AARHUS UNIVERSITET

Software Engineering and Architecture

Deriving State Pattern

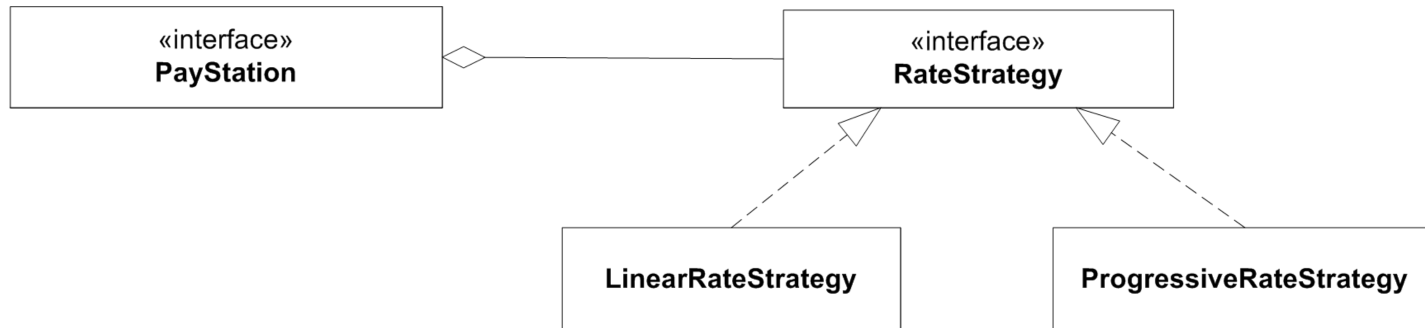
Combining Behavior

AARHUS UNIVERSITET

New requirement

- Gammatown County wants:

  “In weekdays we need Alphatown rate (linear);
  in weekends Betatown rate (progressive)”



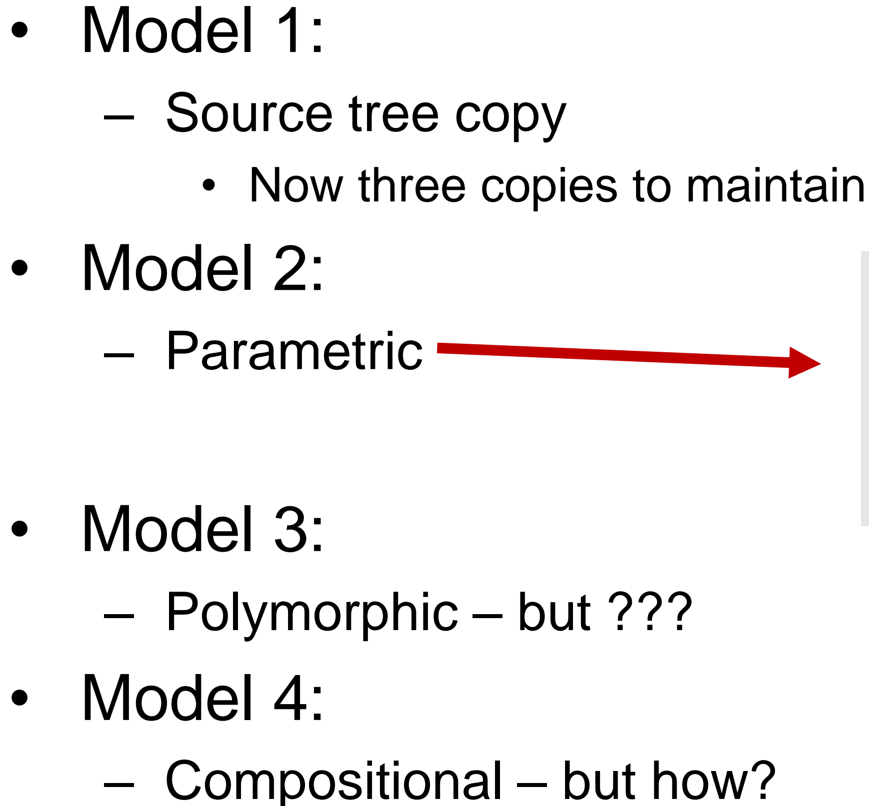
Exercise

AARHUS UNIVERSITET

"In weekdays we need Alphatown rate (linear);
in weekends Betatown rate (progressive)"

- Exercise: **How?**

Same Analysis



- Model 1:
  - Source tree copy
    - Now three copies to maintain
- Model 2:
  - Parametric

```
if (town == Town.ALPHATOWN) {
  timeBought = insertedSoFar * 2 / 5;
} else if (town == Town.BETATOWN) {
  [BetaTown implementation]
} else if (town == Town.GAMMATOWN) {
  [GammaTown implementation]
}
```

- Model 3:
  - Polymorphic – but ???
- Model 4:
  - Compositional – but how?

But…

- I will return to the analysis shortly, but first…

- ***I have a problem!***
  - I want to do TDD – because automated tests feel good…
  - But how can I write *test first* when the outcome of a GammaTown rate strategy… *depends on the day of the week???*

Tricky Requirement

- The test case for AlphaTown:

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

- … but how does it look for GammaTown?

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

Direct and Indirect Parameters

- The day of the week is called an *indirect parameter* to the *calculateTime* method
  - It is not an instance variable of the object
  - It is not a parameter to the method
  - **It cannot be set by our JUnit code ☹**
    - It is 'set' by the computer's clock
      - That is, a parameter set *indirectly* by something *outside* our JUnit test code…

Solutions?

