# Lecture 10

ECE 1145: Software Construction and Evolution

## Variability Management

## State Pattern

## (CH 11)

# Announcements

- Relevant Exercises: **11.4** 11.5
- **Code Review 1**
  - Groups on Canvas
  - Provide other team with your Iteration 3 (Release3) **by the start of class on Wednesday** (complete the Code Swap 1 assignment to confirm)
    - As a zip file or access to GitHub repo
  - Compete code review template and report
    - Time in class Wednesday
- **Midterm Oct. 18 (take-home) – 2 weeks from Today**
  - Midterm review on Wednesday Oct. 13
  - Access and submit via Canvas
  - ~24 hour window
  - Lectures 1 – 9, project iterations 1 – 3 and code review
- Iteration 4 (due Oct. 17) will be code quality improvements

# Questions for Today

How do we combine existing solutions without duplicating code?

# A New Customer!

We have been contacted by GammaTown! They want “almost the same” pay station, but with different rate structures during weekdays and weekends:

- Weekdays: linear rate (like AlphaTown)
- Weekends: progressive rate (like BetaTown)

# A New Customer!

We have been contacted by GammaTown! They want “almost the same” pay station, but with different rate structures during weekdays and weekends:

- Weekdays: linear rate (like AlphaTown)
- Weekends: progressive rate (like BetaTown)

The two rate structures are already implemented, we just need a way to **select** based on the day.

# A New Customer!

Recall our models:

# A New Customer!

Recall our models:

- **Source tree copy:** copy the existing source code, name it GammaTown and implement the alternating rate there

# A New Customer!

Recall our models:

- **Source tree copy:** copy the existing source code, name it GammaTown and implement the alternating rate there
- **Parametric:** add a new GammaTown clause to all switches in the pay station code

# A New Customer!

Recall our models:

- **Source tree copy:** copy the existing source code, name it GammaTown and implement the alternating rate there
- **Parametric:** add a new GammaTown clause to all switches in the pay station code
- **Polymorphic:** make a subclass of PayStation to handle GammaTown's alternating rate

# A New Customer!

Recall our models:

- **Source tree copy:** copy the existing source code, name it GammaTown and implement the alternating rate there
- **Parametric:** add a new GammaTown clause to all switches in the pay station code
- **Polymorphic:** make a subclass of PayStation to handle GammaTown's alternating rate
- **Compositional (with some if's):** use compositional design but introduce a switch in the pay station to select the rate strategy depending on the day of the week

# A New Customer!

Recall our models:

- **Source tree copy:** copy the existing source code, name it GammaTown and implement the alternating rate there
- **Parametric:** add a new GammaTown clause to all switches in the pay station code
- **Polymorphic:** make a subclass of PayStation to handle GammaTown's alternating rate
- **Compositional (with some if's):** use compositional design but introduce a switch in the pay station to select the rate strategy depending on the day of the week
- **Compositional:** compose required behavior, using collaborating objects

# A New Customer!

Recall our models:

- **Source tree copy:** copy the existing source code, name it GammaTown and implement the alternating rate there
- **Parametric:** add a new GammaTown clause to all switches in the pay station code
- **Polymorphic:** make a subclass of PayStation to handle GammaTown's alternating rate
- **Compositional (with some if's):** use compositional design but introduce a switch in the pay station to select the rate strategy depending on the day of the week
- **Compositional:** compose required behavior, using collaborating objects

# Polymorphic Solution Proposal

Use polymorphism to reuse the existing rate algorithms (assuming a polymorphic design for AlphaTown and BetaTown implementation)

# Polymorphic Solution Proposal

Use polymorphism to reuse the existing rate algorithms (assuming a polymorphic design for AlphaTown and BetaTown implementation)

