Lecture 11

ECE 1145: Software Construction and Evolution

Test Stubs
(CH 12)

Announcements

- Relevant Exercises: **12.5**
- **Code Review 1 due Oct. 10**
    - Compete code review template and report
- **Midterm Oct. 18 (take-home)**
    - Open book, open notes, work individually
    - Access and submit via Canvas
    - ~24 hour window
    - Lectures 1 – 9, project iterations 1 – 3 and code review
    - Midterm review on Wednesday Oct. 13
- Iteration 4 (due Oct. 17) will be code quality improvements

Questions for Today

How do we write tests when production code uses resources that are outside of our control?

Recall: GammaTown Alternating Rate

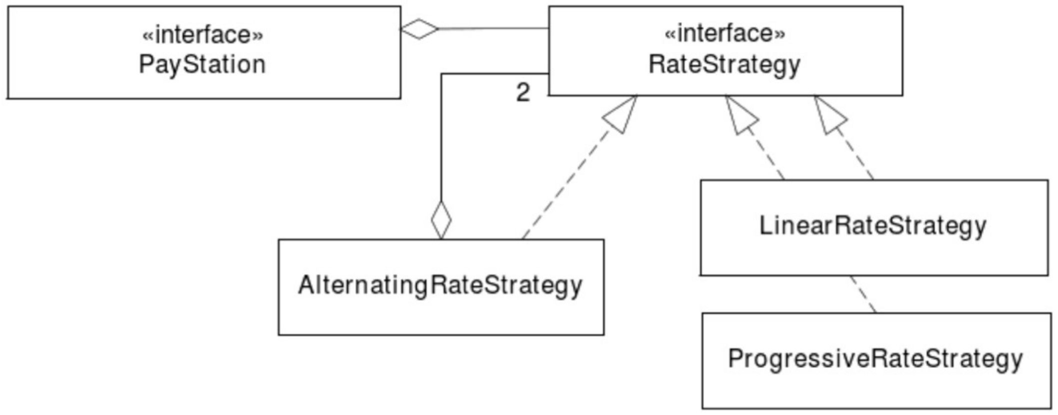
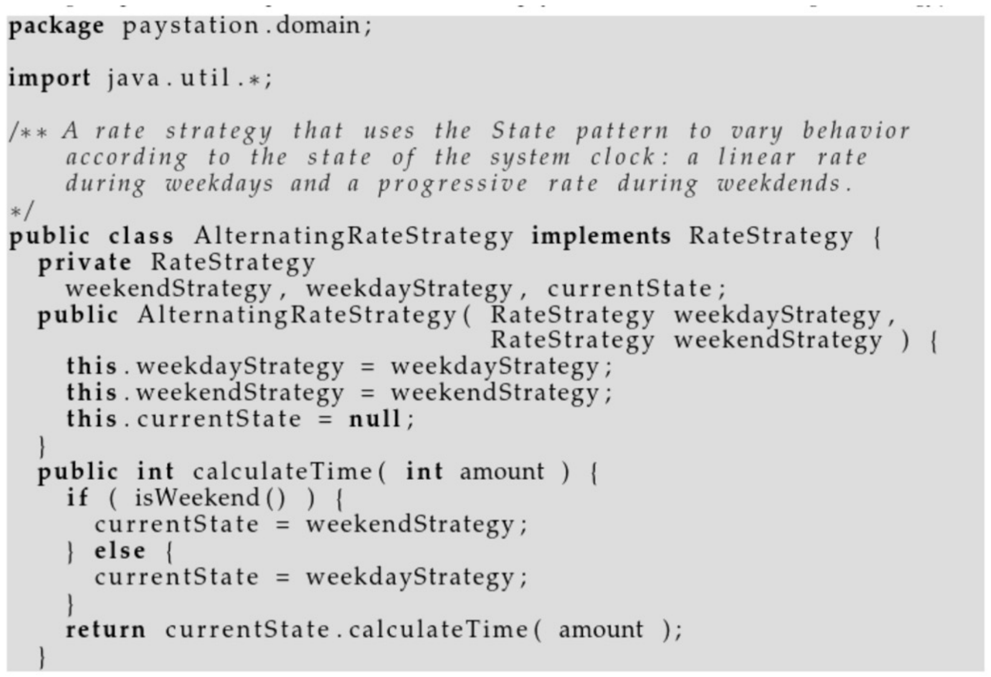
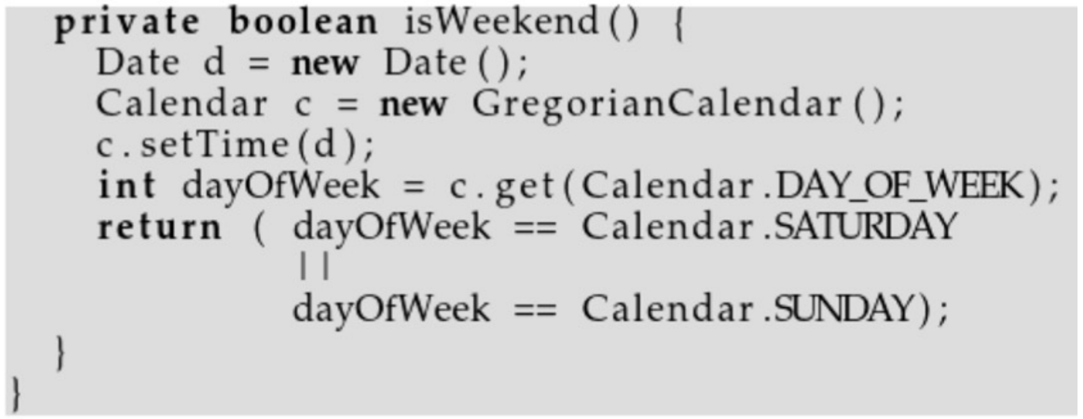


Figure 11.3: Rate calculation as a combined effort.



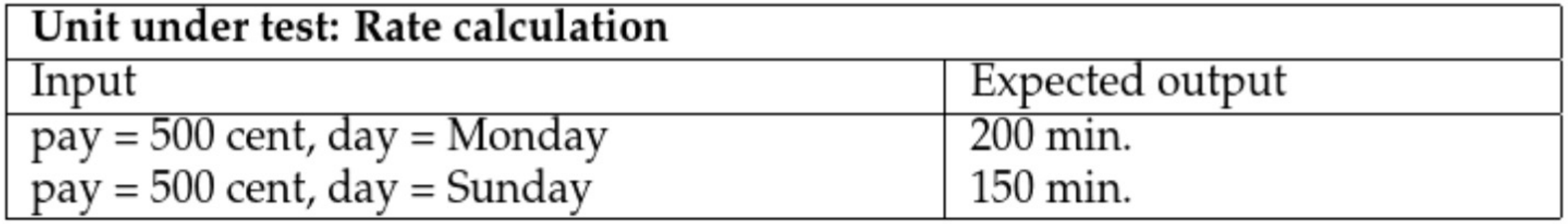
```
package paystation.domain;

import java.util.*;

/** A rate strategy that uses the State pattern to vary behavior
    according to the state of the system clock: a linear rate
    during weekdays and a progressive rate during weekdends.
*/
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);
  }
}
```

Testing the Alternating Rate

GammaTown



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

Day of the week is not a parameter in the pay station or rate calculation!

Day of the week is an **indirect input parameter**

→ How do we automate testing of this?

Direct/Indirect Inputs

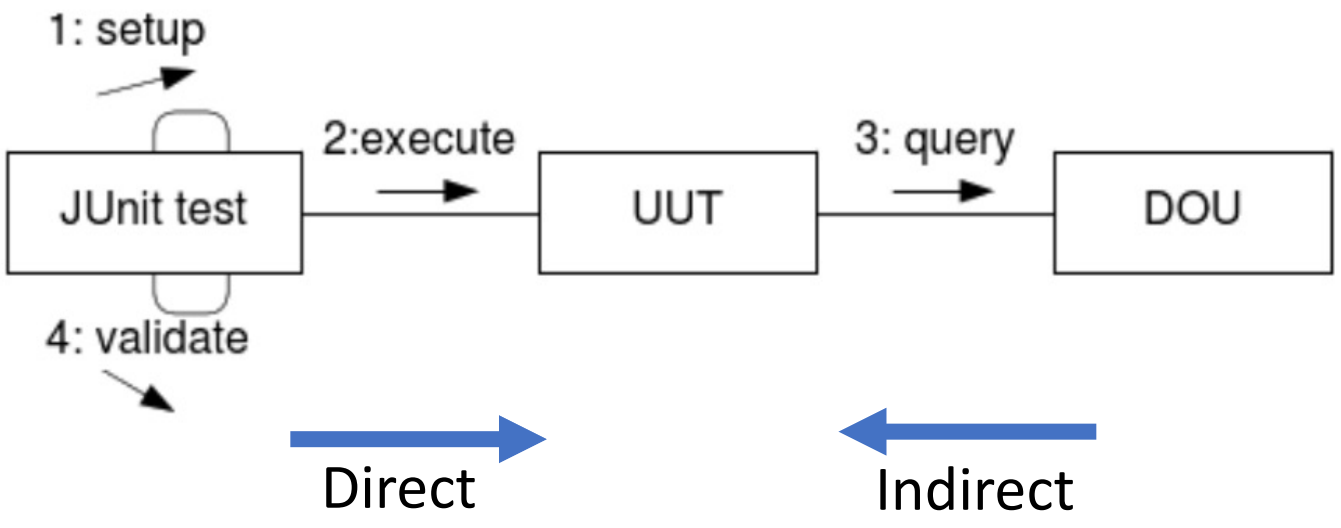
Direct input is values or data that affects the behavior of the unit under test that can be provided **directly** by the testing code