AARHUS UNIVERSITET

- So – what to do?
  - Come in on weekends?
    - Manual testing!
  - Set the clock ?
    - Manual testing!
    - Messes up Gradle as it depends on the clock going forward!
  - Refactor code to make Pay Station accept a Date object?
    - No – pay stations must continuously ask for date objects every time a new coin is entered…

- ***I will return to this problem set soon…***

AARHUS UNIVERSITET

Polymorphic Solutions to the GammaTown Challenge

Using class inheritance

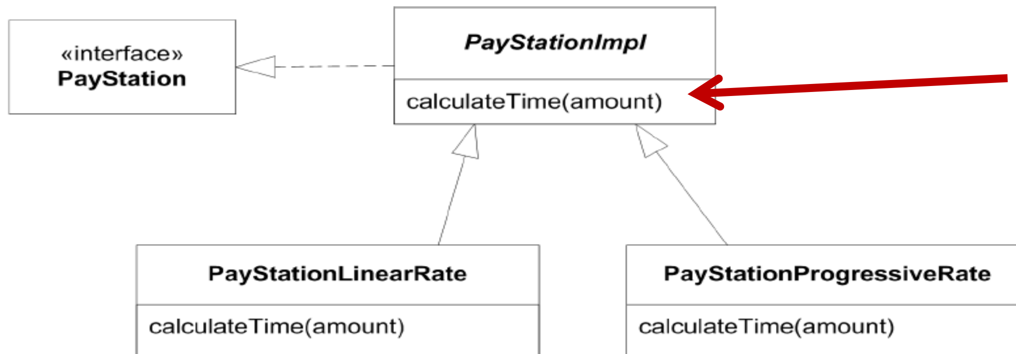
Premise

- Let us *assume* that we have developed the *polymorphic solution* to handle BetaTown!

- **That is: forget the Strategy based solution we did last time for the next analysis...**

Reviewing the Polymorphic

- So – how did the polymorphic solution look like:
  - Make *PayStationImpl* abstract, *calculateTime* abstract
  - Two subclasses, one for linear and one for progressive



abstract

```
public abstract class PayStationAbstract implements PayStation {
  private int insertedSoFar;
  private int timeBought;

  @Override
  public void addPayment(int coinValue) throws IllegalCoinExceptio
    if (coinValue != 5 && coinValue != 10 && coinValue != 25) {
      throw new IllegalCoinException("The entered coin is not vali
 + coinValue);
    }
    insertedSoFar += coinValue;
    timeBought = calculateTime(insertedSoFar);
  }

  abstract int calculateTime(int paidSoFar);

  @Override
  public int readDisplay() {
    return timeBought;
  }
```

The Concrete Classes

```
public class PayStationLinear extends PayStationAbstract {
  @Override
  int calculateTime(int paidSoFar) {
    return paidSoFar / 5 * 2;
  }
}
```

AlphaTown = Linear

```
public class PayStationProgressiveRate extends PayStationAbstract {
  @Override
  protected int calculateTime(int paidSoFar) {
    int time = 0;
    if (paidSoFar >= 150+200) { // from 2nd hour onwards
      paidSoFar -= 350;
      time = 120 + paidSoFar / 5;
    } else if (paidSoFar >= 150) { // from 1st to 2nd hour
      paidSoFar -= 150;
      time = 60 + paidSoFar *3 / 10;
    } else { // up to first hour
      time = paidSoFar / 2 * 5;
    }
    return time;
  }
}
```

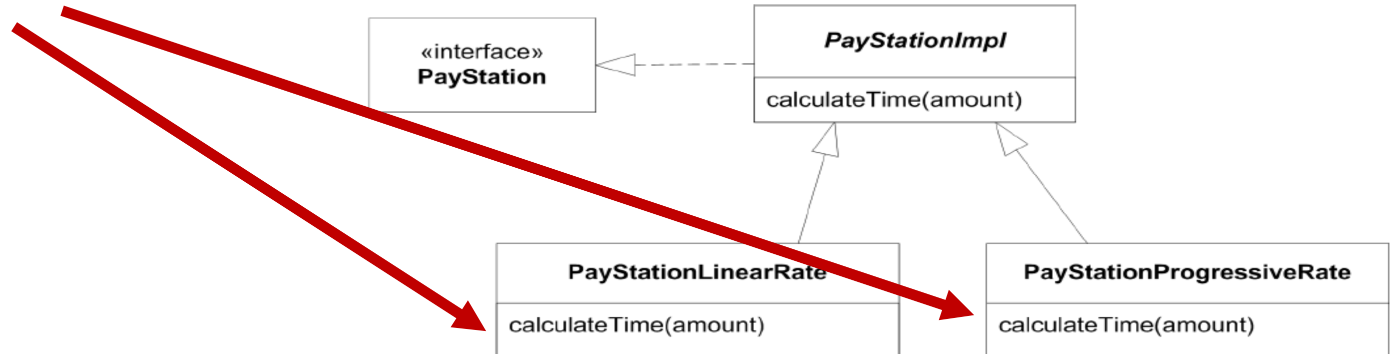
BetaTown = Progressive

The Big Challenge

```
public class PayStationLinear extends PayStationAbstract {
  @Override
  int calculateTime(int paidSoFar) {
    return paidSoFar / 5 * 2;
  }
}
```

