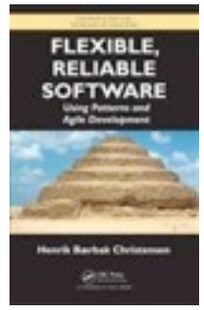
# Theme 2
# Program Design

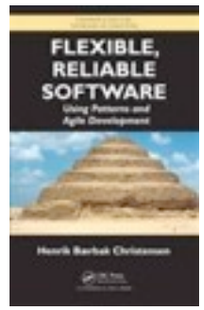
## Deriving State Pattern
## Test Stubs



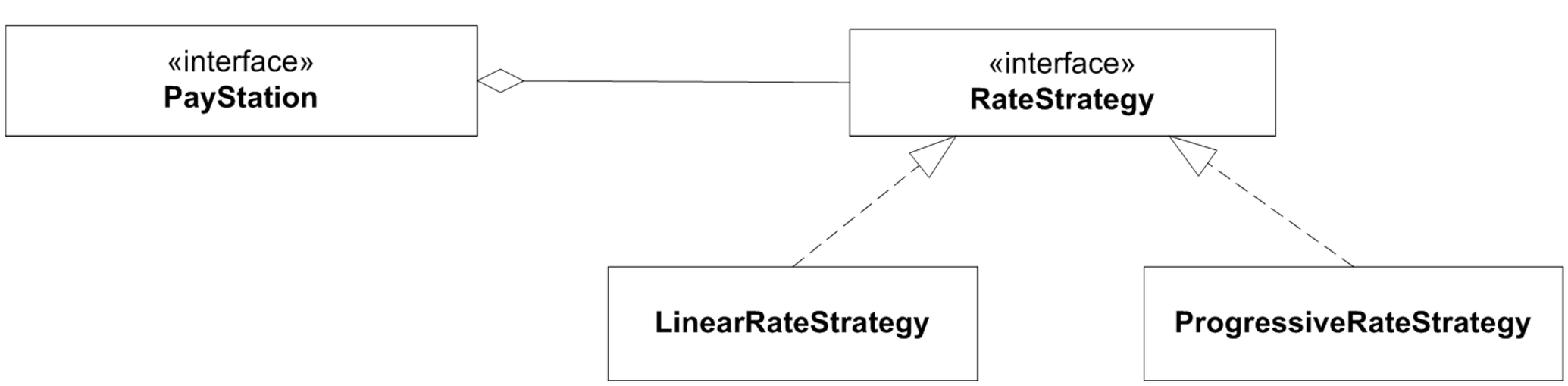

# Learning Objectives

- to analyze how the polymorphic and compositional proposals cope when faced with a requirement that combines existing solutions.
- to demonstrate how the compositional proposal leads to the STATE pattern.

# New Requirement



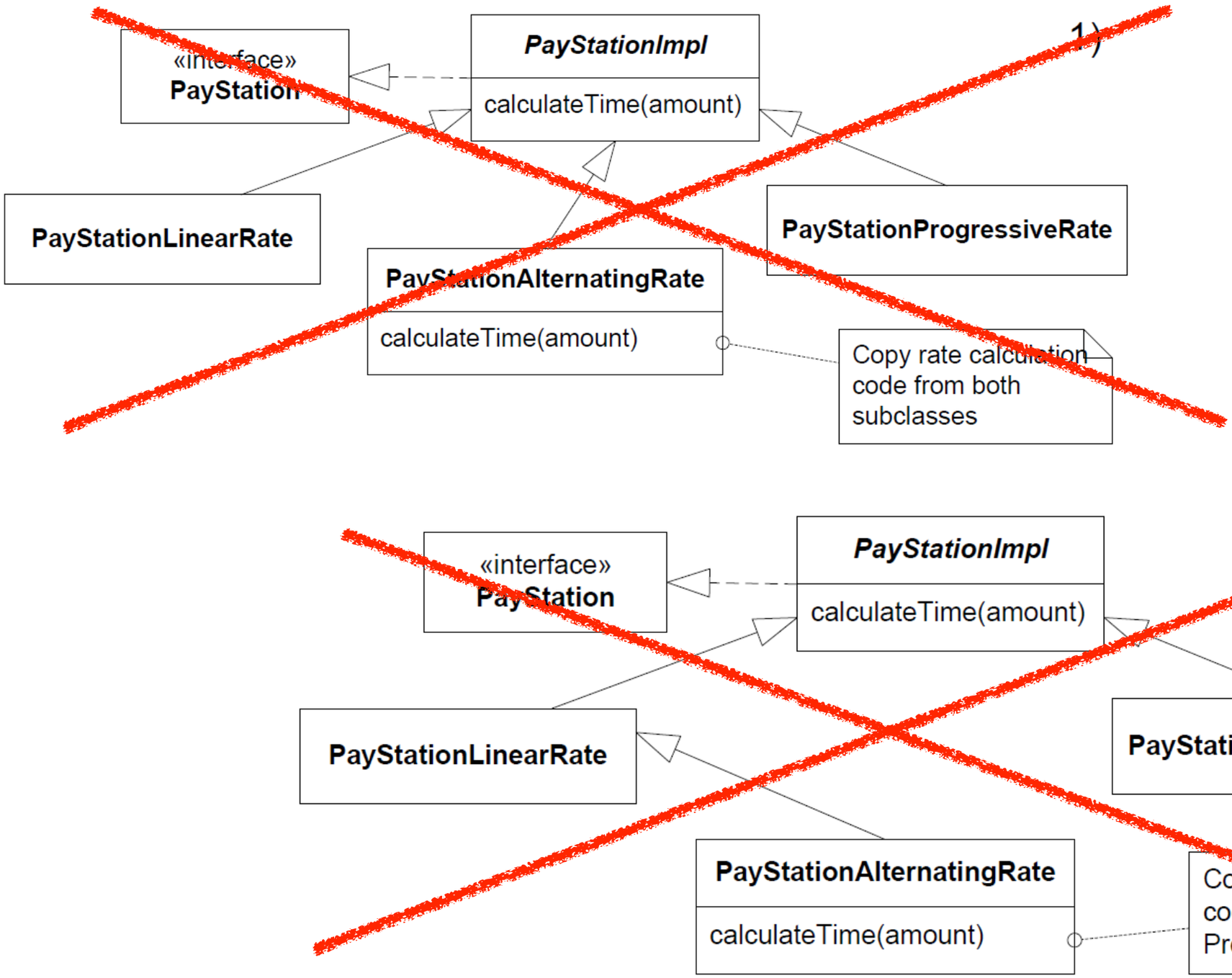

Gammatown County wants:
"In weekdays we need Alphatown rate (linear);
in weekends Betatown rate (progressive)"