Indirect input is values or data that affects the behavior of the unit under test that **cannot** be provided directly by the testing code

A **depended-on unit** is a unit in the production code that provides values or behavior that affect the unit under test

UUT: Unit under test

DOU: Depended-on unit



Direct/Indirect Inputs

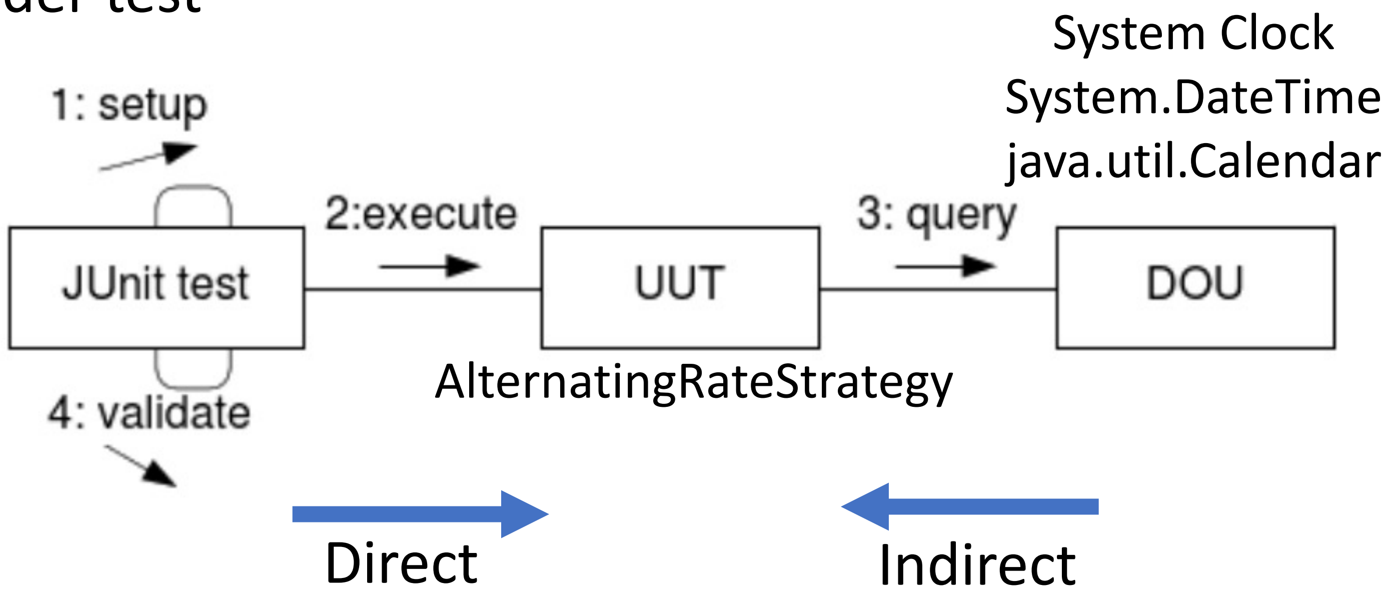
Direct input is values or data that affects the behavior of the unit under test that can be provided **directly** by the testing code

Indirect input is values or data that affects the behavior of the unit under test that **cannot** be provided directly by the testing code

A **depended-on unit** is a unit in the production code that provides values or behavior that affect the unit under test

UUT: Unit under test

DOU: Depended-on unit



Familiar Proposals

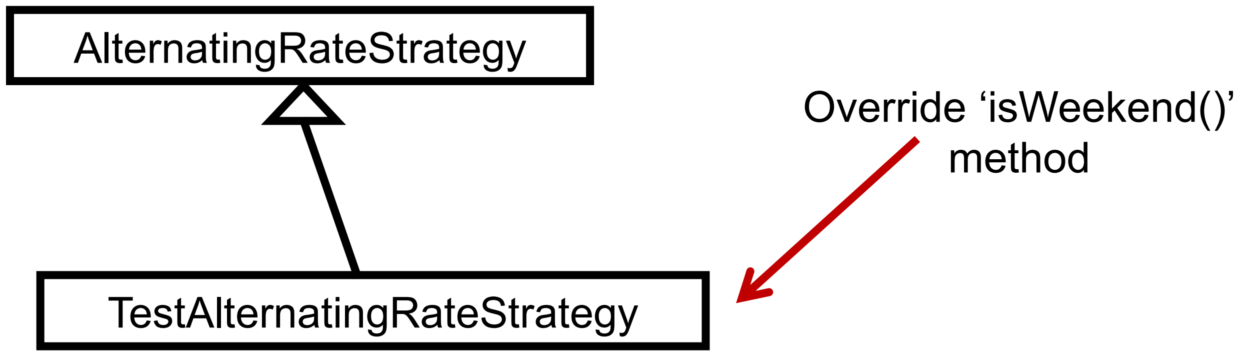
Parametric: define some Boolean parameter in the pay station that defines whether production code is in debug or normal mode. Switch on this parameter in AlternatingRateStrategy. Also need a variable to tell what day it is in debug mode, which is never used in normal operation.

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

Familiar Proposals

Parametric: define some Boolean parameter in the pay station that defines whether production code is in debug or normal mode. Switch on this parameter in AlternatingRateStrategy. Also need a variable to tell what day it is in debug mode, which is never used in normal operation.

Polymorphic: subclass AlternatingRateStrategy into TestAlternatingRateStrategy that overrides the isWeekend() method, and provide the pay station with an instance of TestAlternatingRateStrategy when testing. The subclass must be told which day to return.



Familiar Proposals

Parametric: define some Boolean parameter in the pay station that defines whether production code is in debug or normal mode. Switch on this parameter in AlternatingRateStrategy. Also need a variable to tell what day it is in debug mode, which is never used in normal operation.

Polymorphic: subclass AlternatingRateStrategy into TestAlternatingRateStrategy that overrides the isWeekend() method, and provide the pay station with an instance of TestAlternatingRateStrategy when testing. The subclass must be told which day to return.

Compositional: use the 3-1-2 process to identify, encapsulate, and delegate the behavior that is variable

Familiar Proposals

Parametric: define some Boolean parameter in the pay station that defines whether production code is in debug or normal mode. Switch on this parameter in AlternatingRateStrategy. Also need a variable to tell what day it is in debug mode, which is never used in normal operation.

Polymorphic: subclass AlternatingRateStrategy into TestAlternatingRateStrategy that overrides the isWeekend() method, and provide the pay station with an instance of TestAlternatingRateStrategy when testing. The subclass must be told which day to return.

Compositional: use the 3-1-2 process to identify, encapsulate, and delegate the behavior that is variable

Choose compositional, for similar reasons as before

3-1-2 Process

(3) Identify some behavior that varies

(1) State a responsibility that covers the behavior and express it in an interface

(2) Compose the behavior by delegating

3-1-2 Process

(3) Identify some behavior that varies

→ isWeekend()

(1) State a responsibility that covers the behavior and express it in an interface

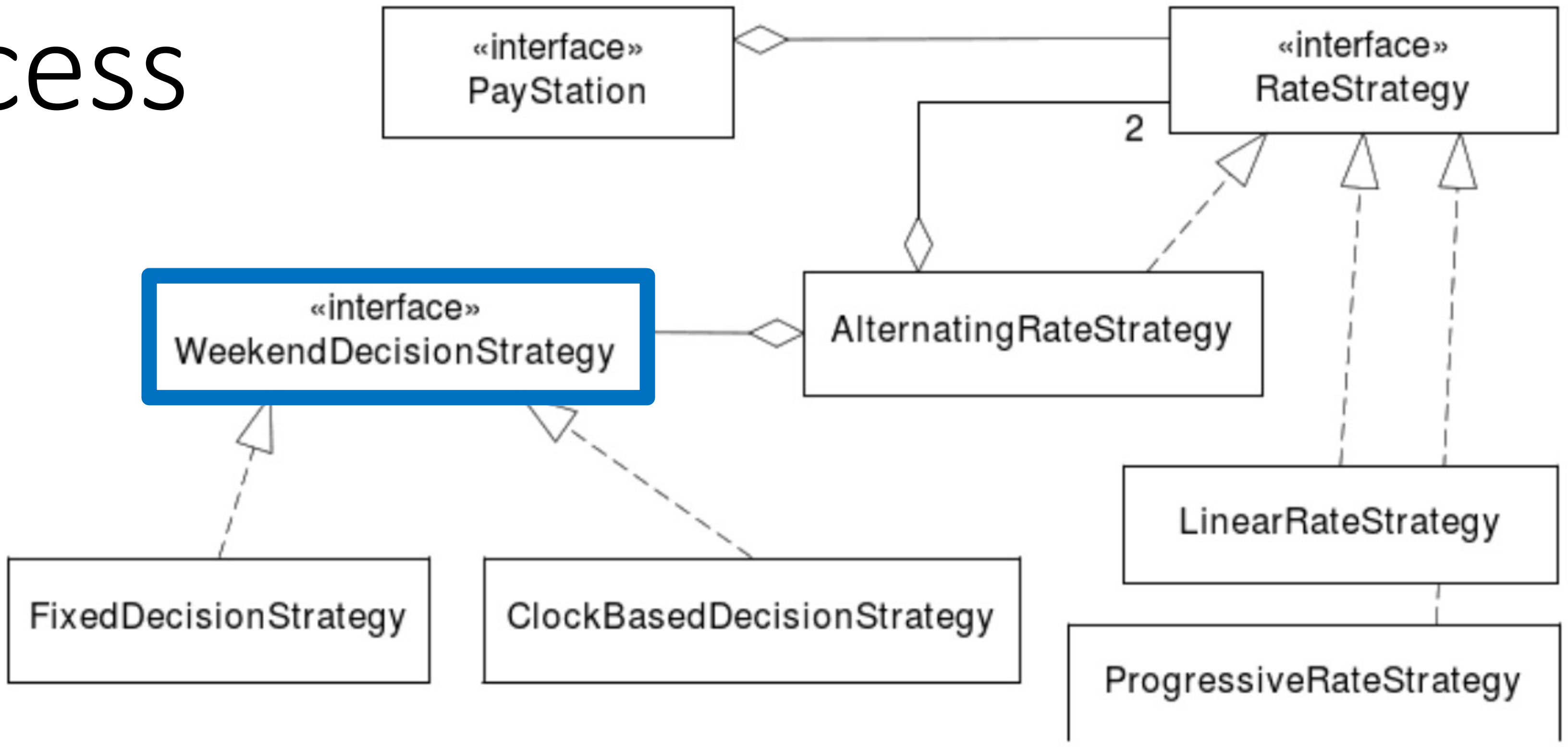
→ WeekendDecisionStrategy, contains isWeekend()