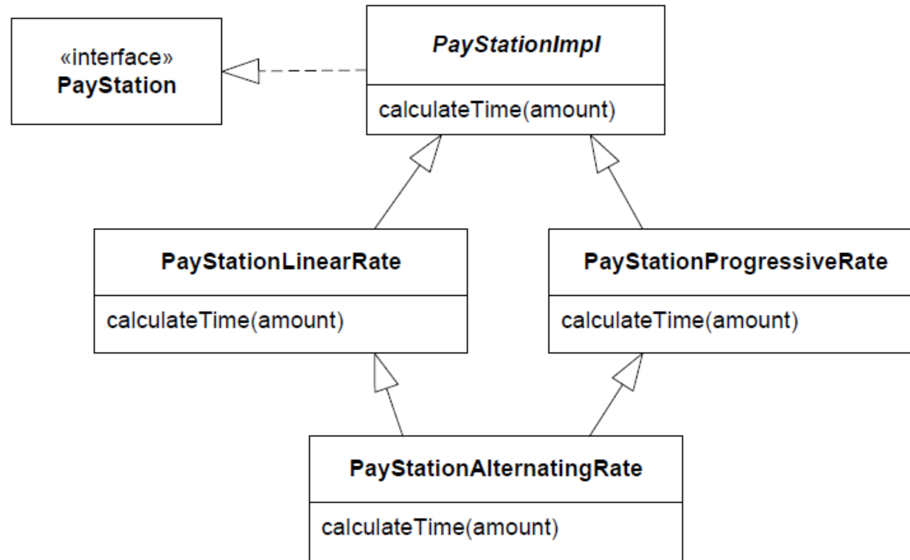
```
public class PayStationProgressiveRate extends PayStationAbstract {
  @Override
  protected int calculateTime(int paidSoFar) {
    int time = 0;
    if (paidSoFar >= 150+200) { // from 2nd hour onwards
      paidSoFar -= 350;
      time = 120 + paidSoFar / 5;
    } else if (paidSoFar >= 150) { // from 1st to 2nd hour
      paidSoFar -= 150;
      time = 60 + paidSoFar *3 / 10;
    } else { // up to first hour
      time = paidSoFar / 2 * 5;
    }
    return time;
  }
}
```

- How do I make a subclass which has **both these algorithms?**
  - They are in two different classes!!!



Model 3a: Multiple Inheritance

- Subclass and override!



C++ Code

```
If (isWeekend()) {
  PSProgressive::calcTime(…);
} else {
  PSLinear::calcTime(…);
}
```

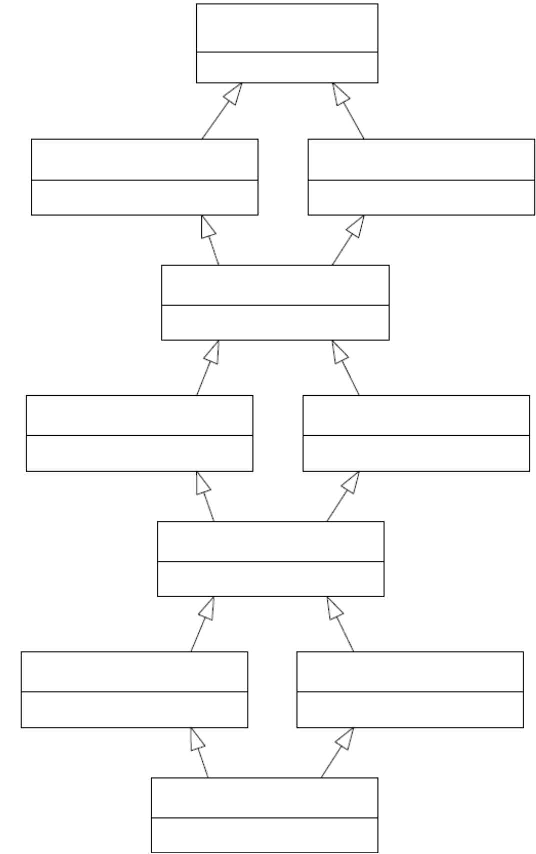
- Could work in C++, but not Java(*) or C#;
  - My experience with fork-join hierarchies in C++ are bad ☹

Fork-Join Hierarchies

- This is a *fork-join hierarchy*

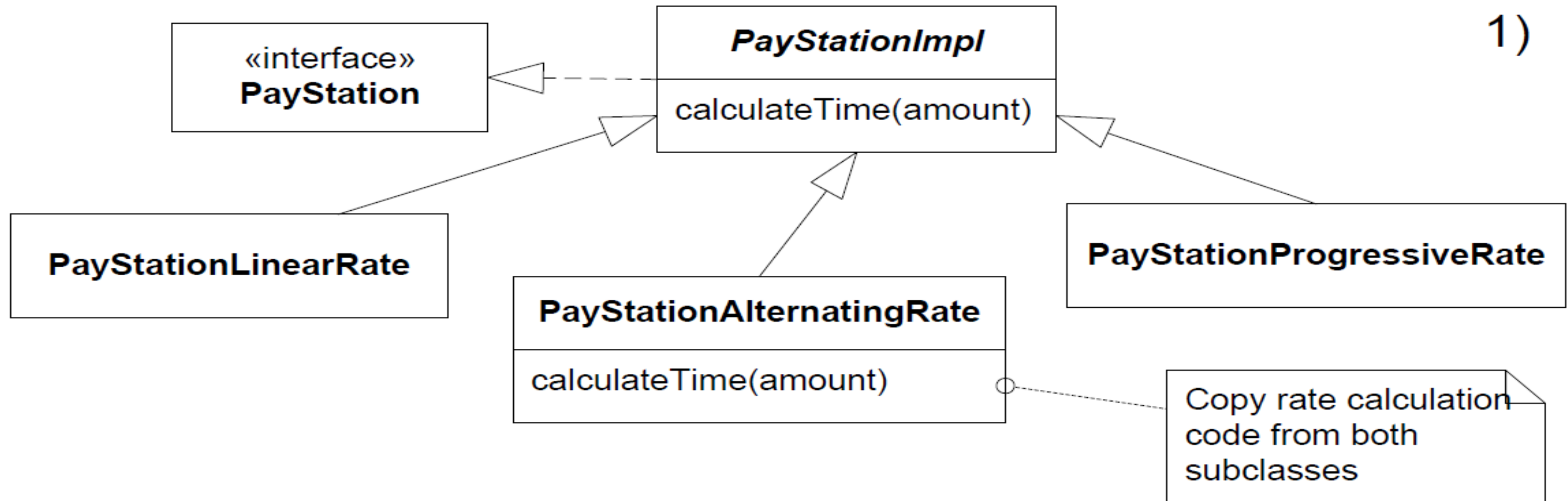
- Fork-join =
  - A root class – that has
  - Two subclasses – that
  - A single class inherit from – that has
  - Two subclasses – that
  - …