## Exercise: HOW?

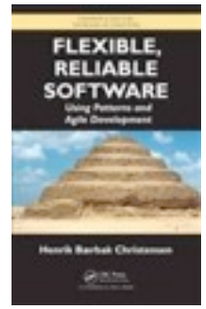
# Polymorphic Solutions


1)

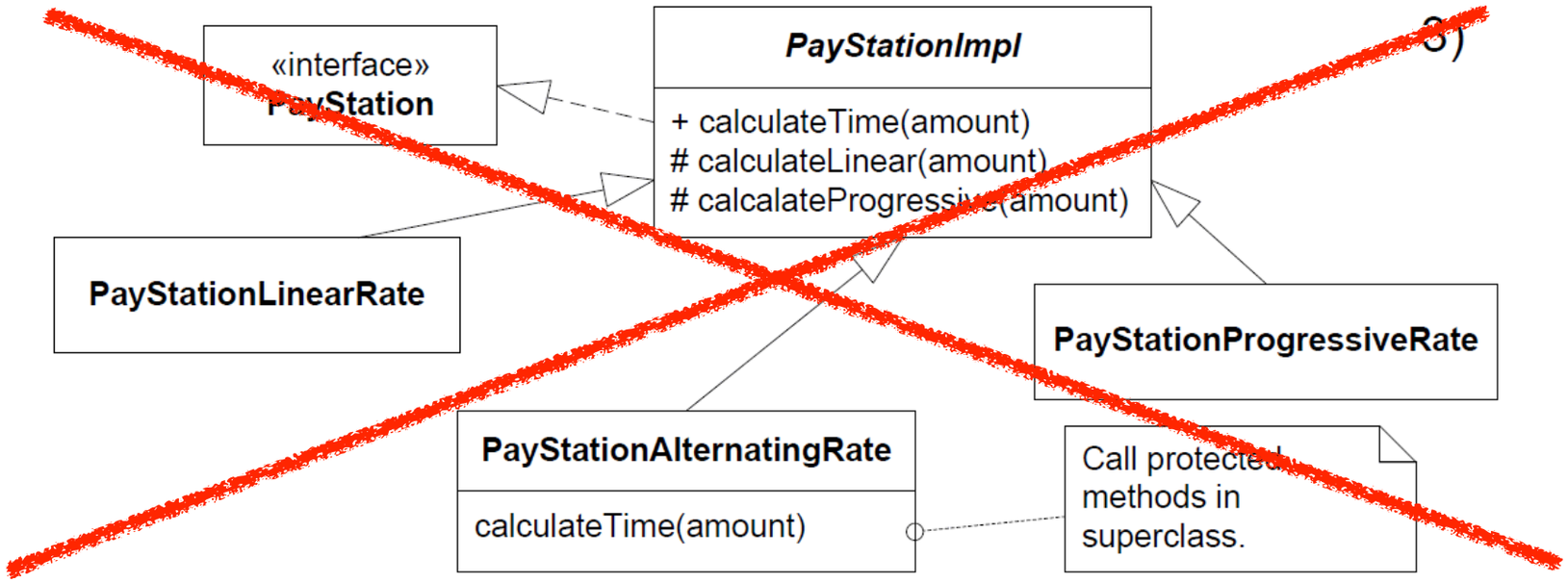



2)

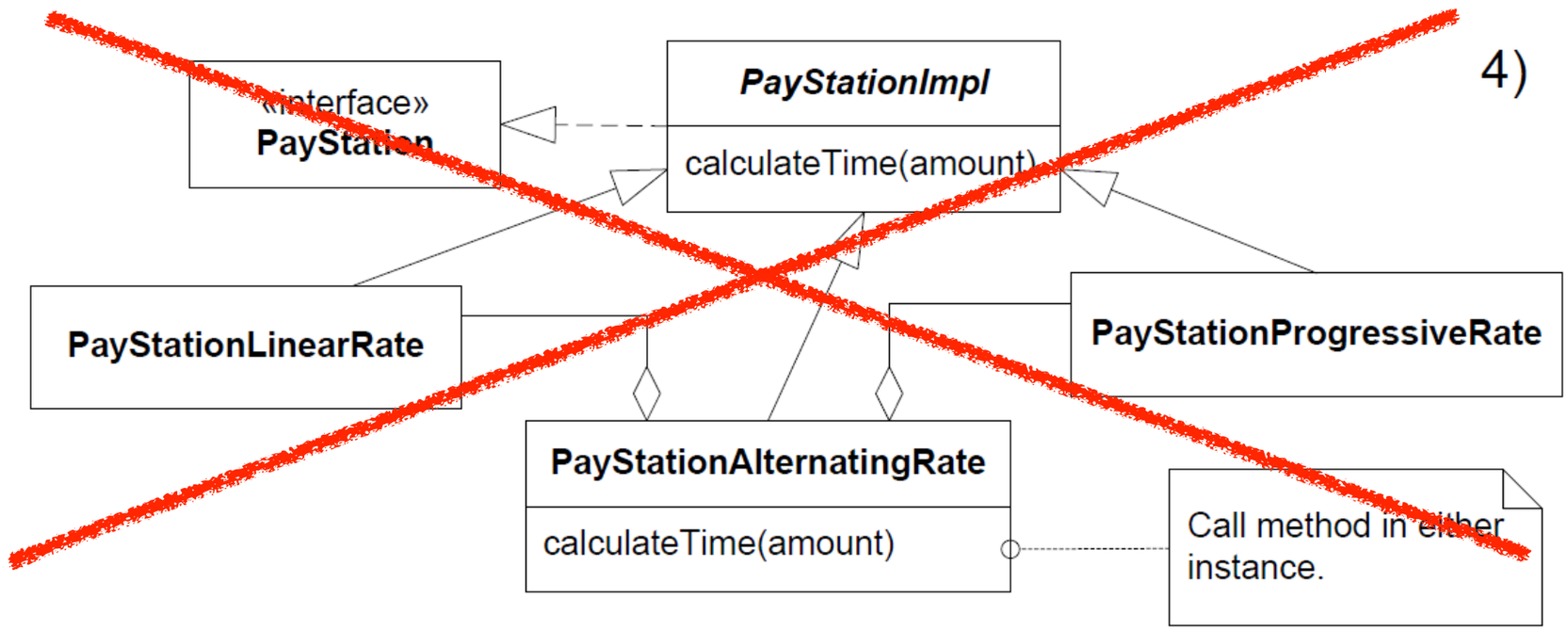
# Polymorphic Solution




3)