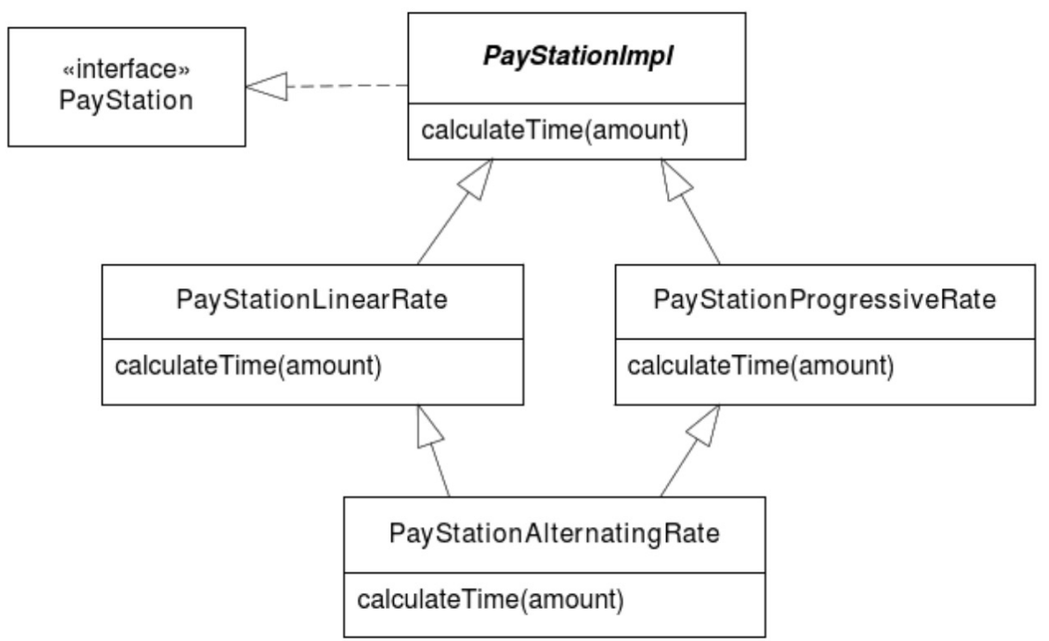



Figure 11.1: Multiple inheritance solution.

# Polymorphic Solution Proposal

Use polymorphism to reuse the existing rate algorithms (assuming a polymorphic design for AlphaTown and BetaTown implementation)