(1) Compose the behavior by delegating

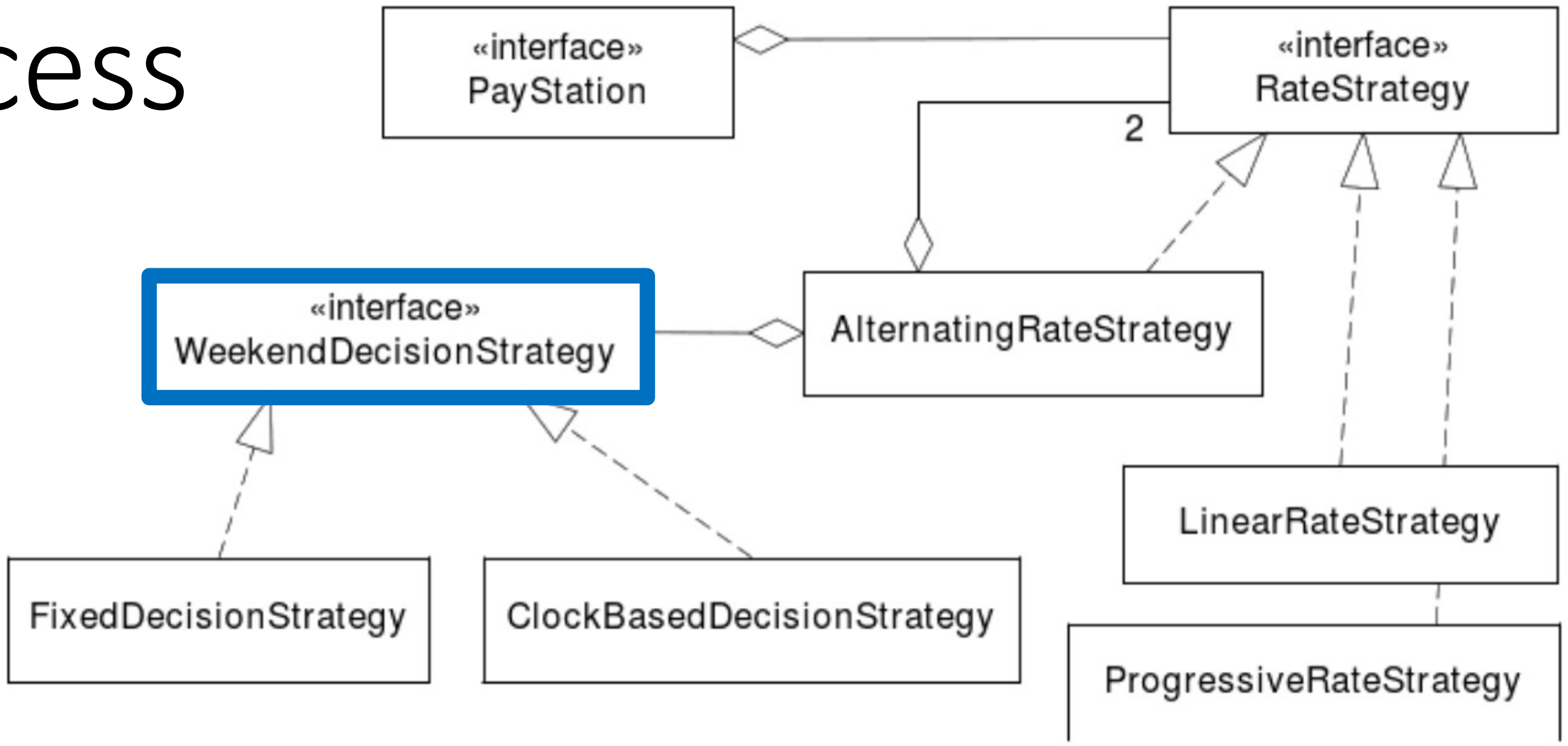
→ AlternatingRateStrategy calls isWeekend() on a concrete WeekendDecisionStrategy object

→ Create implementations that return a preset value for testing or a real value for production use

3-1-2 Process

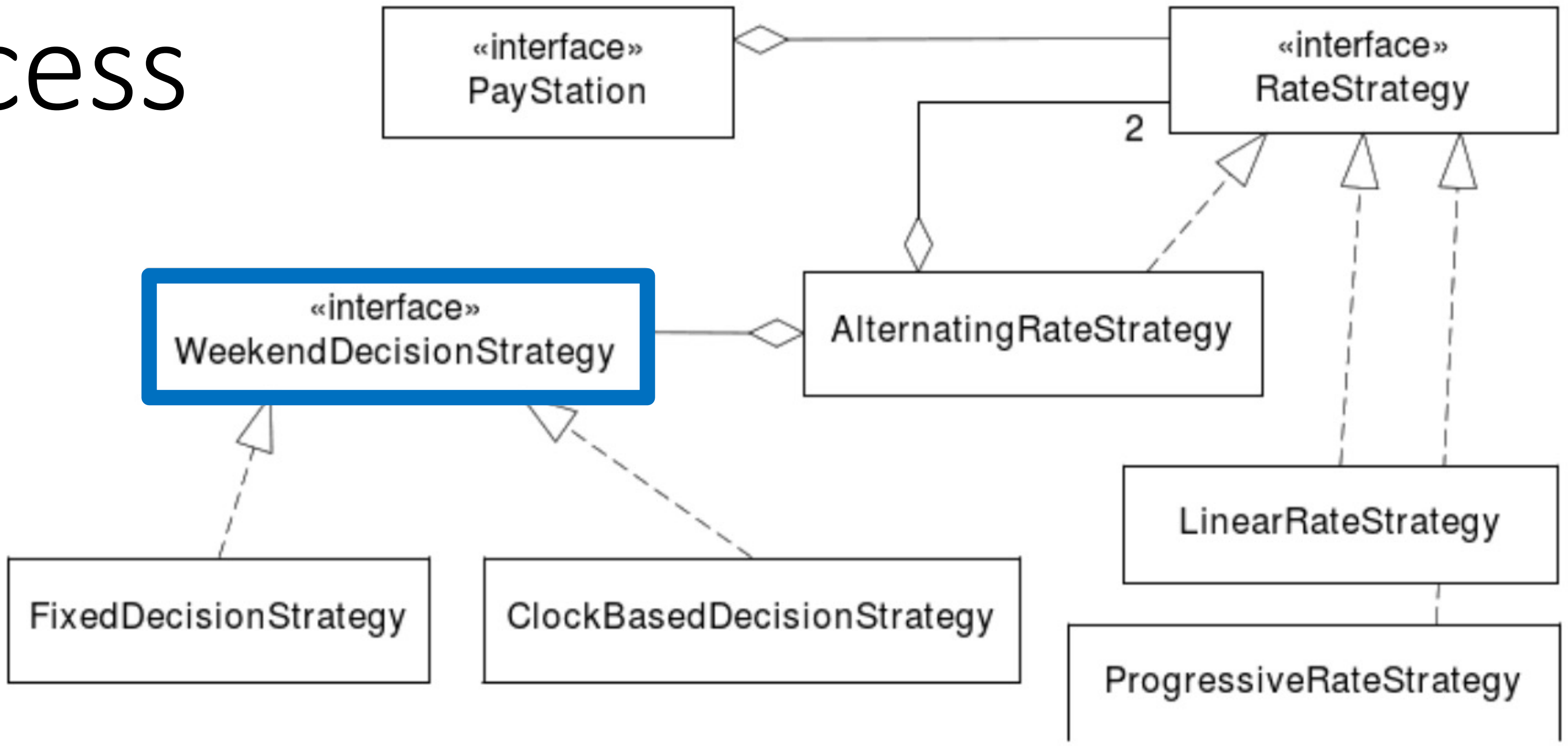

«interface»
PayStation
«interface»
RateStrategy
2
«interface»
WeekendDecisionStrategy
AlternatingRateStrategy
LinearRateStrategy
FixedDecisionStrategy
ClockBasedDecisionStrategy
ProgressiveRateStrategy