4)

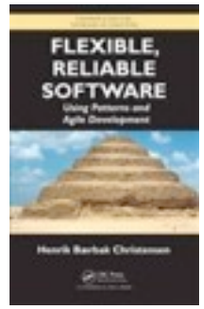


# Compositional + Parameter


```
public class PayStationImpl implements PayStation {
  [...]
  /** the strategy for rate calculations */
  private RateStrategy rateStrategyWeekday;
  private RateStrategy rateStrategyWeekend;

  /** Construct a pay station. */
  public PayStationImpl( RateStrategy rateStrategyWeekday,
                         RateStrategy rateStrategyWeekend ) {
    this.rateStrategyWeekday = rateStrategyWeekday;
    this.rateStrategyWeekend = rateStrategyWeekend;
  }
  public void addPayment( int coinValue ) throws IllegalCoinException {
    [...]
    if ( isWeekend() ) {
      timeBought = rateStrategyWeekend.calculateTime(insertedSoFar);
    } else {
      timeBought = rateStrategyWeekday.calculateTime(insertedSoFar);
    }
  }
  [...]
  private boolean isWeekend() {
  [...]
  }
}
```

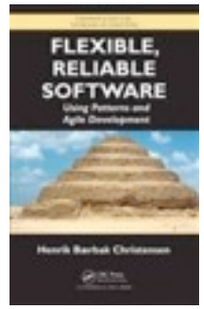
Terrible solution too!!!

# Compositional Process



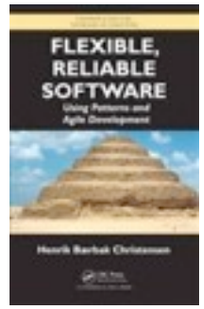

- We have **identified some behaviour that varies**.
  - The rate calculation behaviour is what must vary for Gammatown and this we have already identified.
- We **stated a responsibility** that **covers the behaviour** that varies and encapsulate it by expressing it as an interface.
  - The RateStrategy interface already defines the responsibility to "Calculate parking time" by defining the method calculateTime.
- We compose the resulting behaviour by **delegating** the concrete behaviour to subordinate objects.
  - This is the point that takes on a new meaning concerning our new requirement.

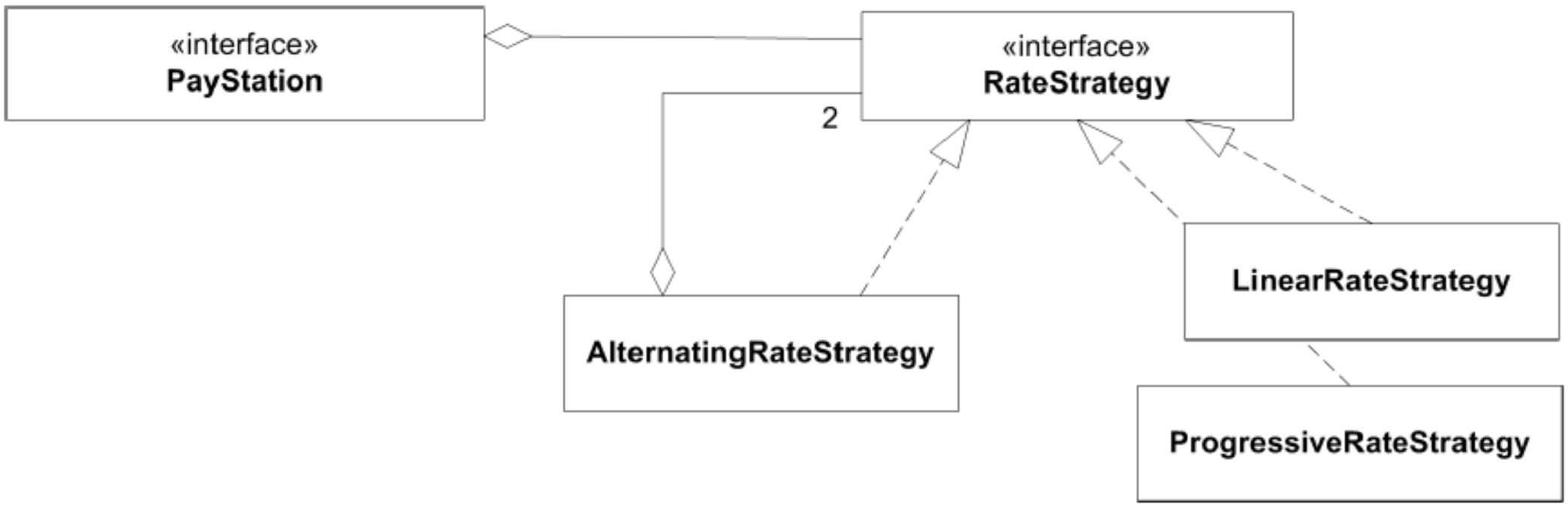
# Compose the behaviour



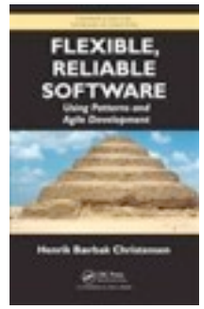

- That is:
  - the best object to calculate linear rate models has already been defined and tested – why not use its expertise ? Same goes with progressive rate.
  - so let us make a **small team** – one object responsible for taking the decision; the two other responsible for the individual rate calculations.