- My experience is … *bad…*



AARHUS UNIVERSITET

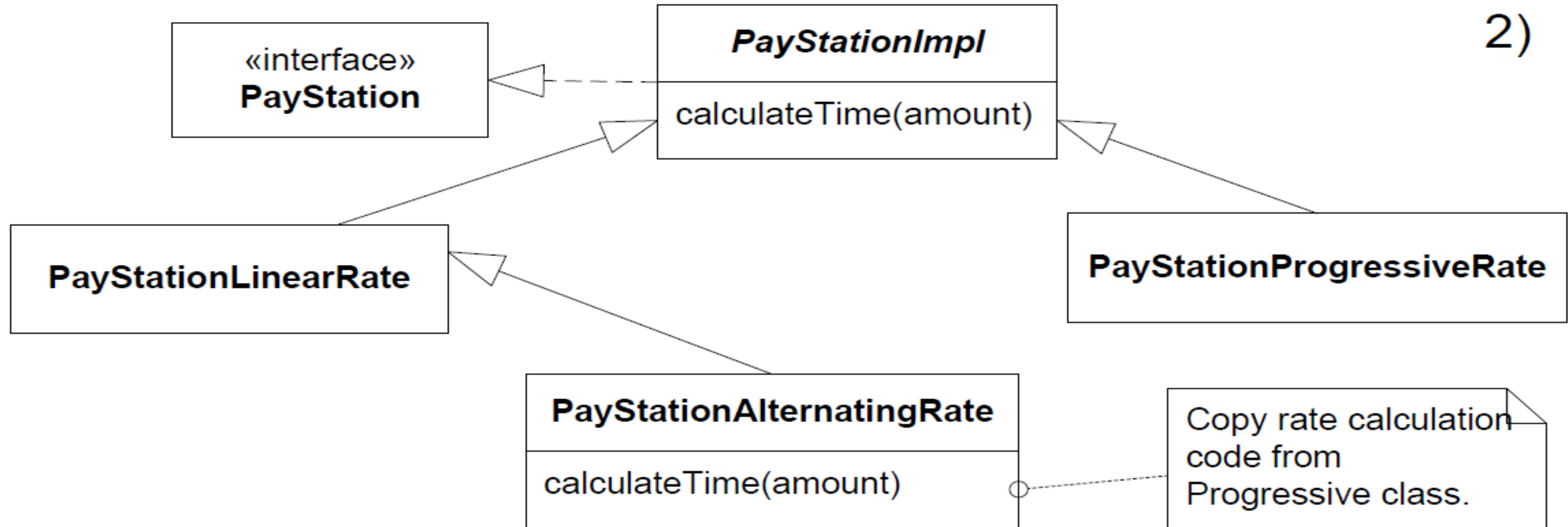
Model 3b: Direct Subclass



- Cut code from linear and progressive, paste into alternating… And we have *multiple copies of code…*

AARHUS UNIVERSITET

Model 3c:Sub-sub Class



- Cut code from progressive, paste into alternating

AARHUS UNIVERSITET

Code view

```
public class PayStationAlternatingRate
  extends PayStationLinearRate {
  [...]
  private int calculateTime( int amount ) {
    int time;
    if ( isWeekend() ) {
      [Paste progressive calculation code here]
    } else {
      time = super.calculateTime( amount );
    }
    return time;
  }
}
```

Model 3d: Bubbling up/Superclass

3)

«interface»
PayStation

PayStationImpl

+ calculateTime(amount)
# calculateLinear(amount)
# calcalateProgressive(amount)

PayStationLinearRate

PayStationProgressiveRate

PayStationAlternatingRate

calculateTime(amount)

Call protected methods in superclass.

- Make protected calculation methods in abstract PayStationImpl, and call these from Alternating
  - This is a classic solution often seen in practice

Code view

- The super class

```
public class PayStationImpl implements PayStation {
  [...]
  protected int calculateLinearTime( int amount ) { [...] }
  protected int calculateProgressiveTime( int amount ) { [...] }
}
```

- Alpha then becomes

```
public class PayStationLinearStrategy
  extends PayStationImpl {
  [...]
  protected int calculateTime( int amount ) {
    return super.calculateLinearTime( amount );
  }
  [...]
}
```

Code view

- Gamma is then

```
public class PayStationAlternatingRate
  extends PayStationImpl {
  [...]
  protected int calculateTime( int amount ) {
    int time;
    if ( isWeekend() ) {
      time = super.calcProgressiveTime( amount );
    } else {
      time = super.calcLinearTime( amount );
    }
    return time;
  }
```