3-1-2 Process



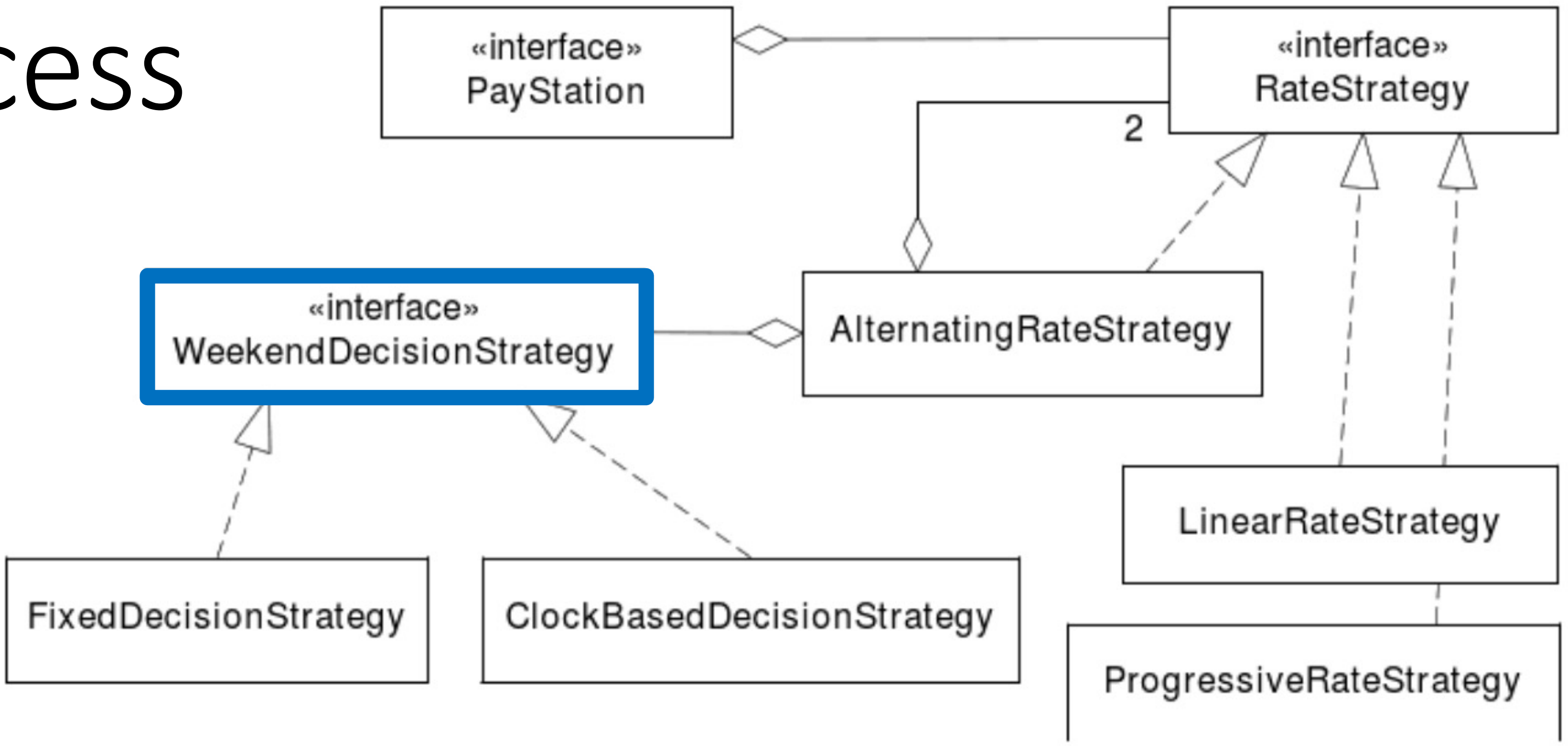
What pattern is this?

3-1-2 Process



What pattern is this? Strategy!
→ Why not state?

3-1-2 Process

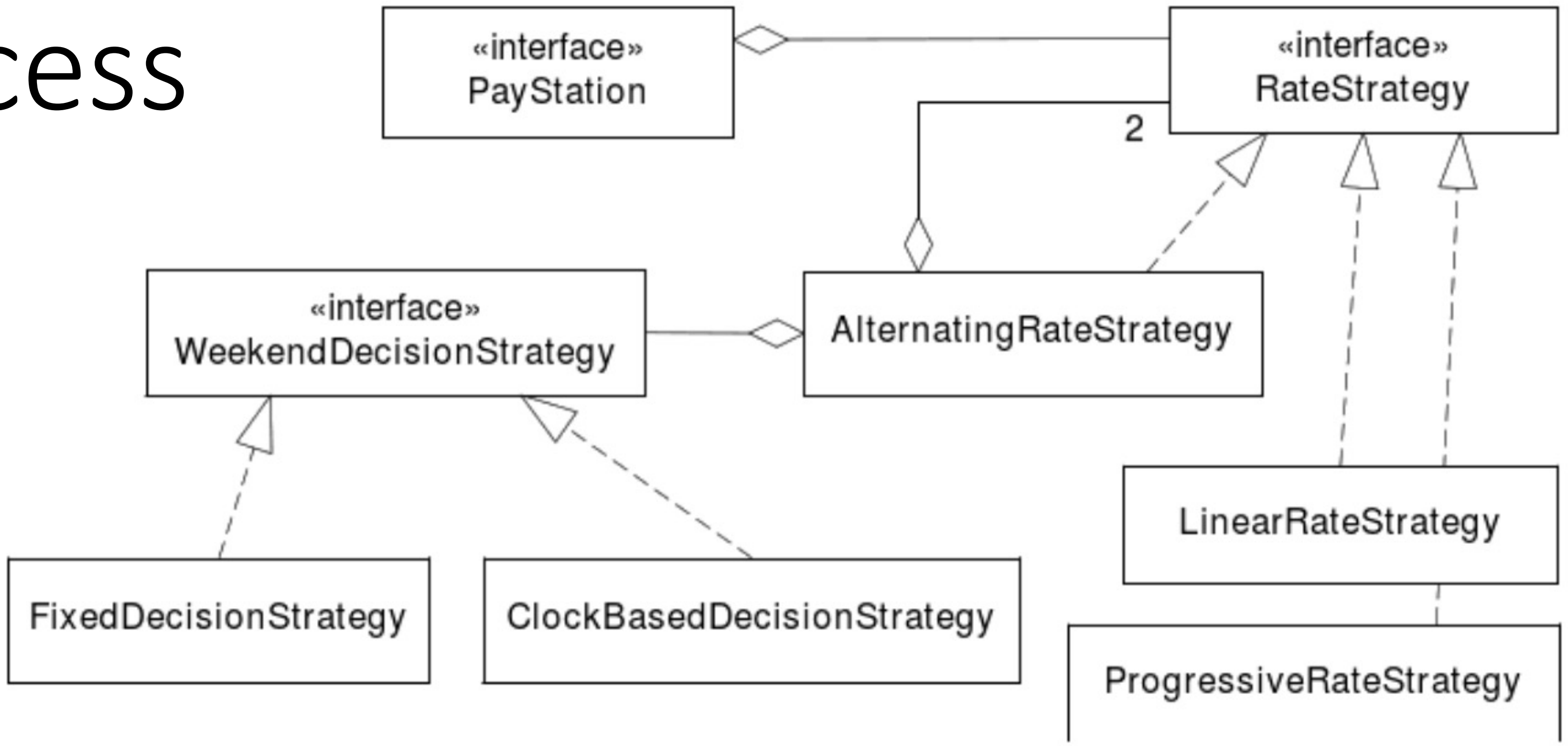


What pattern is this? Strategy!

→ Why not state?