# Solution


FLEXIBLE,
RELIABLE
SOFTWARE
Using Patterns and
Agile Development
Henrik Bærbak Christensen


«interface»
PayStation
«interface»
RateStrategy
2
AlternatingRateStrategy
LinearRateStrategy
ProgressiveRateStrategy

# Code View

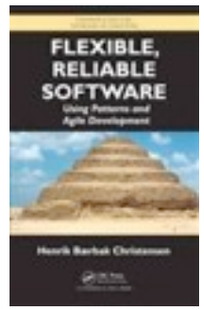



```
public class AlternatingRateStrategy implements RateStrategy {
  RateStrategy weekendStrategy, weekdayStrategy, currentState;
  public AlternatingRateStrategy( RateStrategy weekdayStrategy,
                                  RateStrategy weekendStrategy ) {
    this.weekdayStrategy = weekdayStrategy;
    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);
  }
}
```

Check the clock

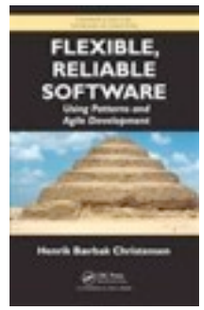
Delegate to expert

# Consequences



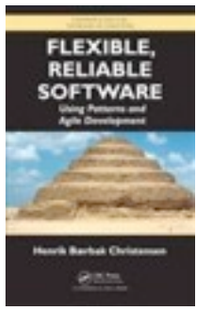

- Minimal new code, thus very little to test
  - most classes are untouched, only one new is added.
- Change by addition, not modification
- No existing code is touched
  - so no new testing
  - no review
- Parameterization of constructor
  - All models possible that differ in weekends...

# Importance



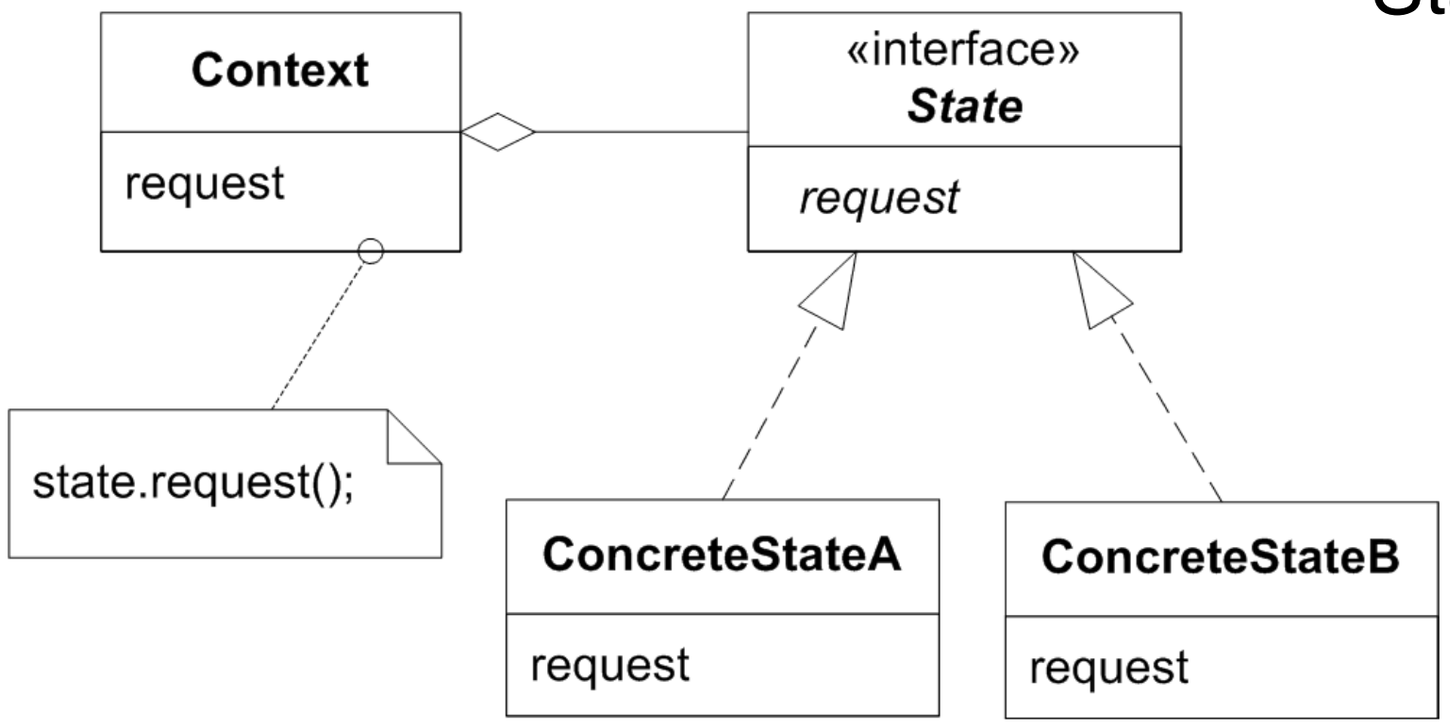

- again the importance of:
  - Encapsulate what **varies**: the rate policy
  - Define well-defined **responsibilities** by interfaces
  - Only let objects **communicate** using the interfaces
    - Then the respective roles (pay station / rate strategy) can be played by many different concrete objects
    - And each object is free to implement the responsibilities of the roles as it sees fit
  - also to let most of the dirty job be done by others.

# State Pattern




- **Intent**
  - **Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.**
  - The rate policy algorithm alters its behaviour according to the state of the system clock
  - Seen from the *PayStationImpl* the *AlternatingRateStrategy* object appears to change class because it changes behaviour over the week.

# Roles


State changes?