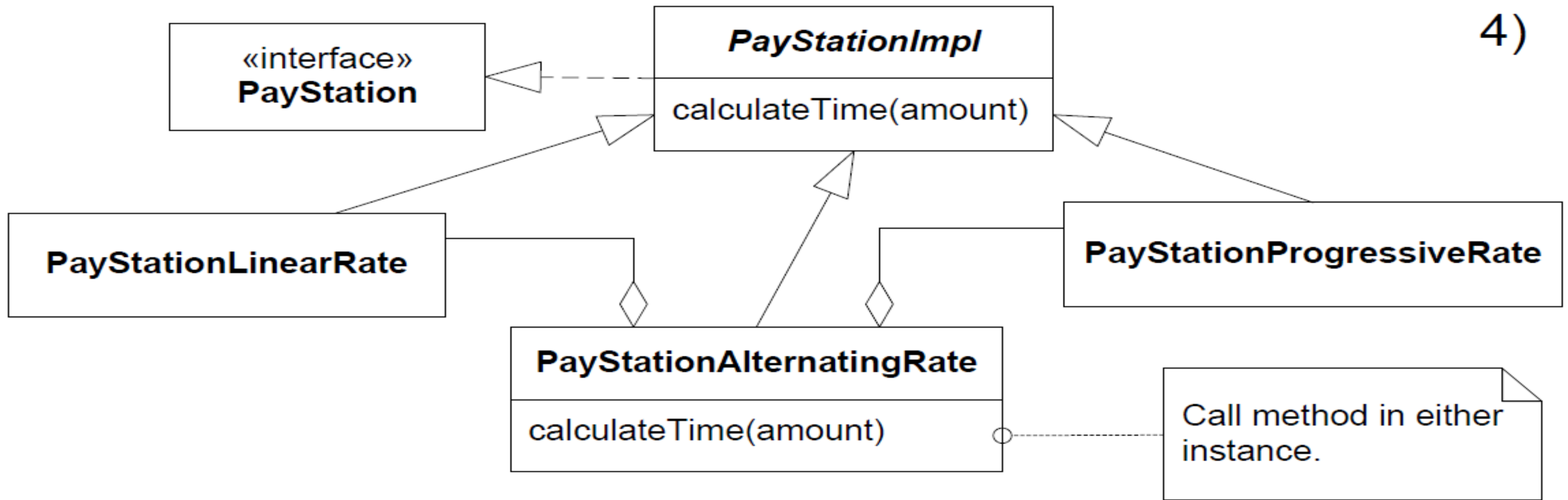
- Discussion
  - No code duplication
  - Exercise: what are the liabilities?

Liabilities

- Superclass **stability**
  - Tendency to modify super classes over and over again
- Superclass **analyzability** and cohesion
  - Becomes a *junk pile of methods* over time
  - The methods are **unrelated to the superclass itself**, it is just a convenient parking lot for them
  - *This is an example of an abstraction with little **cohesion***
  - Grave yard of forgotten methods?

AARHUS UNIVERSITET

Model 3e: Stations in Stations



Model 3e: Stations in Stations

- The “pay stations in pay station” way:
  - Create an gamma pay station containing both an alpha and beta pay station

```
public class PayStationAlternatingRate
  extends PayStationImpl {
  private PayStation psLinear, psProgressive;
  [...]
  private int calculateTime( int amount ) {
    int time;
    if ( isWeekend() ) {
      time = psProgressive.calculateTime( amount );
    } else {
      time = psLinear.calculateTime( amount );
    }
    return time;
  }
```

- Exercise: Benefits and liabilities?

Morale

- *It simply does not work cleanly!*
- I have never seen a polymorphic solution that handles this very simple requirement in a natural and concise way!

(*) SideBar

- Multiple inheritance of implementation is *evil IMO…*
- Java 8 managed to sneak it in anyway ☹
  - Default methods

```
interface PayStationImpl {
  int calculateTime(int amount);
}

interface PayStationLinearRate extends PayStationImpl {
  default int calculateTime(int amount) { return amount / 5 * 2; }
}

interface PayStationProgressiveRate extends PayStationImpl {
  default int calculateTime(int amount) { return amount / 5 * 8; }
}

class PayStationAlternatingRate implements PayStationLinearRate,
                                           PayStationProgressiveRate {
  boolean isWeekend;
  public int calculateTime(int amount) {
    if (isWeekend)
      return PayStationProgressiveRate.super.calculateTime(amount);
    else
      return PayStationLinearRate.super.calculateTime(amount);
  }
```

Do not use default methods for fork-join hierarchies. It is not its intended use! (Library evolution is)

```
csdev@m51f19hbc:~/proj/frsproject/state-fork-join$ java PayStationForkJoin2
=== PayStation Fork Join ===
 isWeekend == false. Calculate on 20 cents and get 8 minutes.
 isWeekend == true. Calculate on 20 cents and get 32 minutes.
```