We are **choosing an algorithm** to check if it is the weekend

3-1-2 Process



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

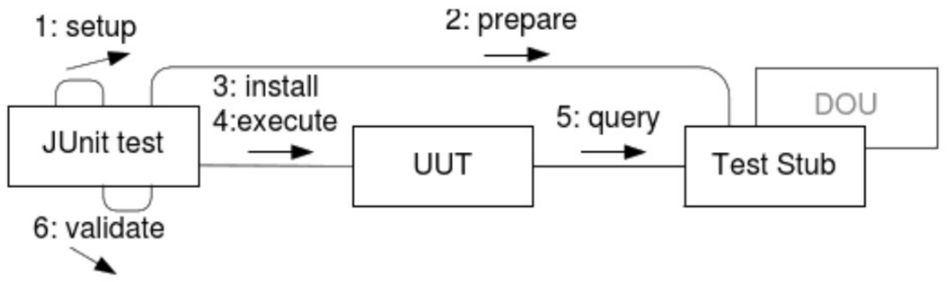
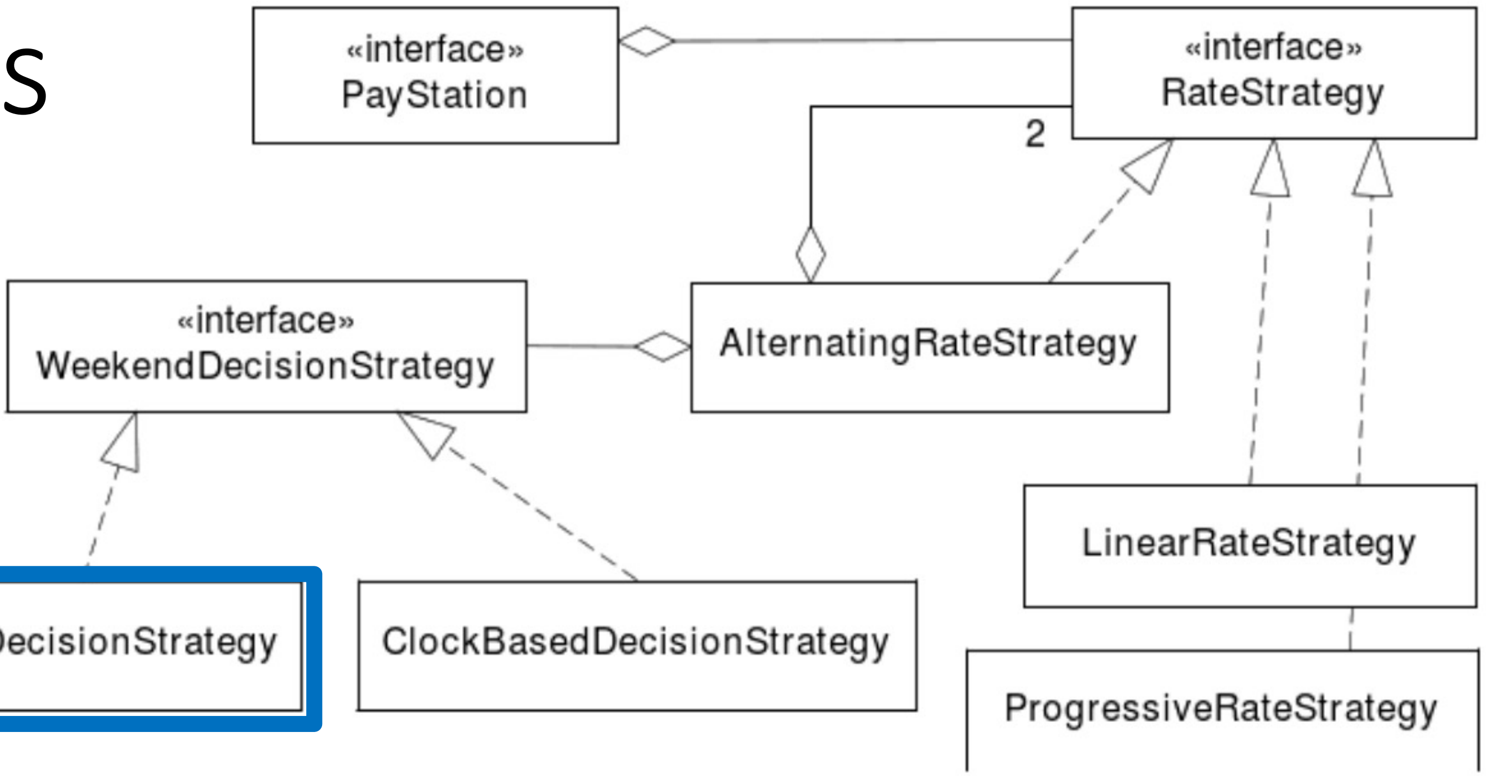


Figure 12.3: Test Stub replacing the DOU.

3-1-2 Process



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

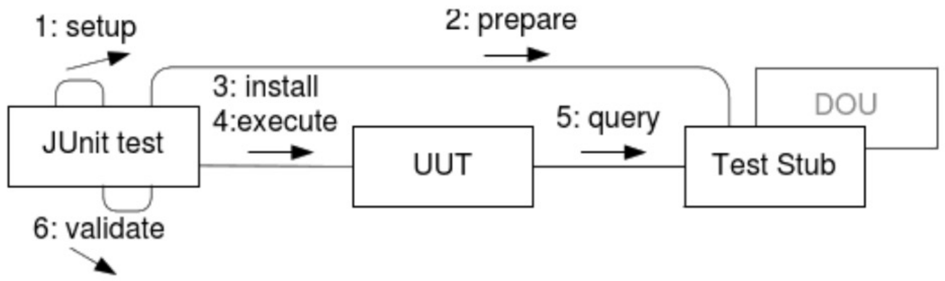
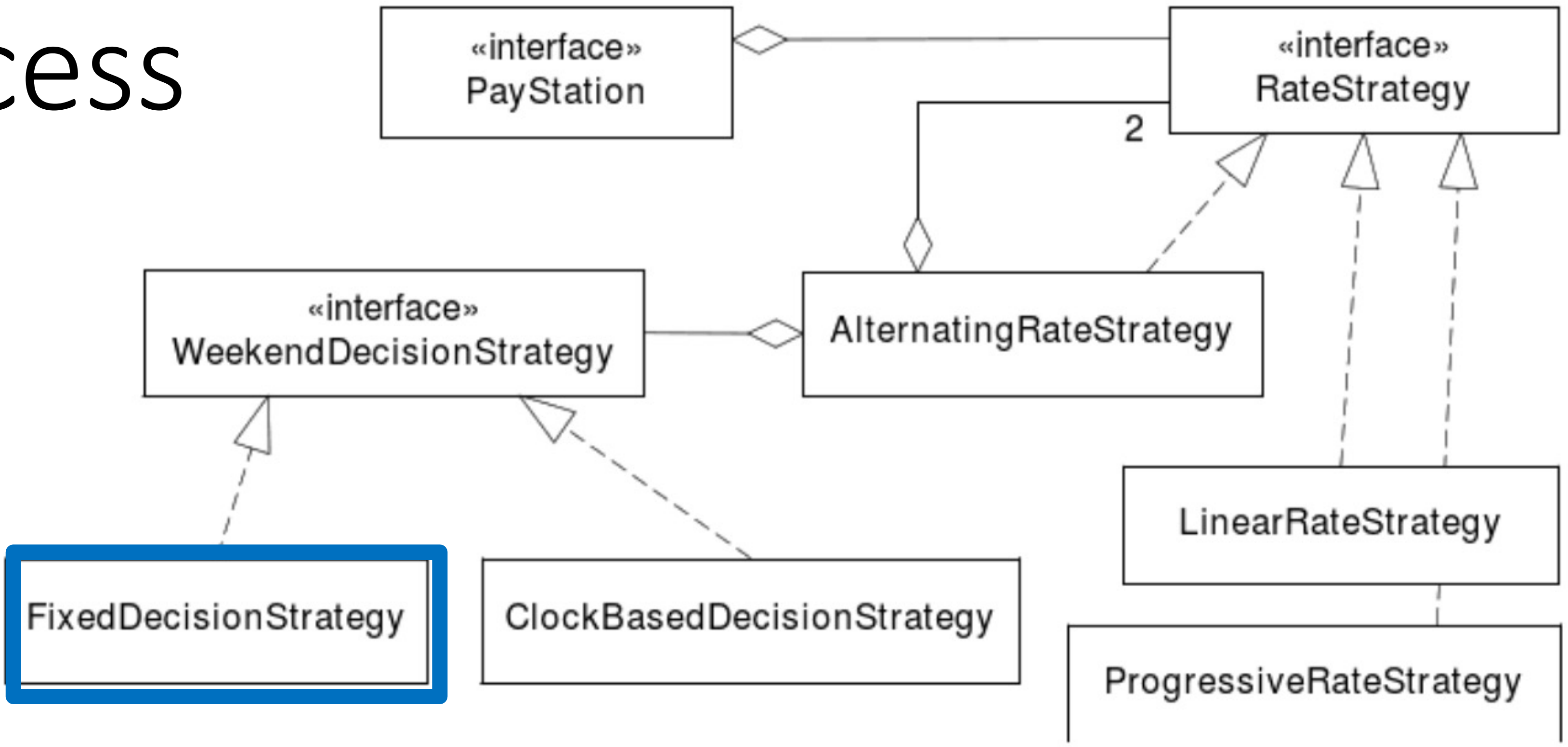


Figure 12.3: Test Stub replacing the DOU.