May be defined either in

Context or in

ConcreteState

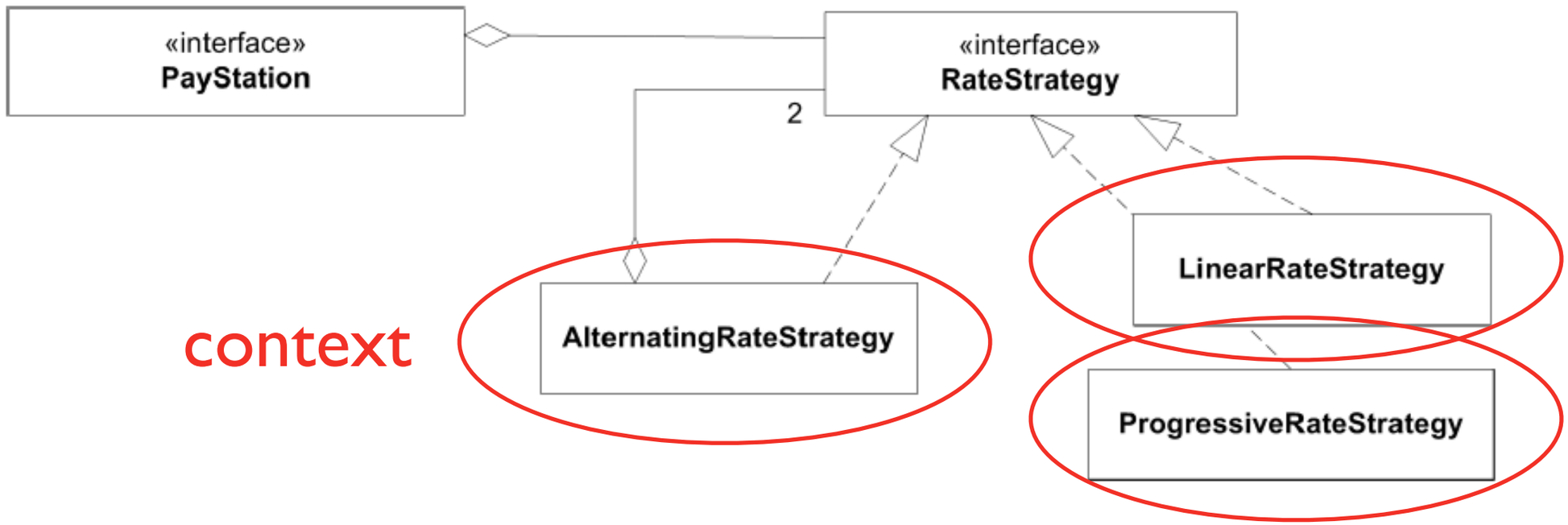
subclasses decide

**Context** delegate to its current state object

**State** specifies responsibilities of the behaviour that varies according to state

**ConcreteState** defines state specific behaviour

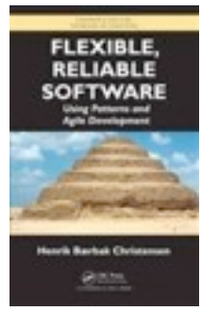
# Exercise


Which object/interface fulfils which role in the pay station?

Who is responsible for state changes?

# Consequences



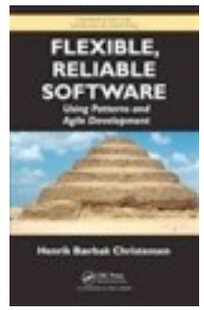

- + State specific behaviour is localized
  - in a single ConcreteState object
- + State changes are explicit
  - as you just find the assignments of ‘currentState’
- - Increased number of objects
  - as always with compositional designs

# What are design patterns?

# Definition

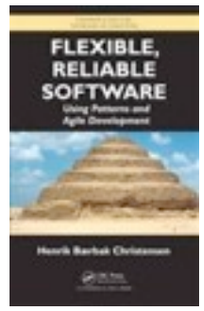
- **Design Patterns** 
  are descriptions of communicating objects and classes that are customised to solve a general design problem in a particular context
- Elements of a design pattern:
  - A pattern name
  - The problem that the pattern solves
    Including conditions for the pattern to be applicable
  - The solution to the problem brought by the pattern.
    The elements (classes-objects) involved, their roles, responsibilities, relationships and collaborations
    Not a particular concrete design or implementation
  - The consequences of applying the pattern
    Time and space trade off
    Language and implementation issues
    Effects on flexibility, extensibility, portability

# Differentiating Patterns




- Be aware that many patterns are *structurally equal.*
  - their UML class diagrams are more or less identical!
- Patterns are defined by the *problem they solve!*
- Strategy is the problem of
  - *Handling variability of algorithms / business rules, making them interchangeable*
- State is the problem of
  - *providing behavior that varies according to object's internal state*

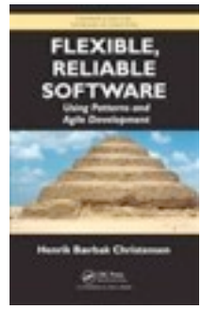
# Test Stubs



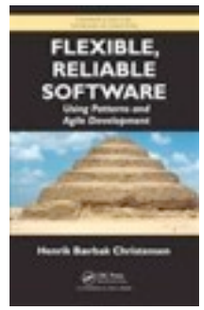

# Learning Objectives

- to show the problems of doing test-driven development when the production code uses resources that are not under direct testing control
- to learn the terminology for test stubs and
- to show how they help us in our quest to automate testing as much as possible.

# Road map




- New requirement
  - demanding that all rate strategies are under fully automatic testing control.
- Definition of direct and indirect input
- Discussion on ways to handle indirect input




# Test Cases

A test case for AlphaTown :

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

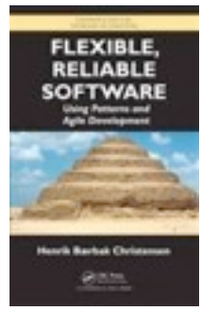
A test case for GammaTown :

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

Gammatown has one more parameter in the rate policy test case

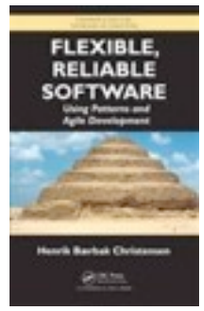
***This parameter is not accessible from the testing code!***

