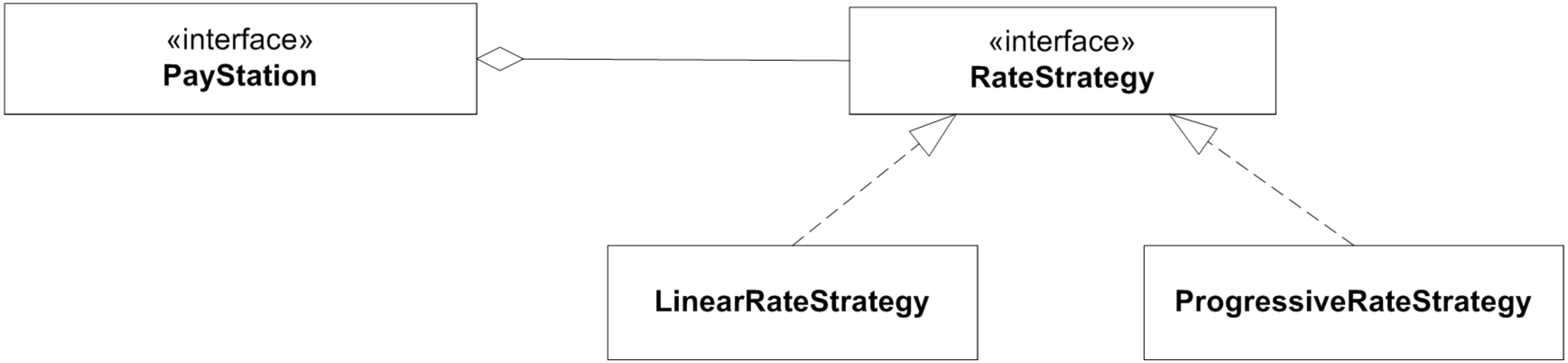
Deriving State…

...and an example of combining behaviour

New requirement

Gammatown County wants:

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



Exercise

“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 == alpha) {} else if (town == beta) {} else if (town == gamma) {}

Model 3:

- Polymorphic – but ???

Model 4:

- Compositional – but how?

But…

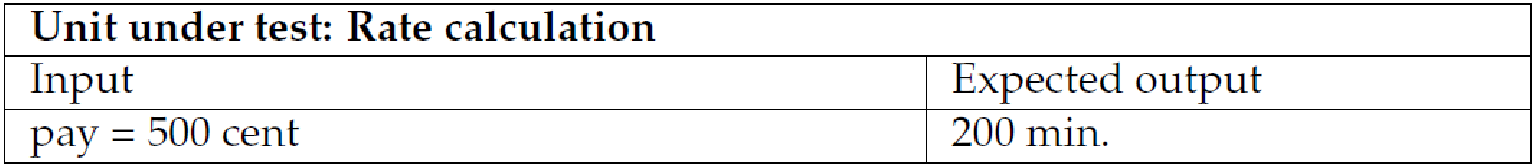
I will return to the analysis shortly, but first…

I have a problem!

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

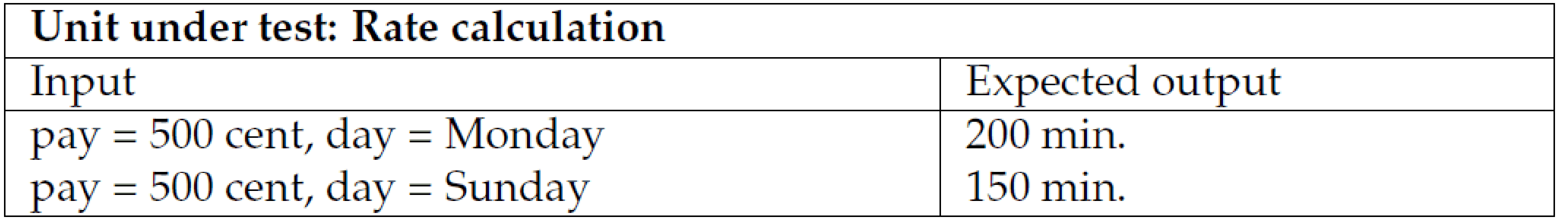
Tricky Requirement

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 ☹**

Solutions?

So – what to do?

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

I will return to this problem set soon…

Polymorphic Solutions to the GammaTown Challenge

Premise

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

That is, forget the nice Strategy based solution we did last time ☺

Revisited

How did it look?

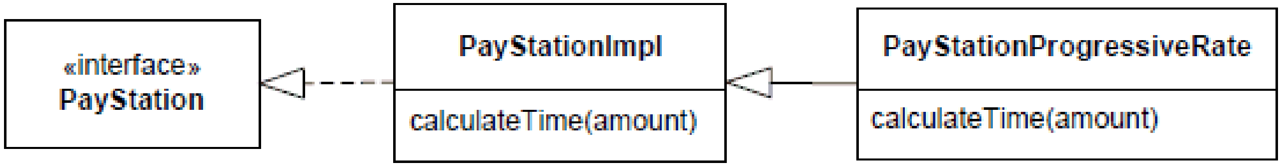
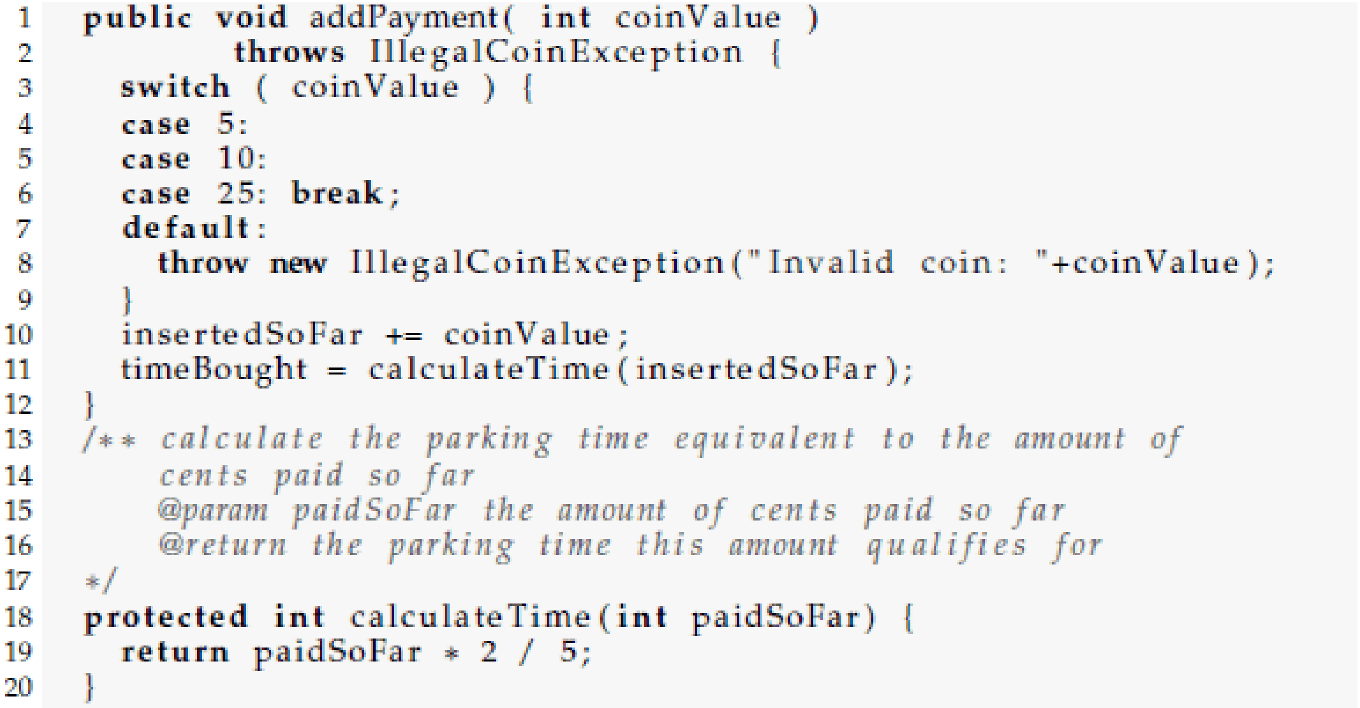


Figure 7.2: Subclassing proposal for new rate calculation.



```
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 = calculateTime(insertedSoFar);
}
/** calculate the parking time equivalent to the amount of
    cents paid so far
    @param paidSoFar the amount of cents paid so far
    @return the parking time this amount qualifies for
*/
protected int calculateTime(int paidSoFar) {
  return paidSoFar * 2 / 5;
}
```

So...