3-1-2 Process



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

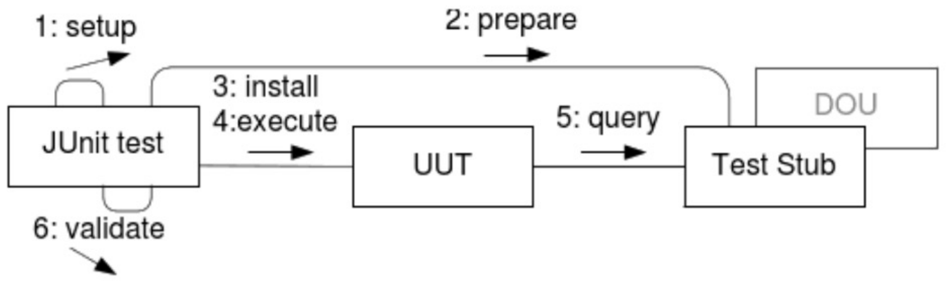
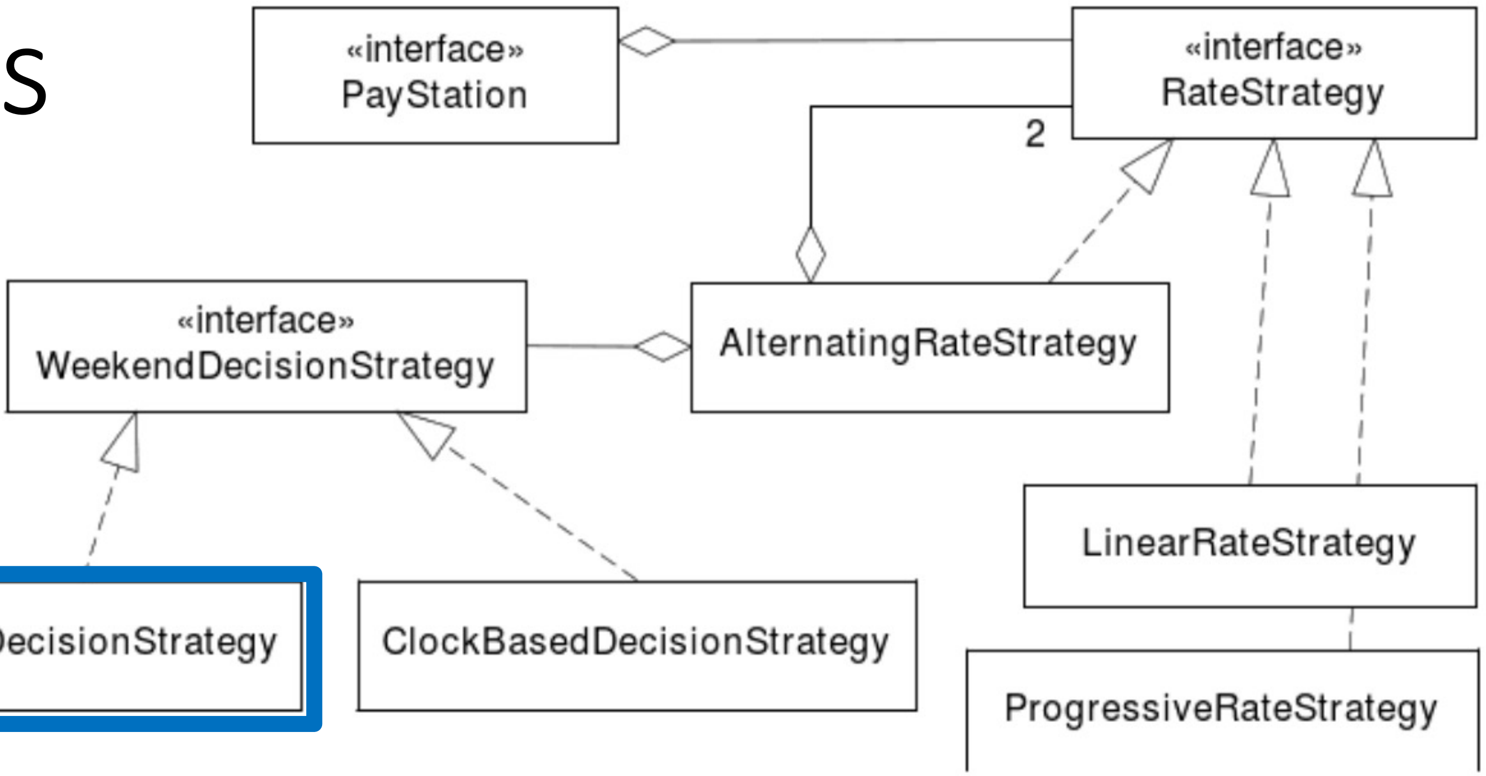


Figure 12.3: Test Stub replacing the DOU.

→ Usually created by the test unit to set and pass values to the UUT (Recall One2OneRateStrategy)

3-1-2 Process



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

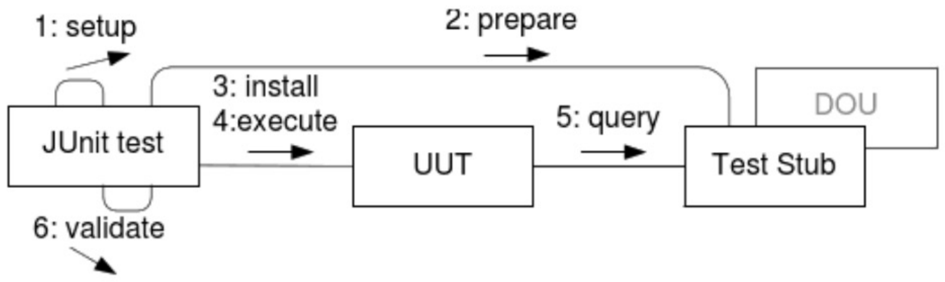


Figure 12.3: Test Stub replacing the DOU.

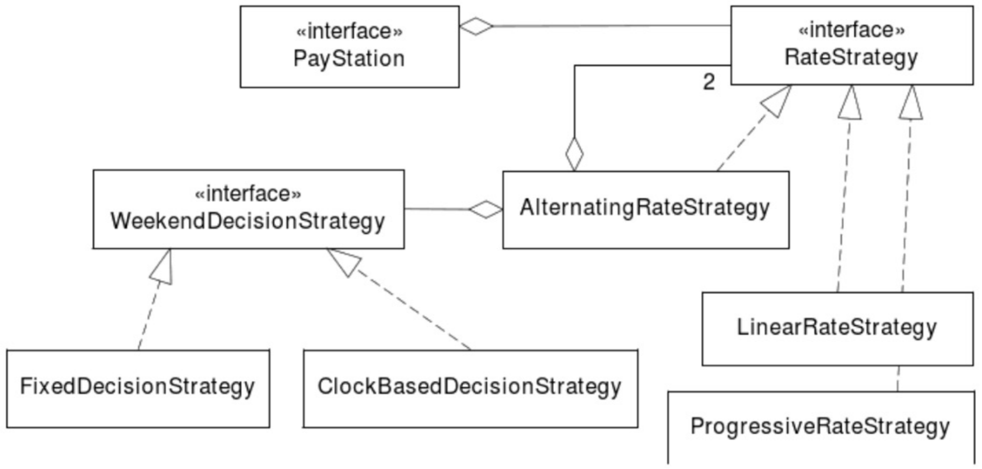
→ Usually created by the test unit to set and pass values to the UUT (Recall One2OneRateStrategy)

→ Not possible if UUT is tightly coupled to DOU (e.g., UUT creates DOU)

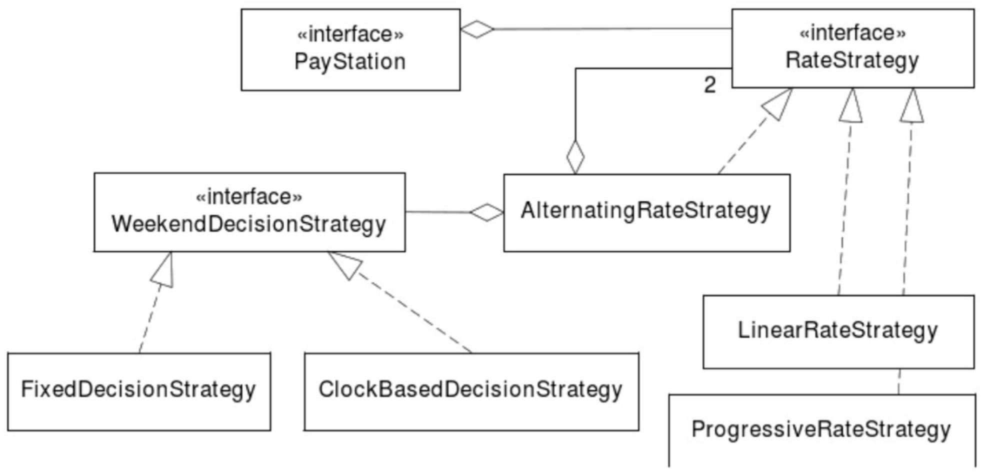
→ "Inject your dependencies"!

Implementation

How do we implement the test stub?



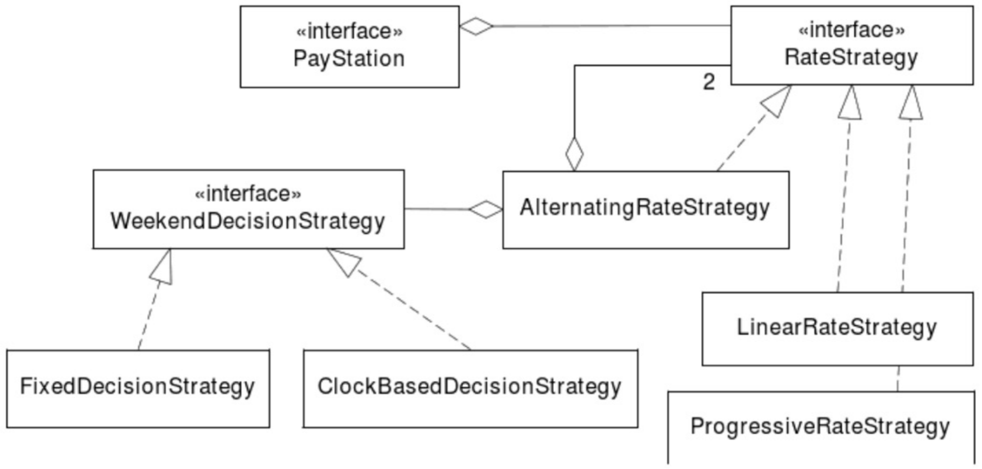
Implementation



How do we implement the test stub?