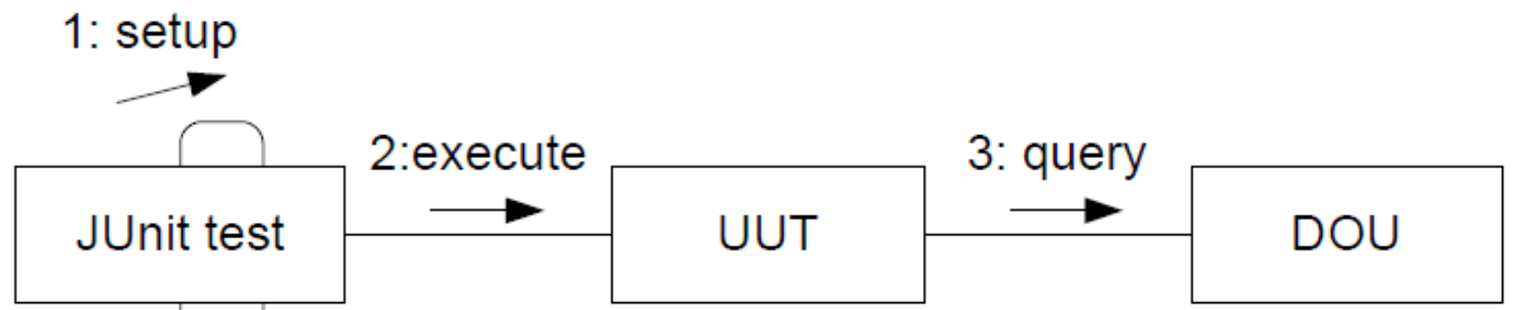
# Definition of Parameters




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

# Structure of xUnit Tests



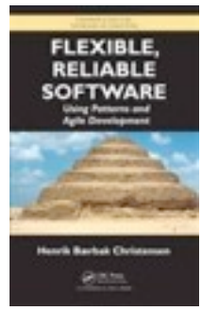




system clock
java.util.Calendar

## Depended-on Unit (DOU)

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

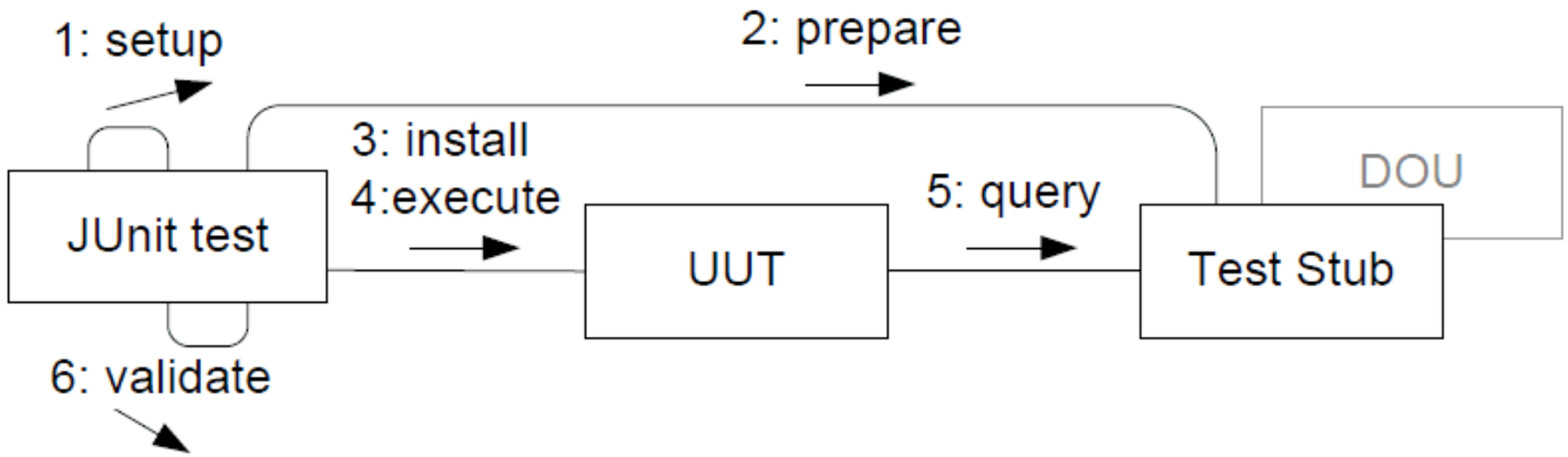
```
@Test public void shouldDisplay120MinFor300cent() {
  RateStrategy rs =
    new AlternatingRateStrategy( new LinearRateStrategy(),
                                 new ProgressiveRateStrategy() );
  assertEquals( 300 / 5 * 2, rs.calculateTime(300) );
}
```



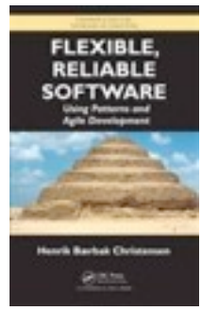

# Test Stub

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

- **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.

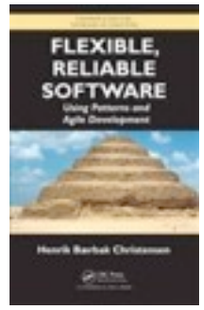


# Key point

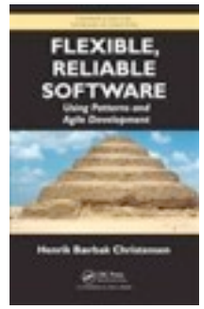



- **Test stubs make software testable**
  Many software units depend on direct input that influence their behaviour.
  Typical indirect input are external resources like hardware sensors, random-number generators, system clocks etc.
  Test stubs replace the real units and allow the testing code to control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