Given this:

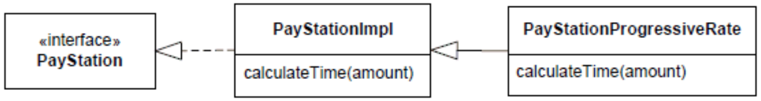
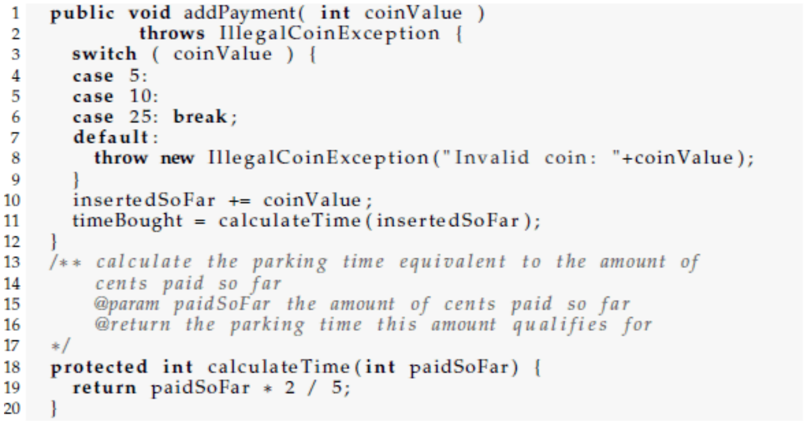


Figure 7.2: Subclassing proposal for new rate calculation.



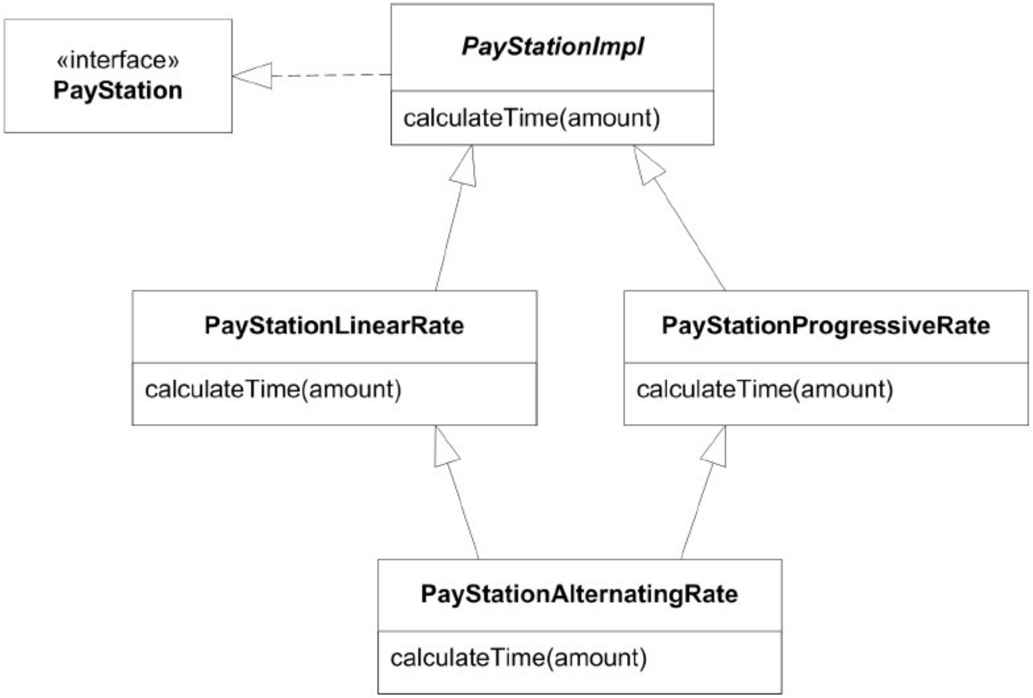
```
public void addPayment( int coinValue )
        throws IllegalCoinException {
  switch ( coinValue ) {
  case 5:
  case 10:
  case 25: break;
  default:
    throw new IllegalCoinException("Invalid coin: "+coinValue);
  }
  insertedSoFar += coinValue;
  timeBought = calculateTime(insertedSoFar);
}
/** calculate the parking time equivalent to the amount of
    cents paid so far
    @param paidSoFar the amount of cents paid so far
    @return the parking time this amount qualifies for
*/
protected int calculateTime(int paidSoFar) {
  return paidSoFar * 2 / 5;
}
```

How to make a subclass that *reuse the calculateTime implementations of two different classes in the class hierarchy?*

Actually, we can do this in quite a number of ways

Model 3a: Multiple Inheritance

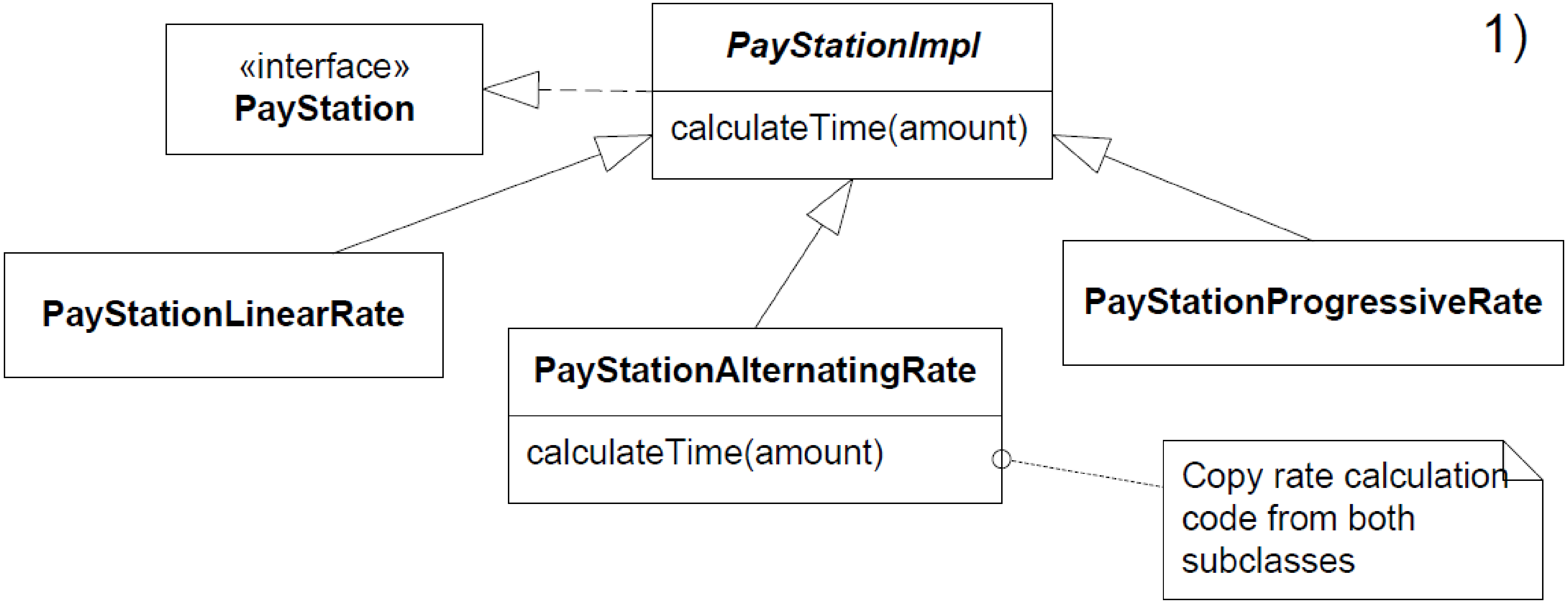
Subclass and override!



