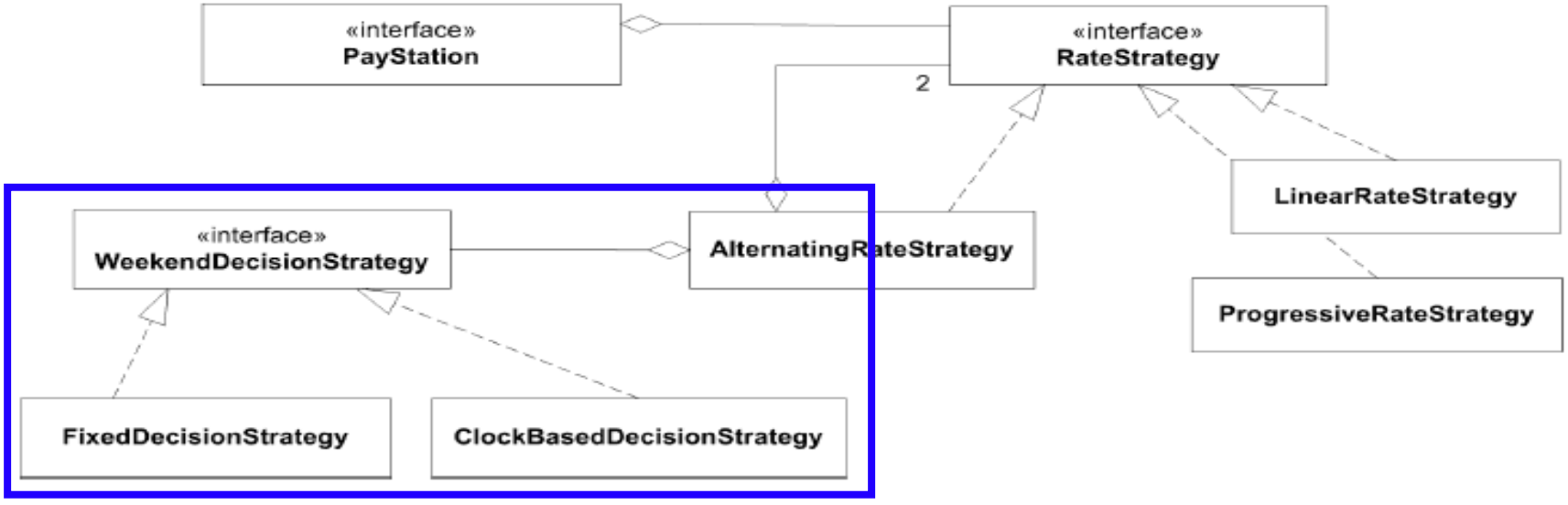
Variability points to the rescue

- 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 ☺**

AARHUS UNIVERSITET

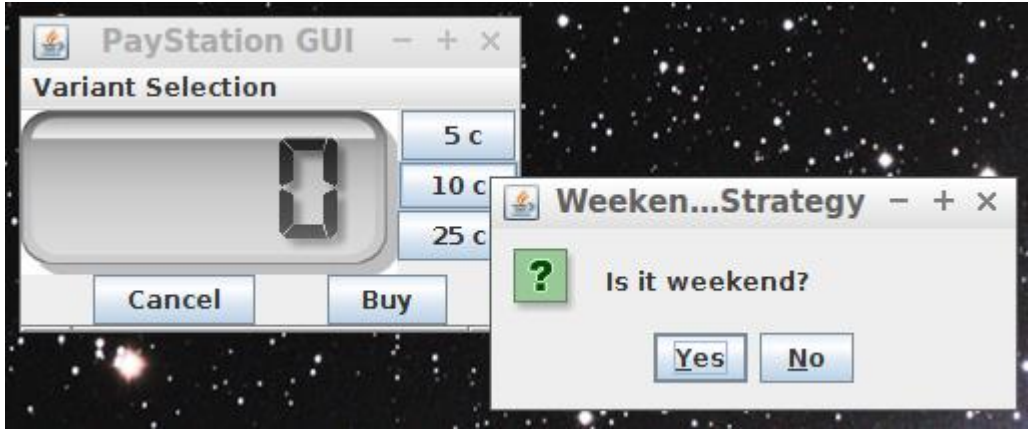
Static Architecture View



AARHUS UNIVERSITET

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 );
  }
}
```

Discussion

Test Doubles

- Test Stub is a subtype of Test Double. Other sub types exists:
  - **Stub**: Get indirect input under control
  - **Spy**: Get indirect **output** under control
    - to validate that UUT use the proper protocol
      - count method calls, ensure proper call sequence
  - **Mock**: A spy with *fail fast* property
    - Frameworks exists that test code can ‘program’ mocks without every coding them in the native language
    - Fail fast: fail on first breaking of protocol
  - **Fake**: A lightweight but realistic double
    - when the UUT-DOU interaction is slow and tedious
    - when the Double interaction is not the purpose of test

AARHUS UNIVERSITET

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?

AARHUS UNIVERSITET

C# Delegates / Java 8 Lambda

- The strategy 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 closures

Summary

Key Points

- *Test Stubs 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
  - 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…

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** ☺

AARHUS UNIVERSITET

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 ☺*