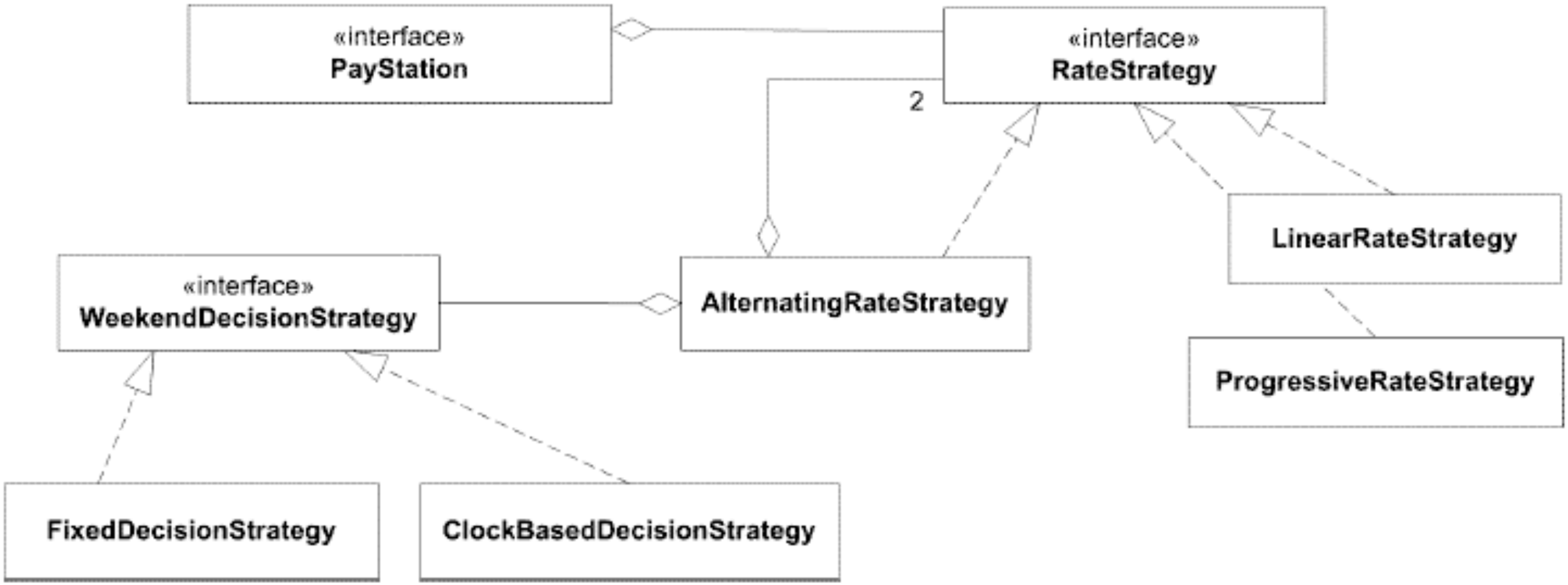
# Implementation



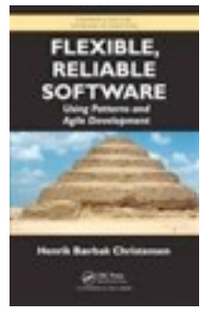

- Basically it is a variability problem
  - during testing, use data given by test code
  - during normal operation, use data given by system
- Remember:
  - identify some behaviour that varies.
    - it is the behaviour defined by isWeekend() that is variable.
  - state the responsibility that covers the behaviour that varies by an interface.
  - compose the desired behaviour by delegating


FLEXIBLE,
RELIABLE
SOFTWARE
Using Patterns and
Agile Development
Henrik Bærbak Christensen

# Solution

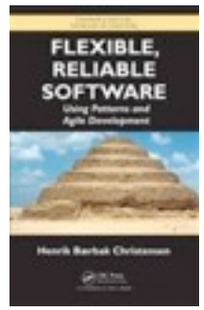

«interface»
PayStation
«interface»
RateStrategy
2
LinearRateStrategy
ProgressiveRateStrategy
«interface»
WeekendDecisionStrategy
AlternatingRateStrategy
FixedDecisionStrategy
ClockBasedDecisionStrategy

# Developing the solution




- **Iteration 1: Refactoring**
  - Introduce the new interface.
    - Refactor the existing *AlternatingRateStrategy* to take instances of this interface as parameter in the constructor. See that it compiles but the tests fail.
    - Refactor the existing design to make all test cases pass again. This will require introducing the *ClockBasedDecisionStrategy*.

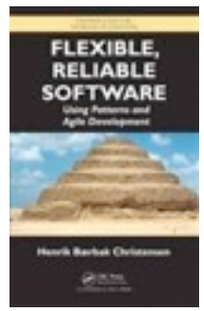
# Iteration 2: Test stub




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;
  }
}
```

# Setting it up



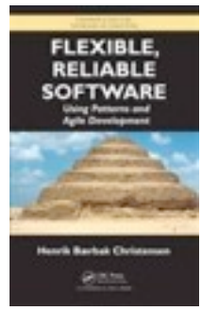

| Input | Expected output |
|---|---|
| pay = 300 cent, day = Wednesday | 120 min. |

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

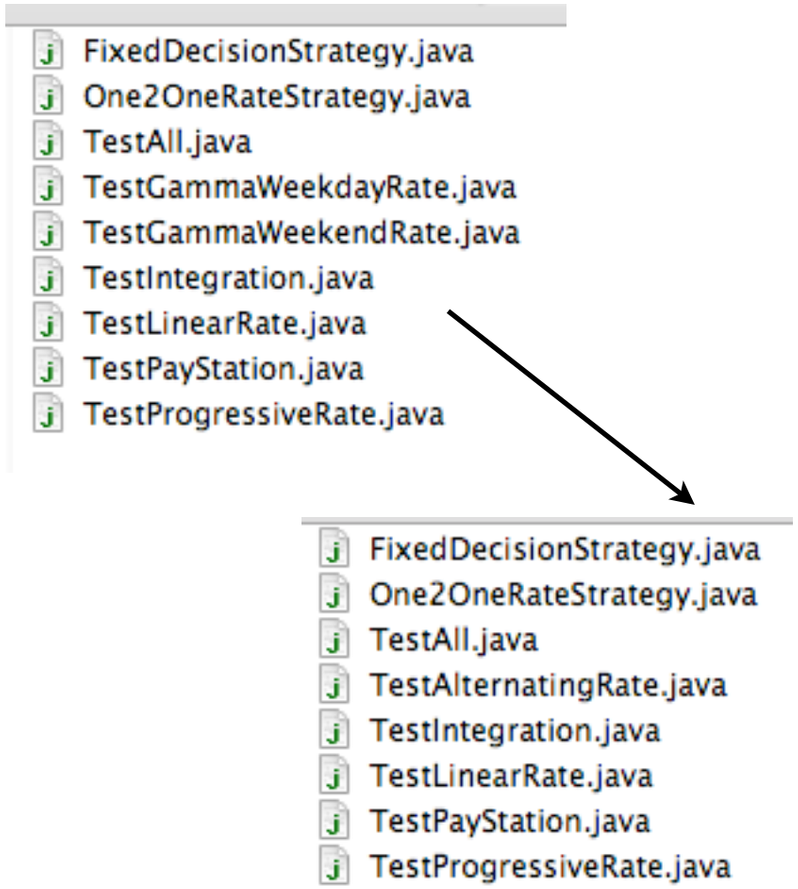
```
@Test public void shouldDisplay120MinFor300cent() {
  RateStrategy rs =
    new AlternatingRateStrategy( new LinearRateStrategy(),
                                 new ProgressiveRateStrategy(),
                                 new FixedDecisionStrategy(false));
  assertEquals( 300 / 5 * 2, rs.calculateTime(300)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



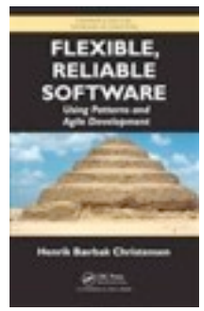

# Iteration 3

- Making a *TestAlternatingRate*
  - moving all Gammatown rate policy test cases here,
  - deleting the two old test case classes.
- Modifying *TestAll* so it includes the new test cases.

```
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 Doubles




- Test Stub is a subtype of Test Double. Other subtypes exists:
  - **Stub**: get indirect input under control
  - **Spy**: get indirect output under control
    - record the UUT's indirect output for (later) verification by the test case.
  - **Mock**: a spy with fail fast property
    - created and programmed dynamically by a mock library
  - **Fake**: a lightweight but realistic double
    - purpose is to be a high performance replacement for a slow or expensive DOU

- For more details, see: *xUnit Test Patterns. Refactoring Test Code.* G. Meszaros. Addison Wesley Signature Series. 2007

# Example Test Spy