Could work in C++, but not Java or C#;

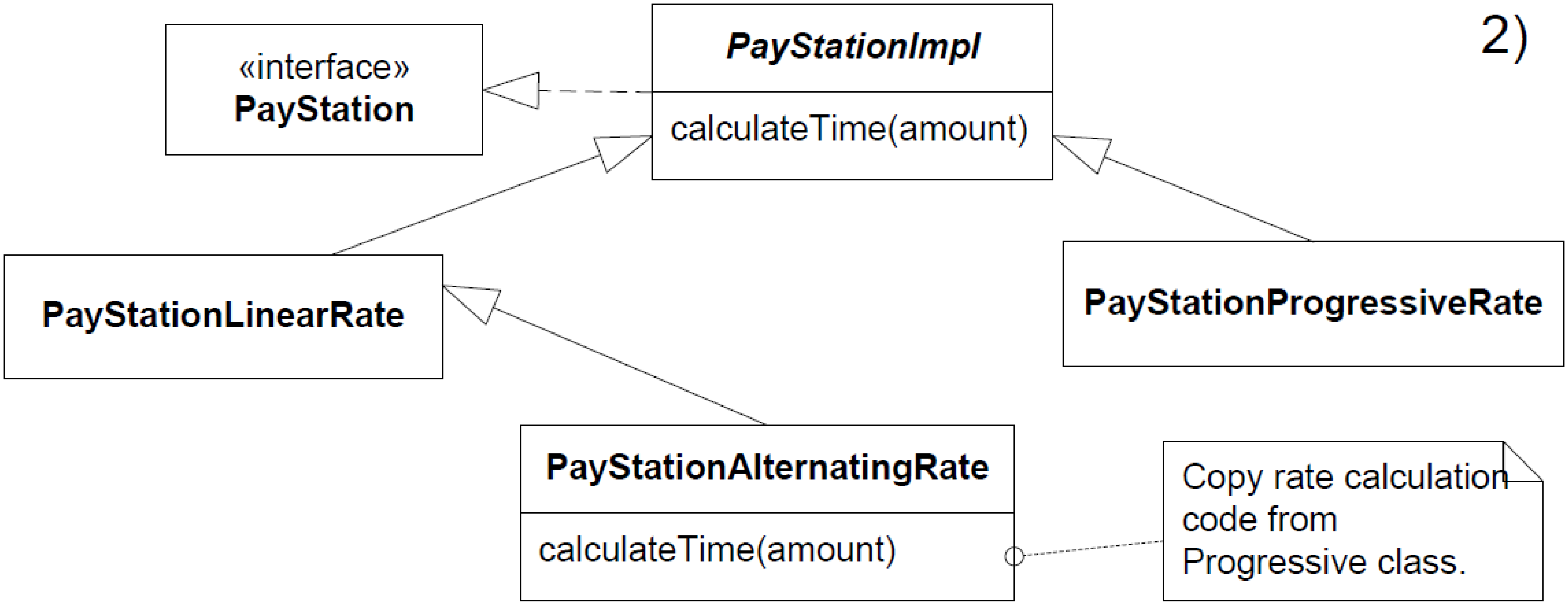
– experience with join-hierarchies in C++ are bad ☹

Model 3b: Direct Subclass



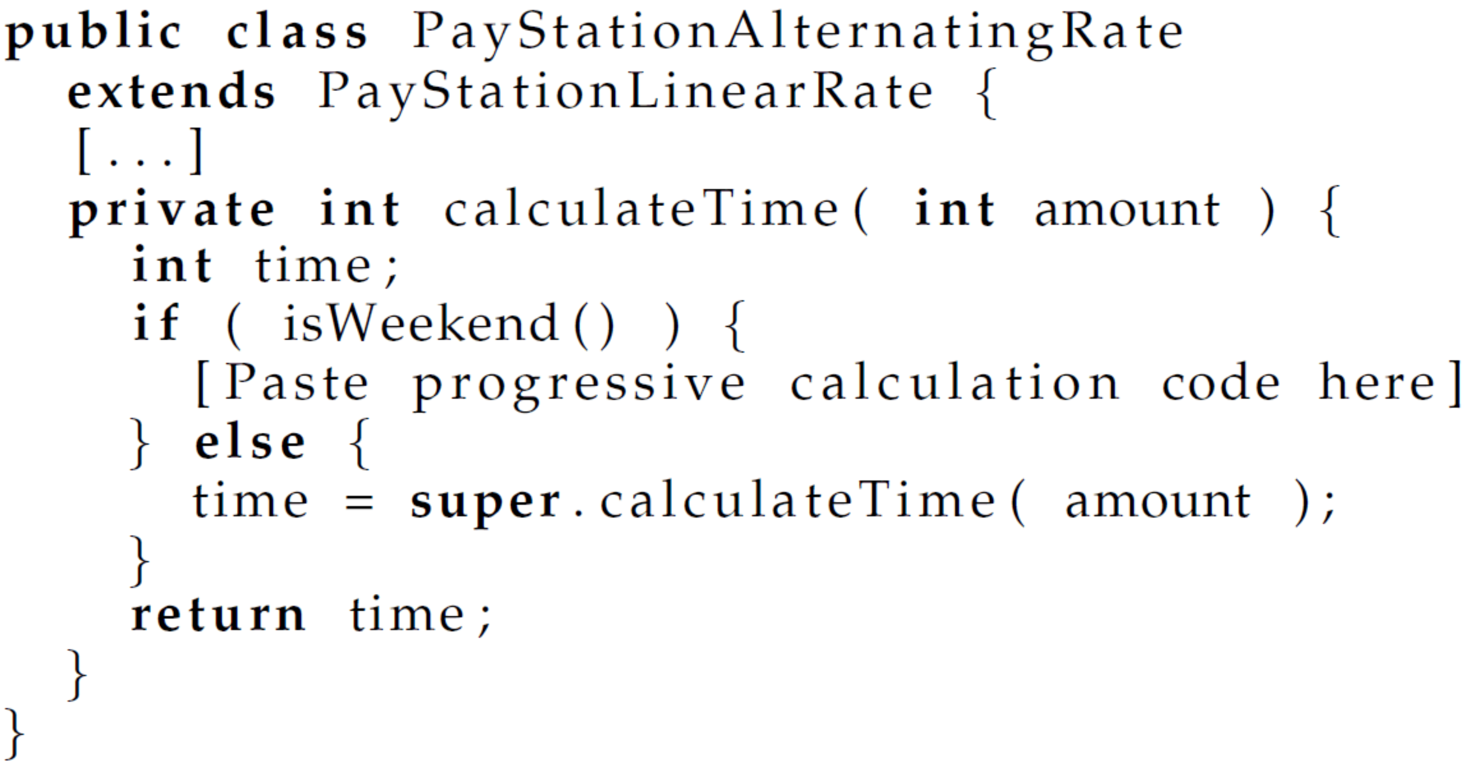
Cut code from liniear and progressive, paste into alternating

Model 3c:Sub-sub Class


2)
«interface»
PayStation
PayStationImpl
calculateTime(amount)
PayStationLinearRate
PayStationProgressiveRate
PayStationAlternatingRate
calculateTime(amount)
Copy rate calculation code from Progressive class.