AARHUS UNIVERSITET

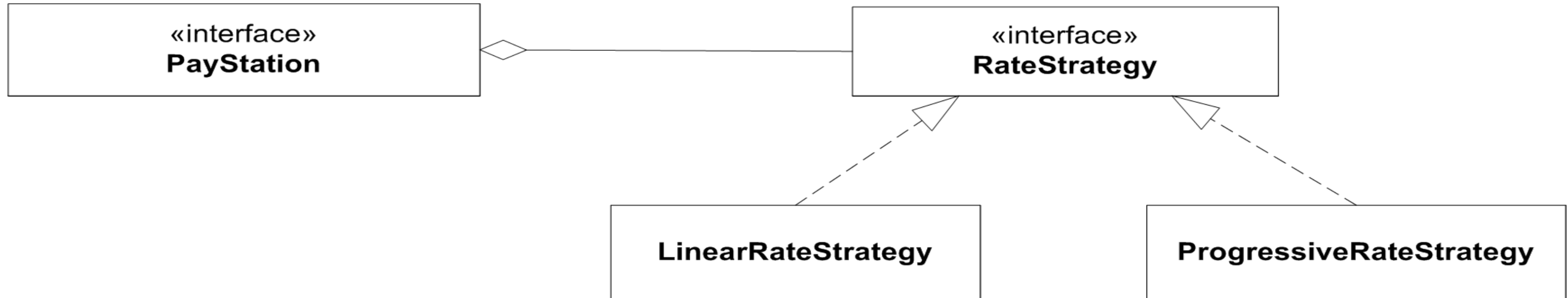
Compositional Variants

Premise

AARHUS UNIVERSITET

- Now, please reset your minds again!

- We now look at the *compositional variant (strategy pattern)* that we made the last time!



Code View

```
public class PayStationImpl implements PayStation {
  private int insertedSoFar;
  private int timeBought;

  /** the strategy for rate calculations */
  private RateStrategy rateStrategy;
```

```
public void addPayment(int coinValue)
        throws IllegalCoinException {
  switch (coinValue) {
  case 5:
  case 10:
  case 25: break;
  default:
    throw new IllegalCoinException("Invalid coin: "+coinValue);
  }
  insertedSoFar += coinValue;
  timeBought = rateStrategy.calculateTime(insertedSoFar);
}
```

Model 4a: Parameter + compositional

```
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() {
  [...]
  }
}
```

Model 4a: Parameter + compositional

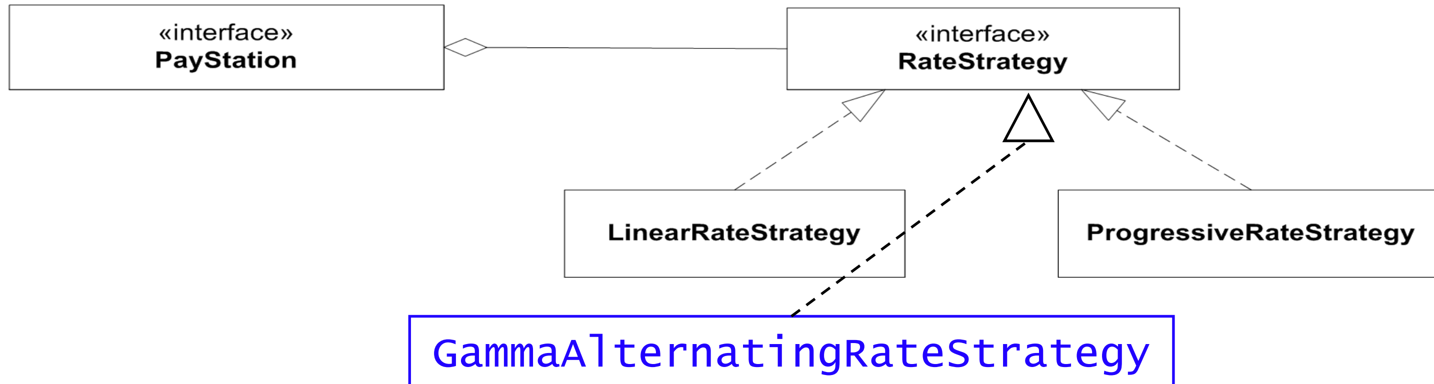
- Liabilities
  - Code change in the constructor
  - Constructor has become really weird for alpha and beta

- Worse: we have just blown the whole idea!
  - now the pay station has resumed the rate calculation responsibility ☠
  - or even worse – the responsibility is *distributed over several objects* ☠ ☠ ☠
    - *The responsibility to know about rate calculations are now distributed into two objects – leading to lower analyzability*
    - leads to duplicated code, and bugs difficult to track.

AARHUS UNIVERSITET

Model 4b: Copy and paste version

- Cut and paste the code into new strategy object



- **Multiple maintenance problem** 💣
  - a bug in price calculation functionality must be corrected in **two** places – odds are you only remember one of them.

Lesson Learned

- Often two variability techniques are used at the same time
  - Polymorphic + parametric
  - Polymorphic + source’code’copy
  - …
- … Which somewhat masks there is a *bit issue here*
- ***Do the same thing, the same way !!!***
  - **If your variability technique does not support it – it is because you are using the wrong technique ☺**

… on to a nice compositional solution: State pattern

Composition is *doing the same thing the same way*

Compositional Idea

- ③ *I identify some behavior that varies.*
  - The rate calculation behavior is what must vary for Gammatown and this we have already identified.
- ① *I state a responsibility that covers the behavior 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.
- ② *I compose the resulting behavior by delegating the concrete behavior to subordinate objects.*
  - This is the point that takes on a new meaning concerning our new requirement.

AARHUS UNIVERSITET

Model 4

- ***Compose the behavior...***

  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.

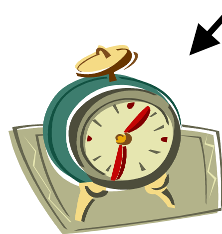
  - so let us make a small **team** – one object *responsible* for taking the decision; the two other *responsible* for the individual rate calculations.