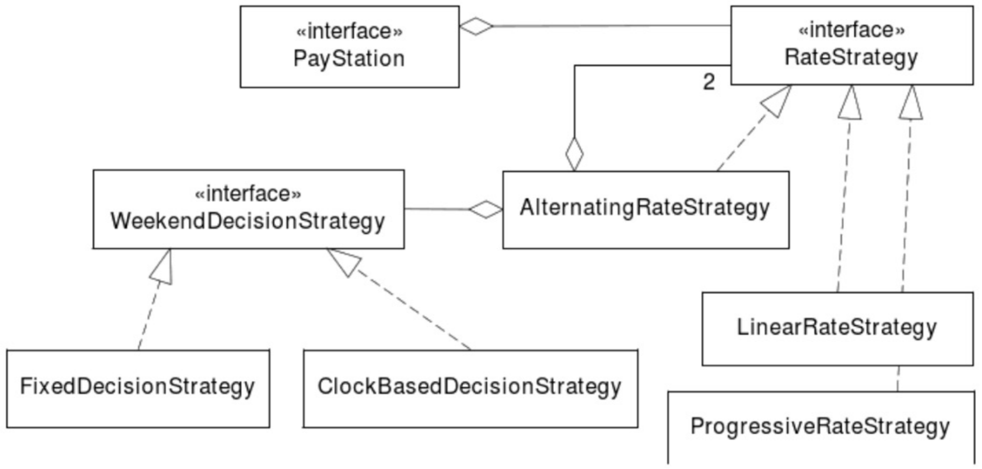
1. Refactor (similar to RateStrategy)
2. Introduce test stub FixedDecisionStrategy (in **test code**)
3. Refactor tests
4. Integration testing

Implementation



1. Refactor (similar to RateStrategy)
   - Introduce the WeekendDecisionStrategy interface
   - Refactor AlternatingRateStrategy to take instances of WeekendDecisionStrategy as a parameter in the constructor
   - See it compile but tests fail
   - Introduce ClockBasedDecisionStrategy and refactor to make all test cases pass again

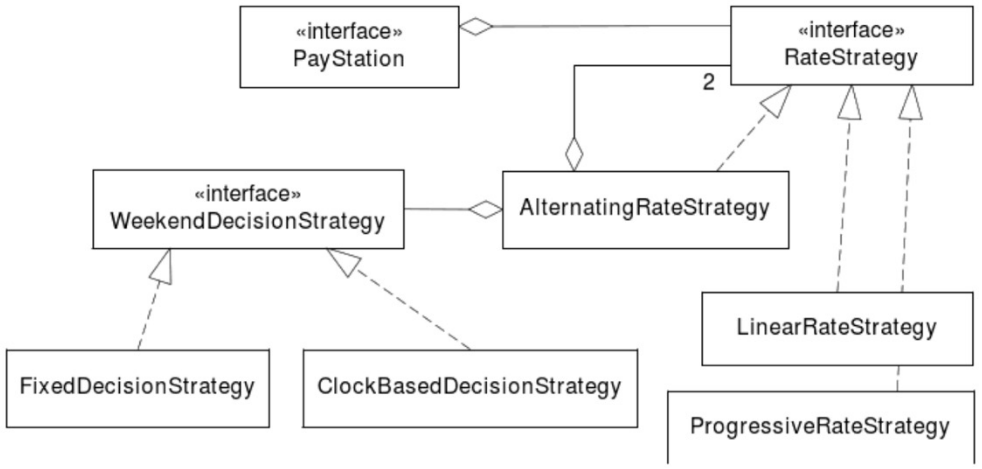
Implementation



1. Refactor (similar to RateStrategy)
2. Introduce test stub FixedDecisionStrategy (in **test code**)

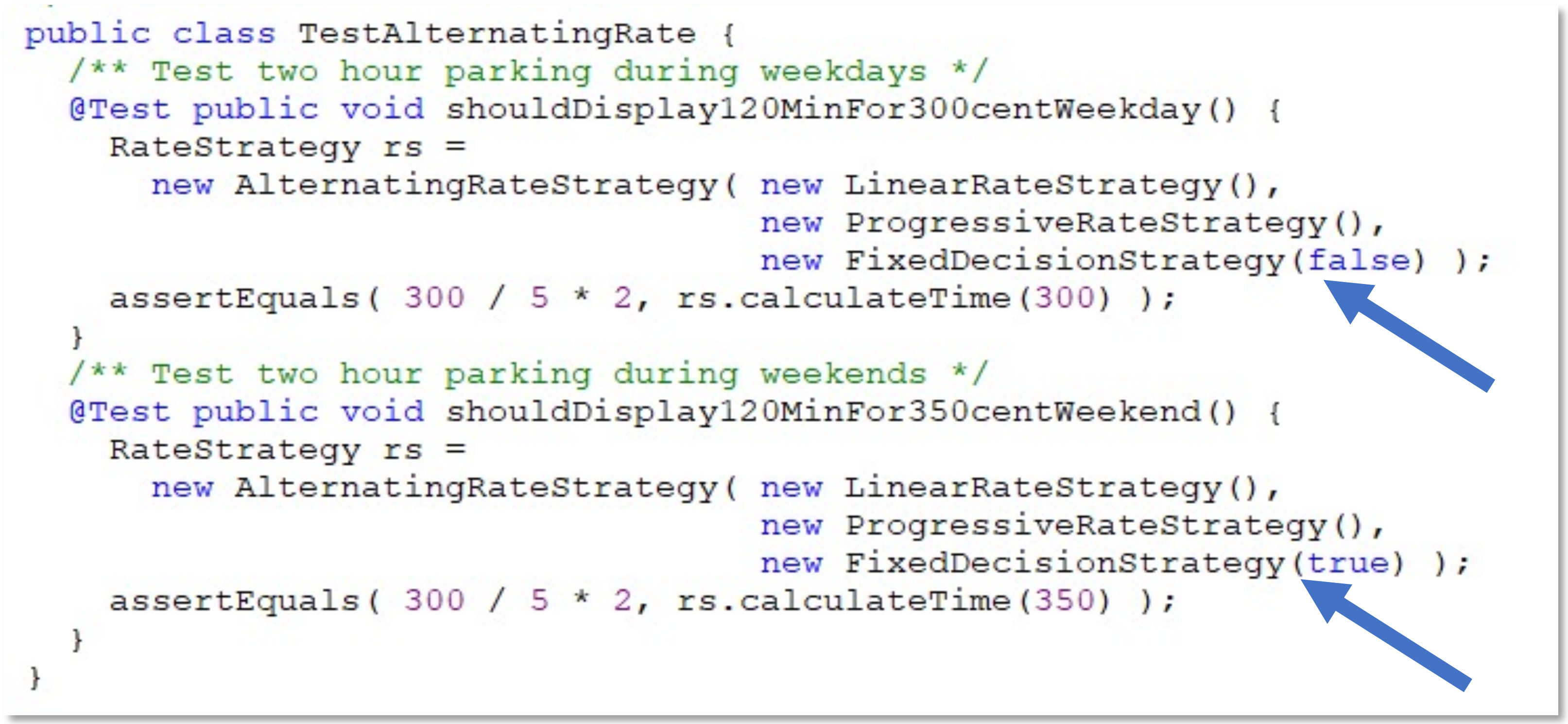
```
package paystation.domain;

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;
  }
}
```

Implementation



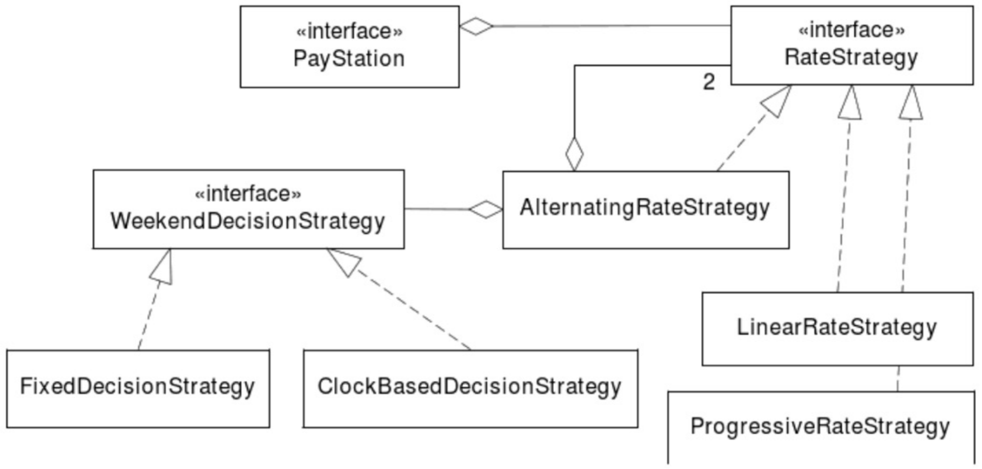
1. Refactor (similar to RateStrategy)
2. Introduce test stub FixedDecisionStrategy (in **test code**)
3. Refactor tests