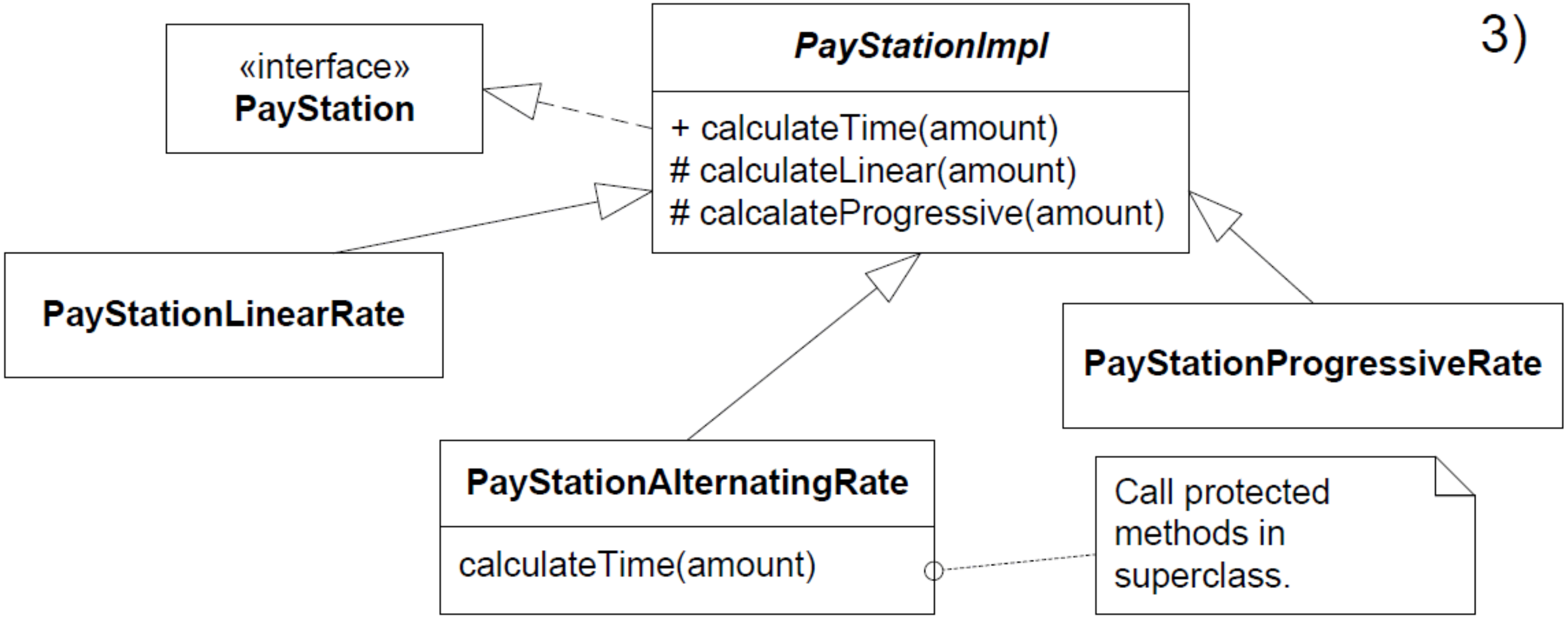
Cut code from progressive, paste into alternating

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

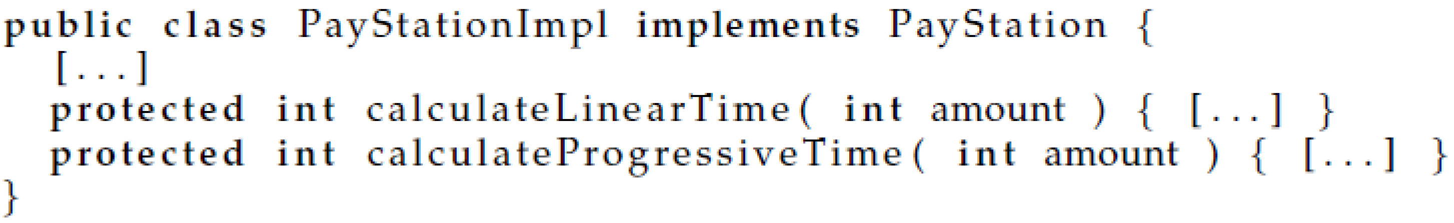

3)
«interface»
PayStation
PayStationImpl
+ calculateTime(amount)
# calculateLinear(amount)
# calcalateProgressive(amount)
PayStationLinearRate
PayStationProgressiveRate
PayStationAlternatingRate
calculateTime(amount)
Call protected methods in superclass.

Make protected calculation methods in abstract PayStationImpl, and call these from Alternating