AARHUS UNIVERSITET

The Cartoon

rate calculation request

Team leader



1. check clock

2.delegate to expert

Rate Policy Expert
Linear

Rate Policy Expert
Progressive

Interpretation

- Note:

Pay Station

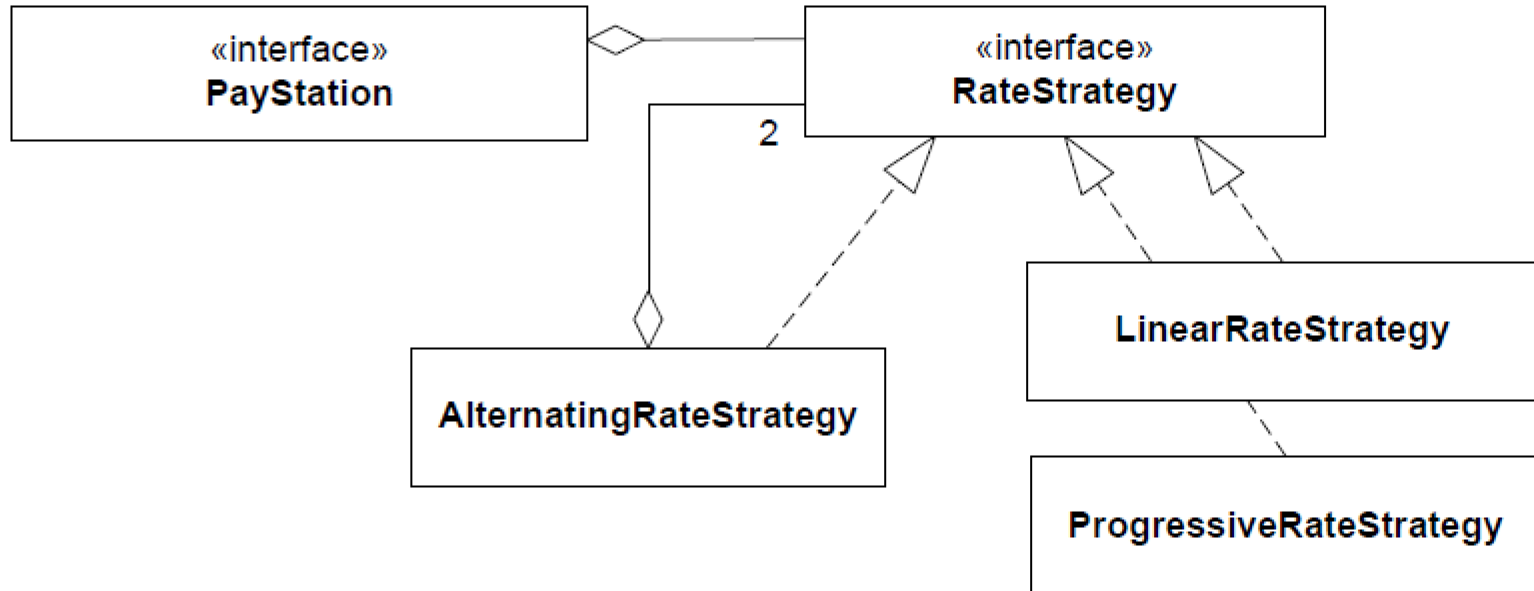
rate calculation request

Team leader

- From the Pay Station's viewpoint the behavior of the "team leader" *change according to the state of the clock!*

Clock State Define Behavior

- Reusing existing, well tested, classes...



Code view

In AlternatingRateStrategy:

```
public int calculateTime( int amount ) {
  if ( isWeekend() ) {
    currentState = weekendStrategy;
  } else {
    currentState = weekdayStrategy;
  }
  return currentState.calculateTime( amount );
}
```

1. check clock, *choose* expert to use
2. *delegate* to expert

In AlternatingRateStrategy: Construction

```
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;
  }
```

Analysis

- Consequence:
  - 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...

Roles revisited

- This once again emphasizes the importance of
  - ③ Encapsulate what varies: the rate policy
  - ① Define well-defined *responsibilities* by interfaces
  - ① Only let objects communicate using the interfaces
    - Then the respective *roles* (pay station / rate strategy) can be played by many difference concrete objects
    - And each object is free to implement the responsibilities of the roles as it sees fit **– like our new ‘team leader’ that does little on his own!**
  - ② **also to let most of the dirty job be done by others** ☺
    - Delegate concrete calculations to the two rate specialists

The State Pattern

AARHUS UNIVERSITET

Analysis

- Yet another application of 3-1-2
  - (but note that the argumentation this time was heavily focused on the ② aspect: composing behavior by delegating to partial behavior)

- Rephrasing what the Gammatown pay system does:
  - *The rate policy algorithm alters its behavior according to the state of the system clock*