TestAlternatingRate



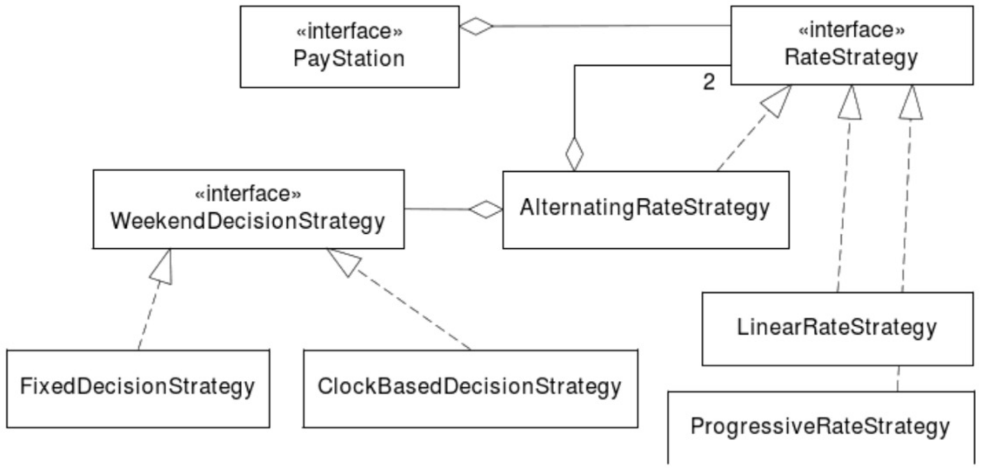
```
public class TestAlternatingRate {
  /** Test two hour parking during weekdays */
  @Test public void shouldDisplay120MinFor300centWeekday() {
    RateStrategy rs =
      new AlternatingRateStrategy( new LinearRateStrategy(),
                                   new ProgressiveRateStrategy(),
                                   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) );
  }
}
```

Implementation



1. Refactor (similar to RateStrategy)
2. Introduce test stub FixedDecisionStrategy (in **test code**)
3. Refactor tests
4. Integration testing
   - Already covered by tests for AlphaTown and BetaTown – they verify that the pay station and rate strategy objects interact properly

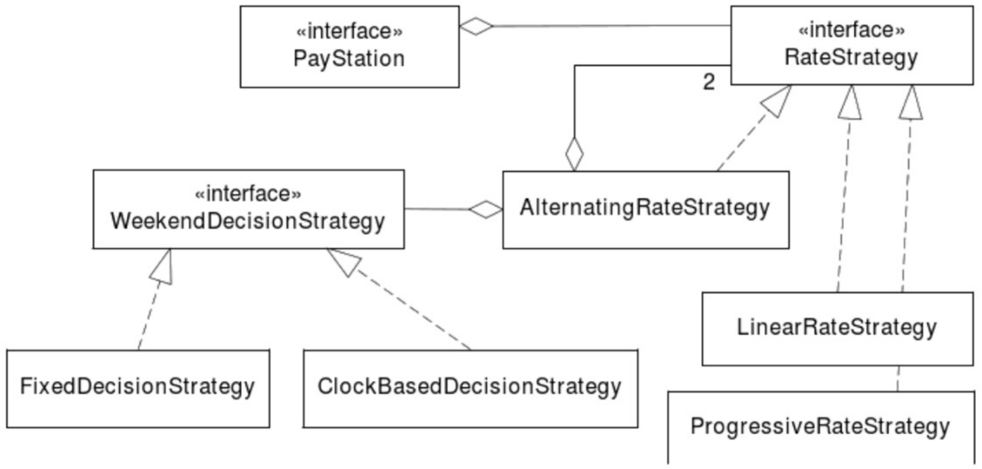
Implementation



1. Refactor (similar to RateStrategy)
2. Introduce test stub FixedDecisionStrategy (in **test code**)
3. Refactor tests
4. Integration testing
   - Already covered by tests for AlphaTown and BetaTown – they verify that the pay station and rate strategy objects interact properly

✓ Fully automated testing using test stubs!

Implementation

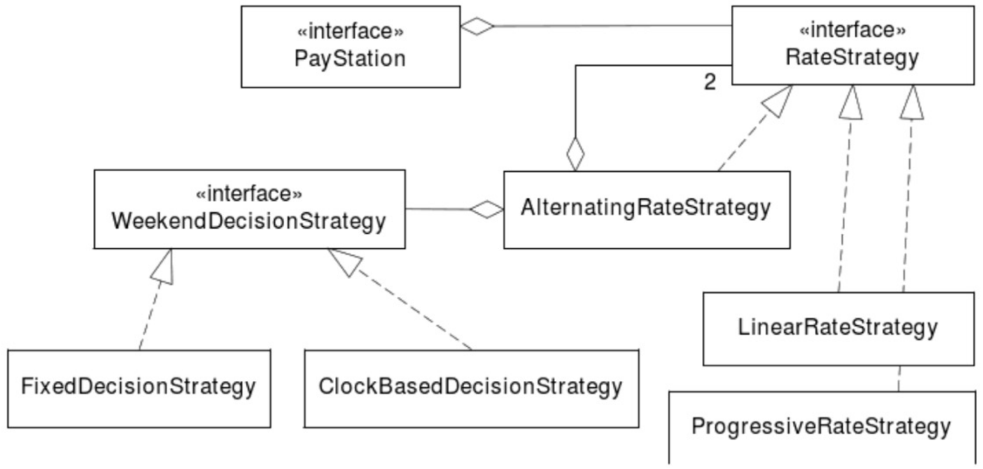


1. Refactor (similar to RateStrategy)
2. Introduce test stub FixedDecisionStrategy (in **test code**)
3. Refactor tests
4. Integration testing
   - Already covered by tests for AlphaTown and BetaTown – they verify that the pay station and rate strategy objects interact properly

✓ Fully automated testing using test stubs!

Weaknesses?

Implementation



1. Refactor (similar to RateStrategy)
2. Introduce test stub FixedDecisionStrategy (in **test code**)
3. Refactor tests
4. Integration testing
   - Already covered by tests for AlphaTown and BetaTown – they verify that the pay station and rate strategy objects interact properly

✓ Fully automated testing using test stubs!

Weaknesses?

- ClockBasedDecisionStrategy still requires manual testing

But, code is simpler! Only isWeekend() needs to be tested

Dependency Injection

This technique is enabled by **compositional design** and proper **encapsulation** of behavior that provides the indirect input

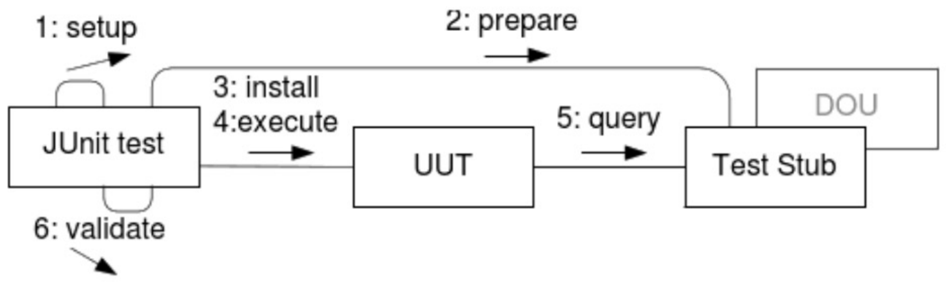


Figure 12.3: Test Stub replacing the DOU.

Dependency Injection

This technique is enabled by **compositional design** and proper **encapsulation** of behavior that provides the indirect input

- Only possible if depended-on unit is not created by or otherwise tightly coupled to the UUT
- “Inject your dependencies” - pass values to the UUT rather than having the UUT create them, to enable testing

Becomes increasingly important with greater complexity

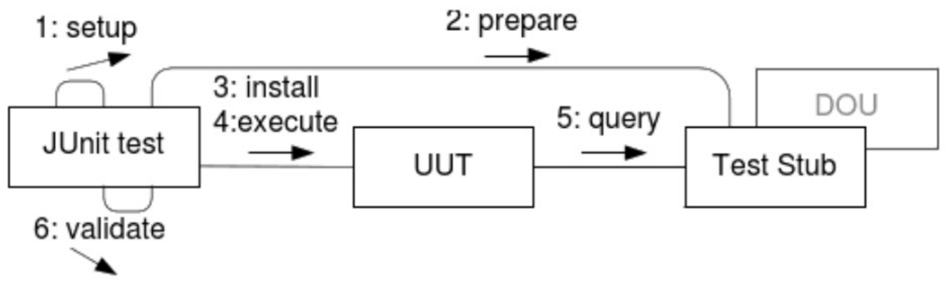


Figure 12.3: Test Stub replacing the DOU.

Test Doubles

A Test Stub is a subtype of **Test Double**

Types of Test Doubles:

- **Test Stub:** a double that feeds **indirect input** (defined by the test case) into the UUT
- **Test Spy:** a double that records the UUT's indirect output for later verification by the test case
- **Fake:** a double that acts as a high-performance replacement for a slow or expensive DOU
- **Mock object:** a double created and programmed dynamically by a mock library that may serve as both a stub and a spy

Test Doubles

A Test Stub is a subtype of **Test Double**

Types of Test Doubles:

- **Test Stub:** a double that feeds **indirect input** (defined by the test case) into the UUT
- **Test Spy:** a double that records the UUT’s indirect output for later verification by the test case
- **Fake:** a double that acts as a high-performance replacement for a slow or expensive DOU
- **Mock object:** a double created and programmed dynamically by a mock library that may serve as both a stub and a spy

Mockito testing framework (Java): https://site.mockito.org/

Test Doubles

Other test stub/double applications?

- External sensors
- Random numbers

Test Doubles

Other test stub/double applications?

- External sensors
- Random numbers

Test doubles make software **testable** by replacing real units and allowing test code to control **indirect input**, detect **indirect output**, or act as a **mimic** of a slow/expensive external resource

Next time: Another pattern and pattern fragility