- This is a classic solution often seen in practice

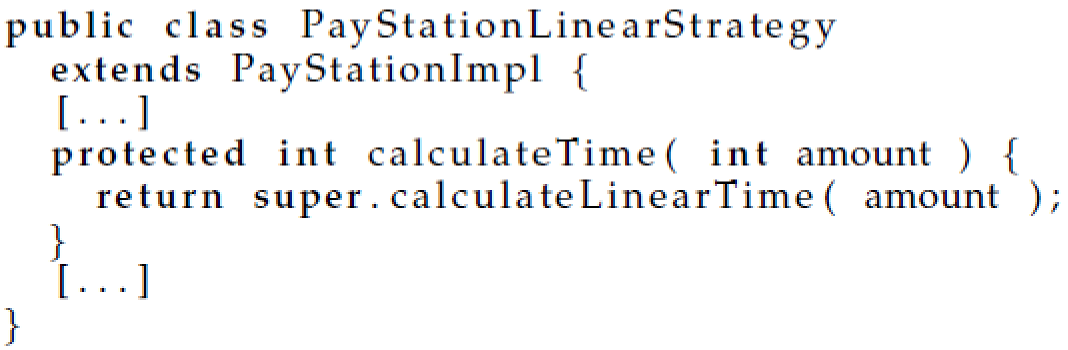
Code view

The super class



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

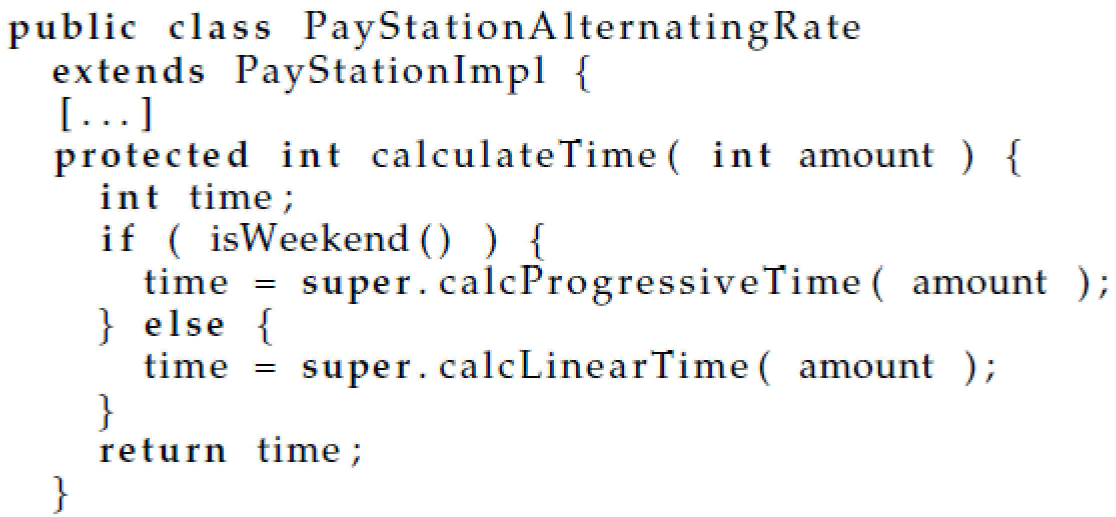
Alpha



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

Code view

Gamma



```
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

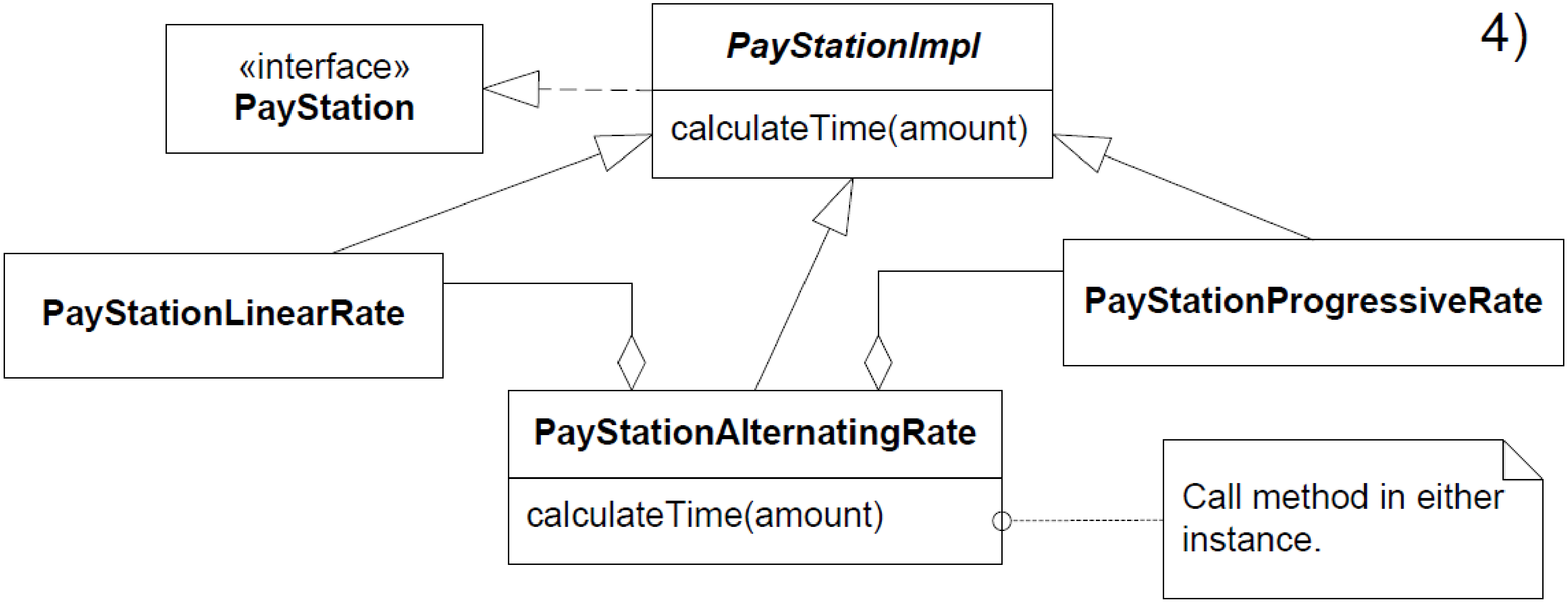
- – No code duplication
- – Exercise: what are the liabilities?

Liabilities

Superclass maintainability

- 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?

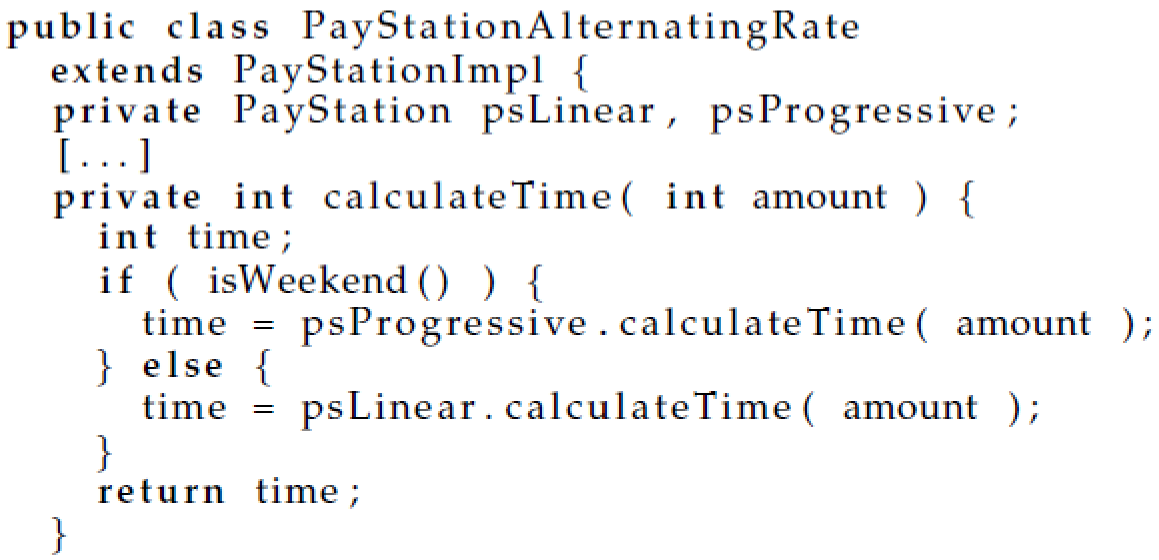
Model 3e: Stations in Stations



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;
  }
```

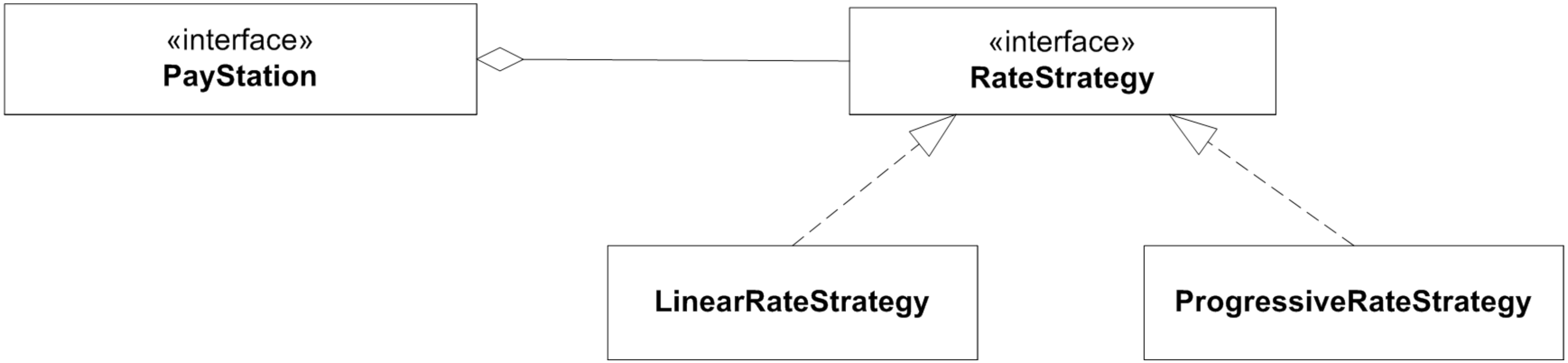
Exercise: Benefits and liabilities?

Compositional Variants

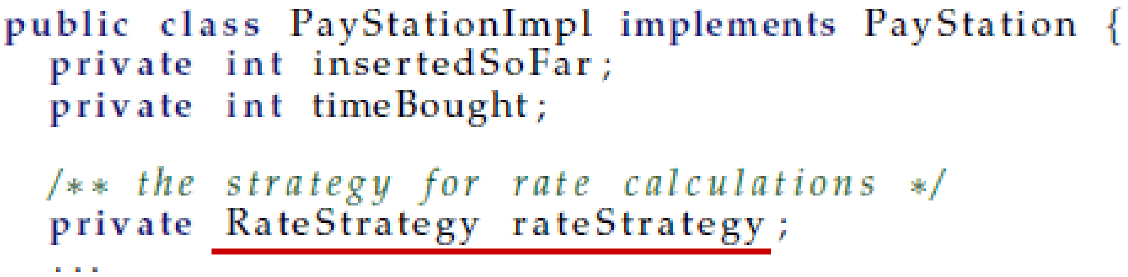
Premise

Now, please reset your minds!

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

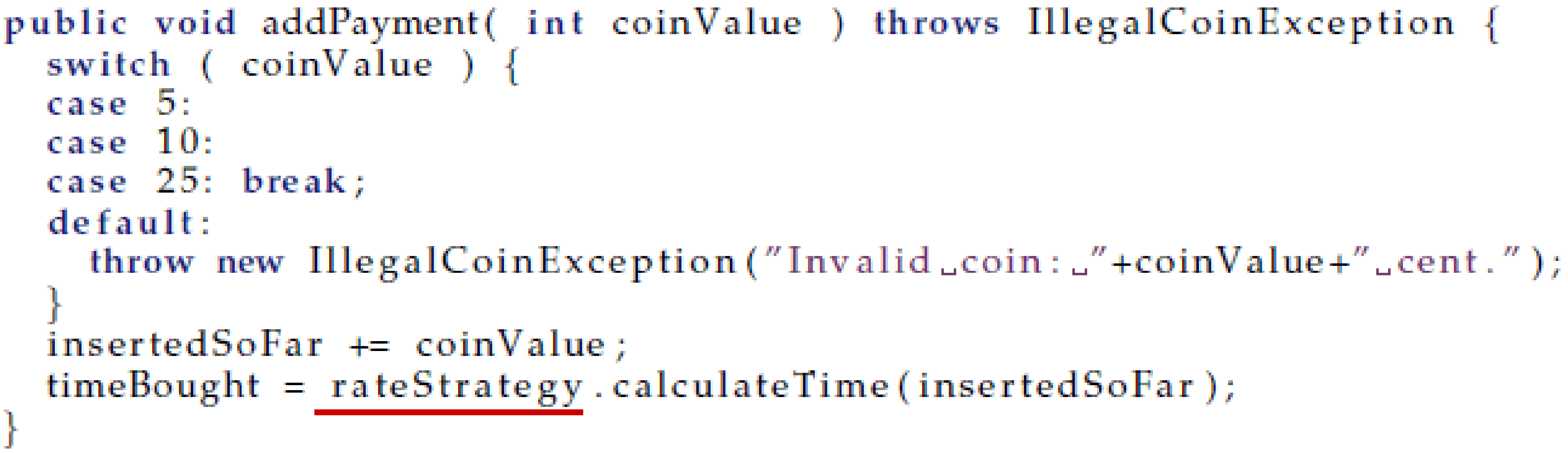


Code View (Java)