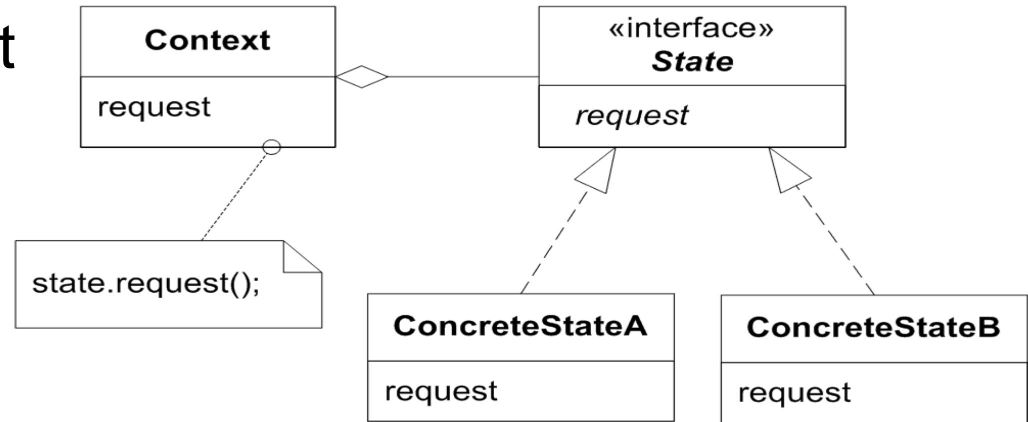
State Pattern

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

AARHUS UNIVERSITET

Roles

Context delegate to it current state object



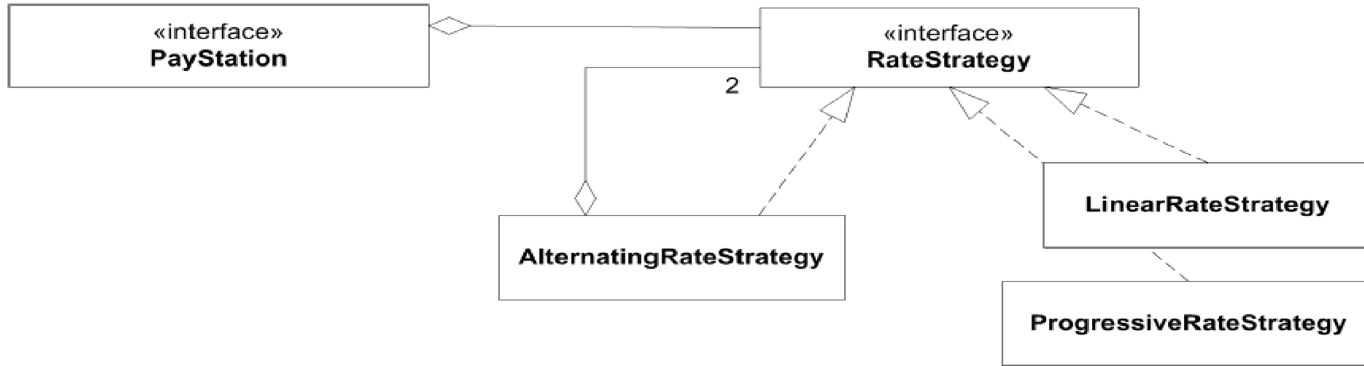
- **State** specifies responsibilities of the behavior that varies according to state
- **ConcreteState** defines state specific behavior

- State changes?
  - May be defined either in Context or in ConcreteState set
  - That who defines it is less reusable

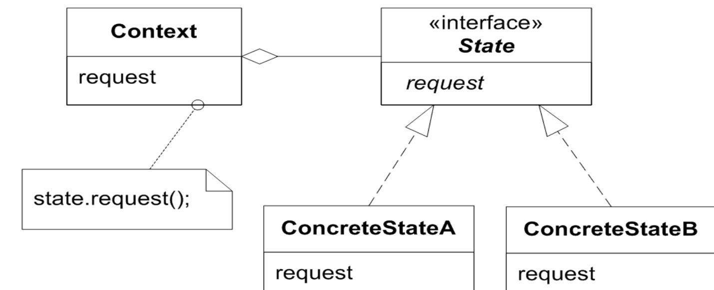
AARHUS UNIVERSITET

Exercise

- Which object/interface fulfil which role in the pay station?



- Who is responsible for state changes?



Benefits/Liabilities of State

- General
  - State specific behavior 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
- Compare common state machines:
  - case INIT_STATE:
  - case DIE_ROLL_STATE:
  - case MOVE_CHECKERS_STATE:

Examples

- All state machines can be modelled by the state pattern
  - and looking for them there are a lot
  - TCP Socket connection state
  - any game has a state machine
  - Protocols
  - etc...

Example: Turnstile

```
public class TurnstileImpl implements Turnstile {
  State
    lockedState = new LockedState(this),
    unlockedState = new UnlockedState(this),
    state = lockedState;
  public void coin() { state.coin(); }
  public void pass() { state.pass(); }

  public static void main(String[] args) {
    System.out.println( "Demo of turnstile state pattern" );
    Turnstile turnstile = new TurnstileImpl();
    turnstile.coin();
    turnstile.pass();
    turnstile.pass();
    turnstile.coin();
    turnstile.coin();
  }
}

abstract class State implements Turnstile {
  protected TurnstileImpl turnstile;
  public State(TurnstileImpl ts) { turnstile = ts; }
}

class LockedState extends State {
  public LockedState(TurnstileImpl ts) { super(ts); }
  public void coin() {
    System.out.println( "Locked state: Coin accepted");
    turnstile.state = turnstile.unlockedState;
  }
  public void pass() {
    System.out.println( "Locked state: Passenger pass: SOUND ALARM");
  }
}
class UnlockedState extends State {
  public UnlockedState(TurnstileImpl ts) { super(ts); }
  public void coin() {
    System.out.println( "Unlocked state: Coin entered: RETURN IT");
  }
  public void pass() {
    System.out.println( "Unlocked state: Passenger pass");
    turnstile.state = turnstile.lockedState;
  }
}
```

Summary

- New requirement
  - a case that *screams* for reusing existing and well-tested production code
  - cumbersome to utilize the reuse potential especially in the sub-classing case (deeper discussion in the book)
  - but handled elegantly by compositional design
    - think in terms of teams of objects playing different roles
  - I derived the State pattern
    - more general pattern handling state machines well