```
public void testRemoveFlight() throws Exception {
   // setup
   FlightDto expectedFlightDto = createARegisteredFlight();
   FlightManagementFacade facade = new FlightManagementFacadeImpl();
   // exercise
   facade.removeFlight(expectedFlightDto.getFlightNumber());
   // verify
   assertFalse("flight should not exist after being removed",
               facade.flightExists( expectedFlightDto.
                                          getFlightNumber()));
}
```

```
public void testRemoveFlightLogging_recordingTestStub()
         throws Exception {
   // fixture setup
   FlightDto expectedFlightDto = createAnUnregFlight();
   FlightManagementFacade facade = new FlightManagementFacadeImpl();
   //    Test Double setup
   AuditLogSpy logSpy = new AuditLogSpy();
   facade.setAuditLog(logSpy);
   // exercise
   facade.removeFlight(expectedFlightDto.getFlightNumber());
   // verify
   assertFalse("flight still exists after being removed",
               facade.flightExists( expectedFlightDto.
                                          getFlightNumber()));
   assertEquals("number of calls", 1,
                logSpy.getNumberOfCalls());
   assertEquals("action code",
                Helper.REMOVE_FLIGHT_ACTION_CODE,
                logSpy.getActionCode());
   assertEquals("date", helper.getTodaysDateWithoutTime(),
                logSpy.getDate());
   assertEquals("user", Helper.TEST_USER_NAME,
                logSpy.getUser());
   assertEquals("detail",
                expectedFlightDto.getFlightNumber(),
                logSpy.getDetail());
}
```

```
public class AuditLogSpy implements AuditLog {
   // Fields into which we record actual usage information
   private Date date;
   private String user;
   private String actionCode;
   private Object detail;
   private int numberOfCalls = 0;

   // Recording implementation of real AuditLog interface
   public void logMessage(Date date,
                          String user,
                          String actionCode,
                          Object detail) {
      this.date = date;
      this.user = user;
      this.actionCode = actionCode;
      this.detail = detail;

      numberOfCalls++;
   }

   // Retrieval Interface
   public int getNumberOfCalls() {
      return numberOfCalls;
   }
   public Date getDate() {
      return date;
   }
   public String getUser() {
      return user;
   }
   public String getActionCode() {
      return actionCode;
   }
   public Object getDetail() {
      return detail;
   }
}
```

# Example Fake Object

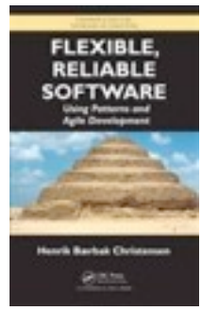
```
public void testReadWrite() throws Exception{
   // Setup
   FlightMngtFacade facade = new FlightMgmtFacadeImpl();
   BigDecimal yyc = facade.createAirport("YYC", "Calgary", "Calgary");
   BigDecimal lax = facade.createAirport("LAX", "LAX Intl", "LA");
   facade.createFlight(yyc, lax);
   // Exercise
   List flights = facade.getFlightsByOriginAirport(yyc);
   // Verify
   assertEquals( "# of flights", 1, flights.size());
   Flight flight = (Flight) flights.get(0);
   assertEquals( "origin",
                 yyc, flight.getOrigin().getCode());
}
```

```
public void testReadWrite_inMemory() throws Exception{
   // Setup
   FlightMgmtFacadeImpl facade = new FlightMgmtFacadeImpl();
   facade.setDao(new InMemoryDatabase());
   BigDecimal yyc = facade.createAirport("YYC", "Calgary", "Calgary");
   BigDecimal lax = facade.createAirport("LAX", "LAX Intl", "LA");
   facade.createFlight(yyc, lax);
   // Exercise
   List flights = facade.getFlightsByOriginAirport(yyc);
   // Verify
   assertEquals( "# of flights", 1, flights.size());
   Flight flight = (Flight) flights.get(0);
   assertEquals( "origin",
                 yyc, flight.getOrigin().getCode());
}
```

```
public BigDecimal createAirport( String airportCode,
                                 String name,
                                 String nearbyCity)
throws FlightBookingException{
   TransactionManager.beginTransaction();
   Airport airport = dataAccess.
         createAirport(airportCode, name, nearbyCity);
   logMessage("Wrong Action Code", airport.getCode());//bug
   TransactionManager.commitTransaction();
   return airport.getId();
}

public List getFlightsByOriginAirport(
                 BigDecimal originAirportId)
      throws FlightBookingException {

   if (originAirportId == null)
      throw new InvalidArgumentException(
              "Origin Airport Id has not been provided",
              "originAirportId", null);
   Airport origin = dataAccess.getAirportByPrimaryKey(originAirportId);
   List flights = dataAccess.getFlightsByOriginAirport(origin);

   return flights;
}
```

```
public class InMemoryDatabase implements FlightDao{
   private List airports = new Vector();
   public Airport createAirport(String airportCode,
                                String name, String nearbyCity)
         throws DataException, InvalidArgumentException {
      assertParamtersAreValid(  airportCode, name, nearbyCity);
      assertAirportDoesntExist( airportCode);
      Airport result = new Airport(getNextAirportId(),
            airportCode, name, createCity(nearbyCity));
      airports.add(result);
      return result;
   }
   public Airport getAirportByPrimaryKey(BigDecimal airportId)
         throws DataException, InvalidArgumentException {
      assertAirportNotNull(airportId);

      Airport result = null;
      Iterator i = airports.iterator();
      while (i.hasNext()) {
           Airport airport = (Airport) i.next();
           if (airport.getId().equals(airportId)) {
              return airport;
           }
        }
        throw new DataException("Airport not found:"+airportId);
     }
```

# Discussion




- Test Stubs make software testable.
- Compositional process helps isolating DOUs.
- The solution is overly complex?
  - perhaps but it scales well to complex DOUs
- Some code units are not automatically testable in a cost-efficient manner.
- Do not test that the return values from the system library methods are correct.