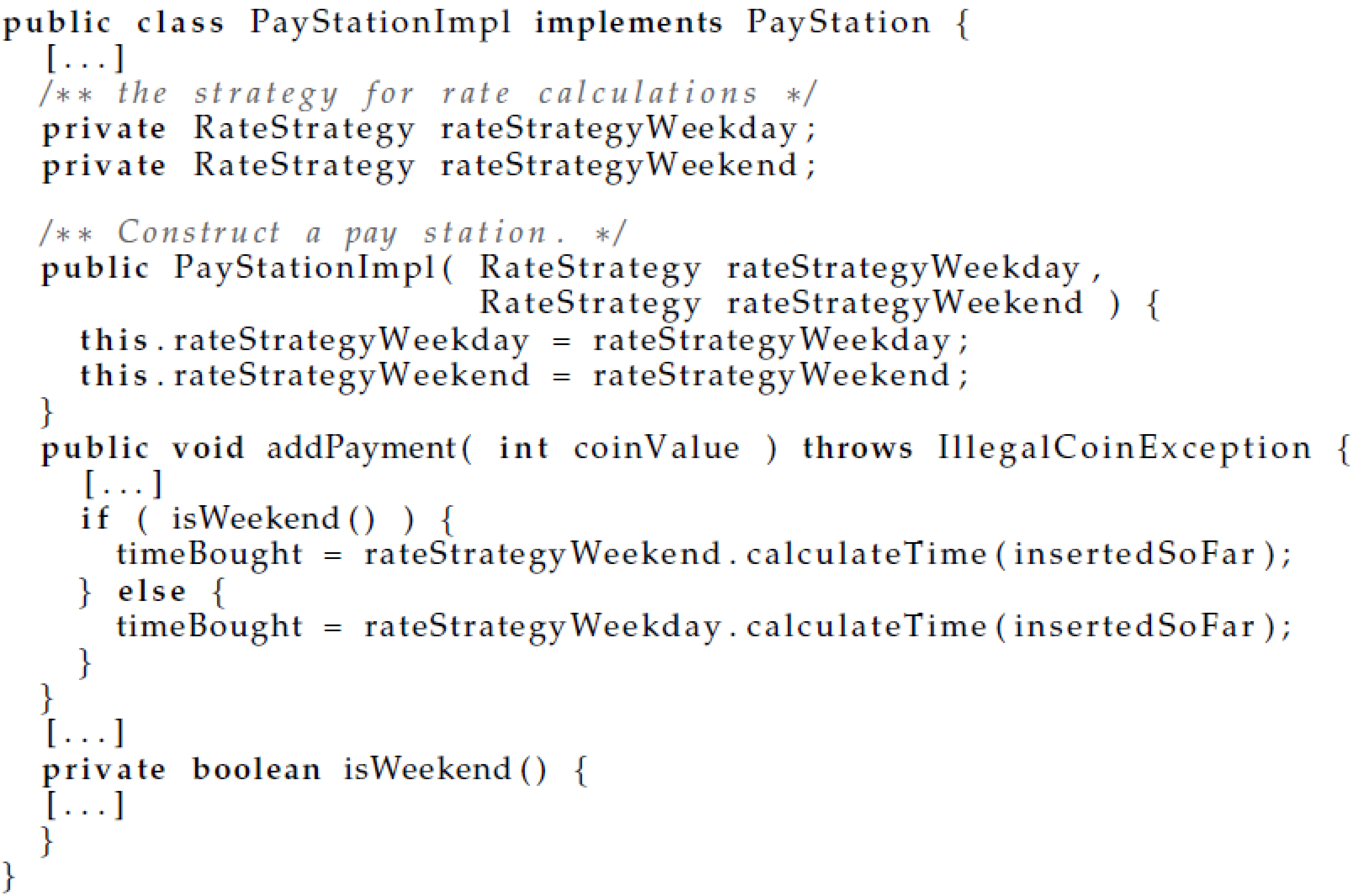
```
public class PayStationImpl implements PayStation {
  private int insertedSoFar;
  private int timeBought;

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

and modify the addPayment method:



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

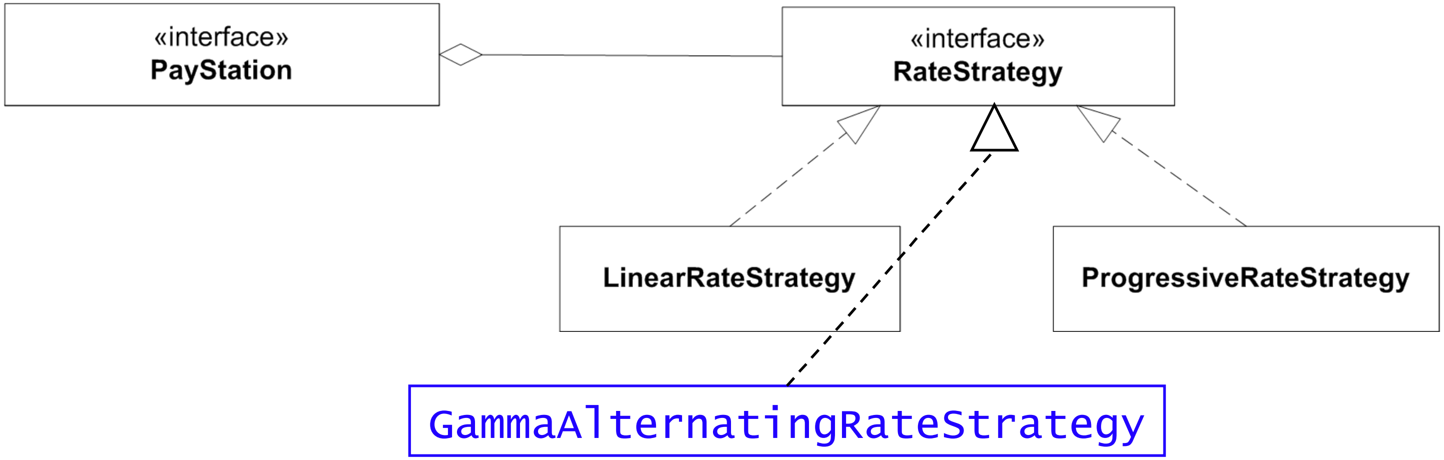
- Code change in the constructor
- Constructor has become really weird for alpha and beta

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 analyzabilty*
  - leads to duplicated code, and bugs difficult to track.

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.

… on to a nice compositional solution: State pattern

Compositional Idea

③ *I identify some behaviour that varies.*

- The rate calculation behaviour is what must vary for Gammatown and this we have already identified.

① *I state 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.

② *I 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.

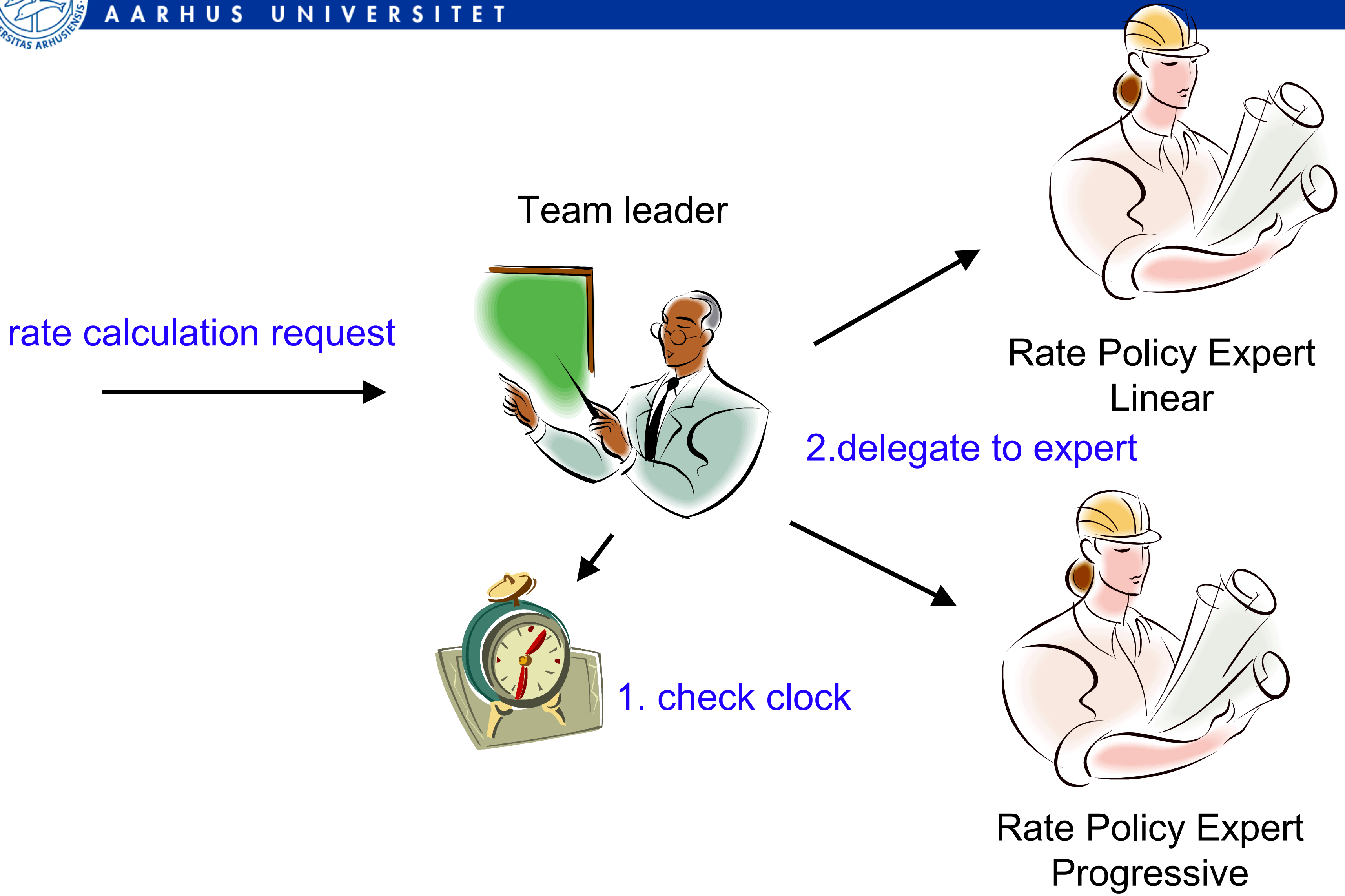
Model 4

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.

The Cartoon



Team leader

rate calculation request

1. check clock

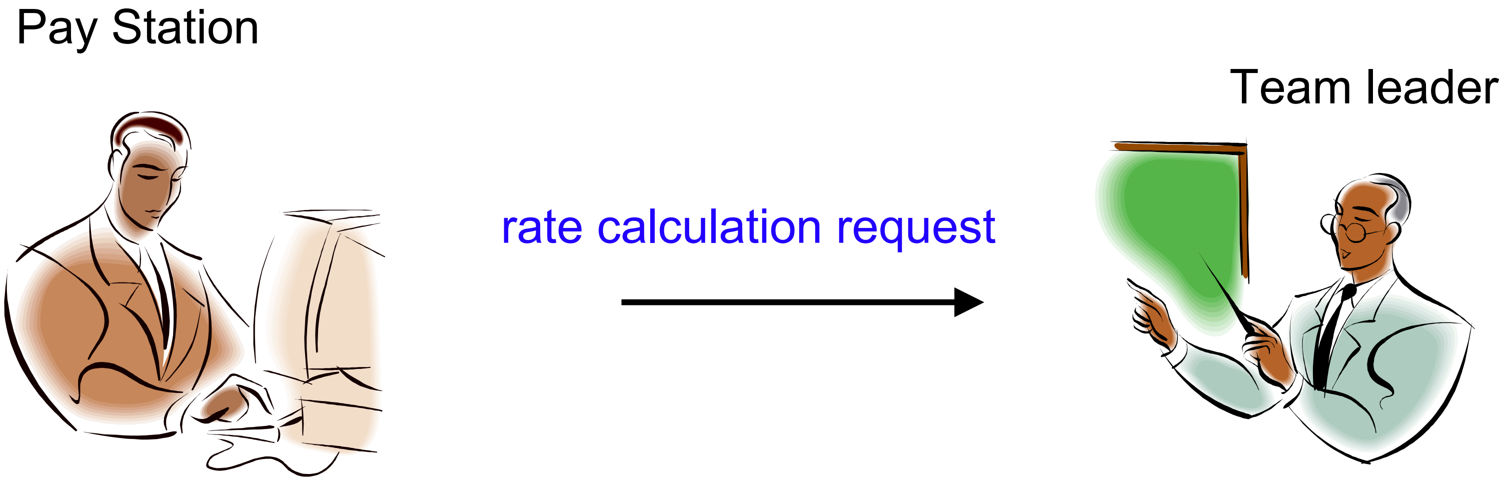
2.delegate to expert

Rate Policy Expert Linear

Rate Policy Expert Progressive

Interpretation

Note:



Pay Station

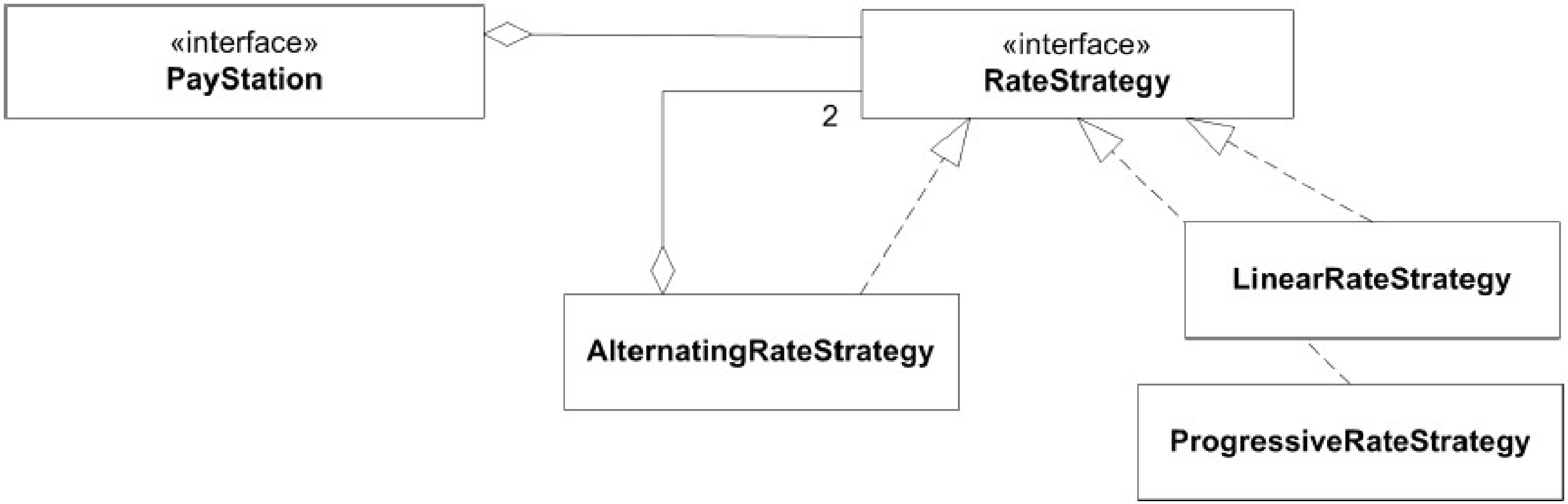
rate calculation request

Team leader

From the Pay Station’s viewpoint the behaviour of the ”team leader” *change according to the state of the clock!*

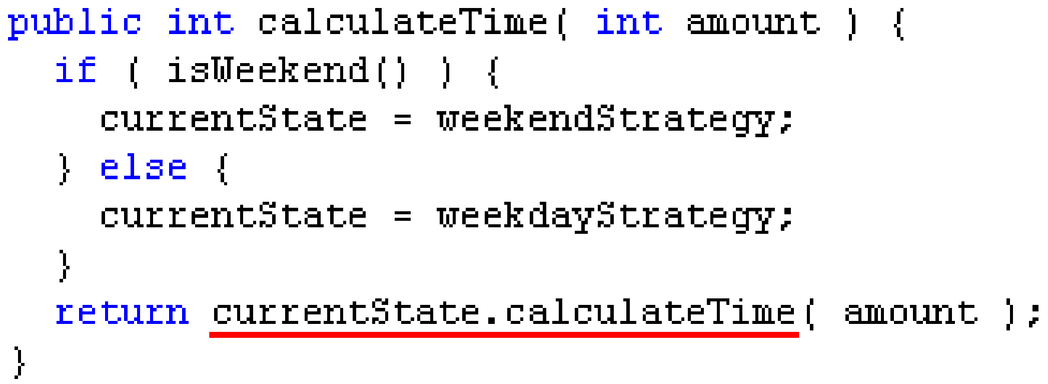
Clock State Define Behavior

Reusing existing, well tested, classes...



Code view

In AlternatingRateStrategy:

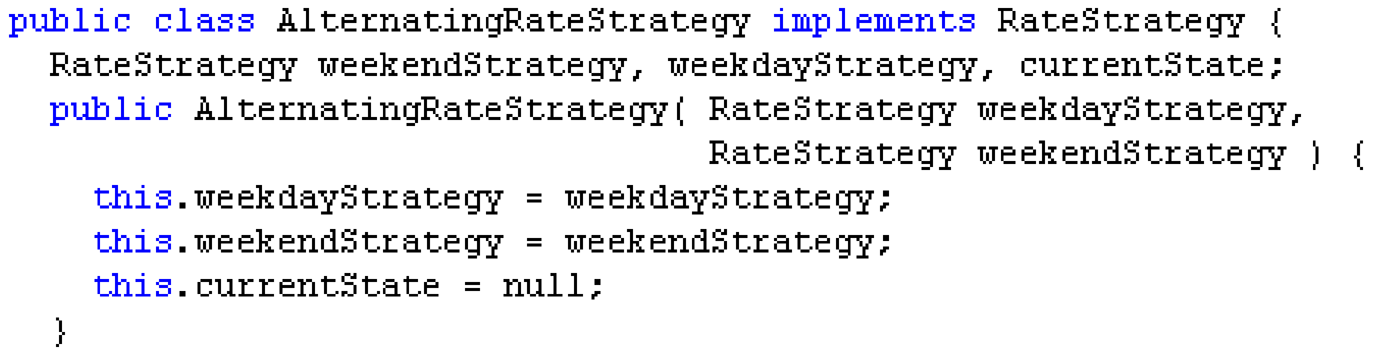


```
public int calculateTime( int amount ) {
  if ( isWeekend() ) {
    currentState = weekendStrategy;
  } else {
    currentState = weekdayStrategy;
  }
  return currentState.calculateTime( amount );
}
```

1. check clock

2.delegate to expert

In AlternatingRateStrategy: Construction



```
public class AlternatingRateStrategy implements RateStrategy {
  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 ☺**

The State Pattern

Analysis

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

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

State Pattern

State pattern intent

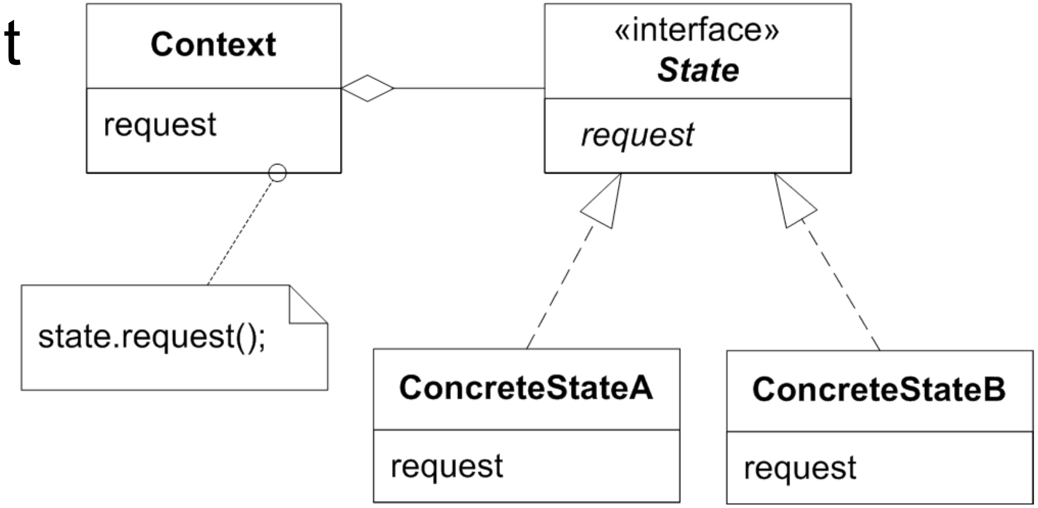
- **Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.**
- *The rate policy algorithm alters its behaviour according to the state of the system clock*
- Seen from the PayStationImpl the AlternatingRateStrategy object appears to change class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

Roles

Context delegate to it current state object

State specifies responsibilities of the behaviour that varies according to state

ConcreteState defines state specific behaviour

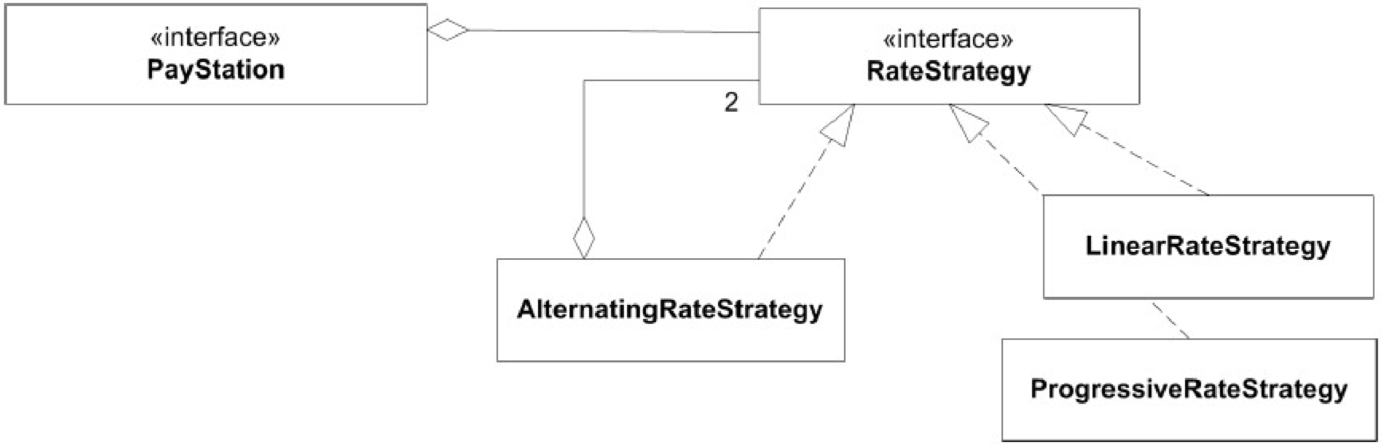


State changes?

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

Exercise

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



Who is responsible for state changes?

Benefits/Liabilities of State

General

- 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

Compare common statemachines:

- case INIT_STATE:
- case DIE_ROLL_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...

Summary

New requirement

- a case that *screams* for reusing existing and well-tested production code
- cumbersome to utilize the reuse potential especially in the subclassing 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