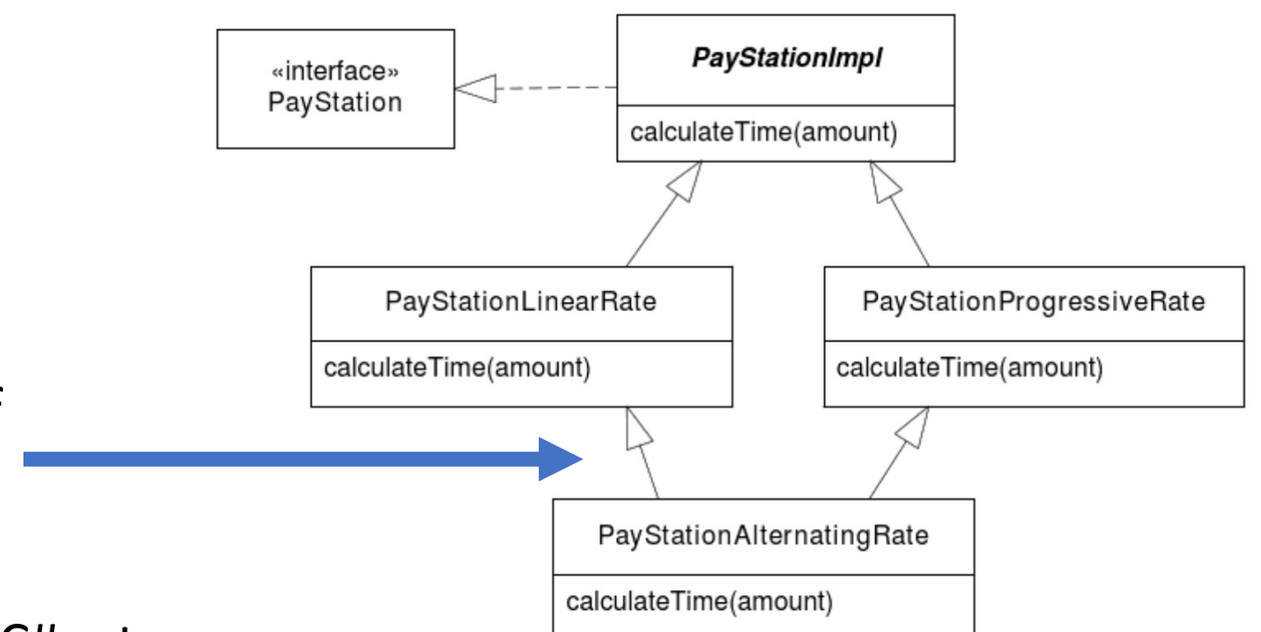



Figure 11.1: Multiple inheritance solution.

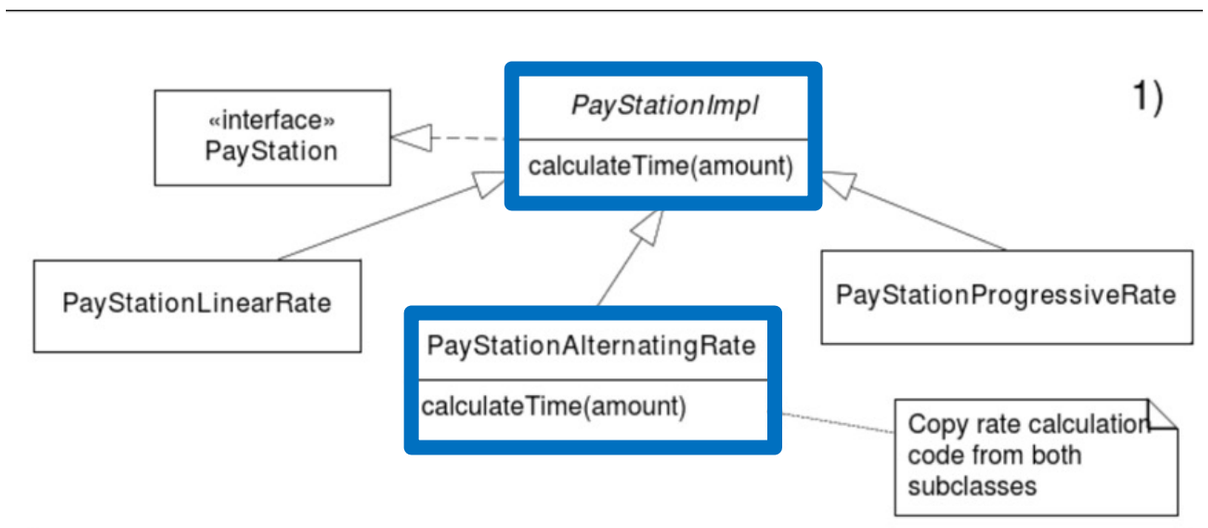
Multiple inheritance of implementation

- Supported in C++
- Not supported in Java, C#, etc.
- High risk of introducing defects!

# Polymorphic Solution Proposal

Other options:

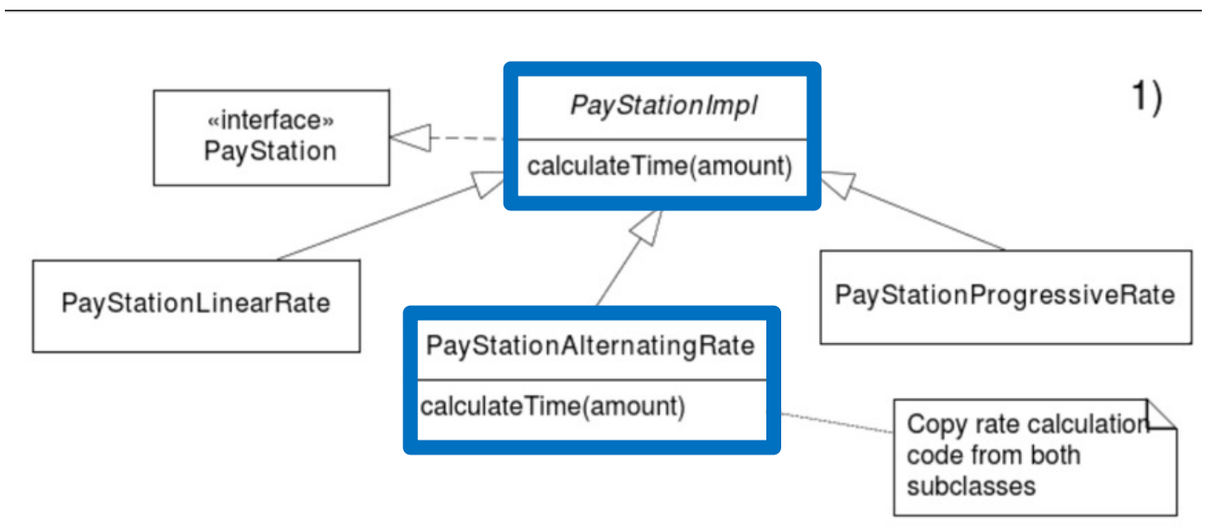
1. Direct subclass of PayStationImpl



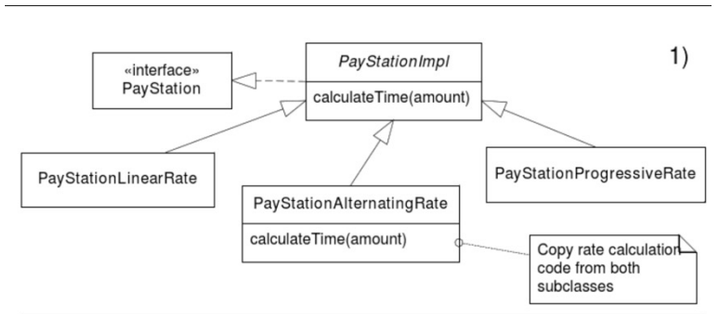
# Polymorphic Solution Proposal

Other options:

1. Direct subclass of PayStationImpl
   - Duplicated code

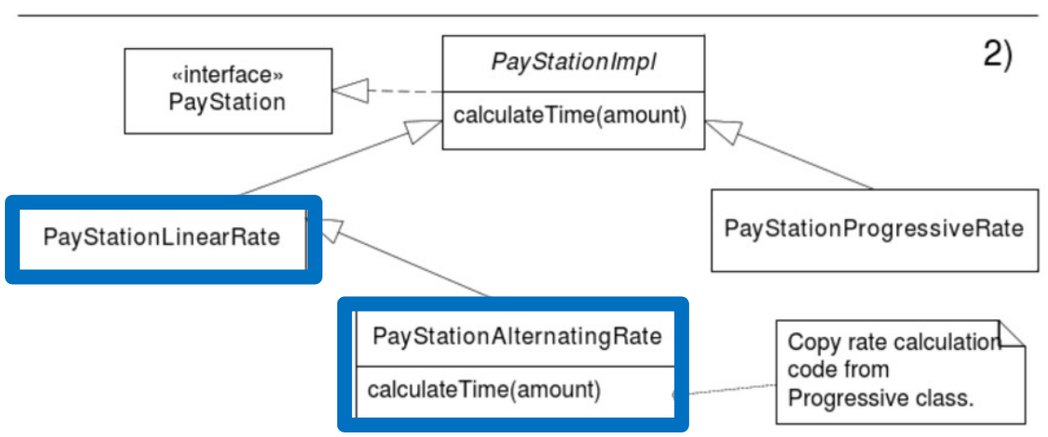


# Polymorphic Solution Proposal

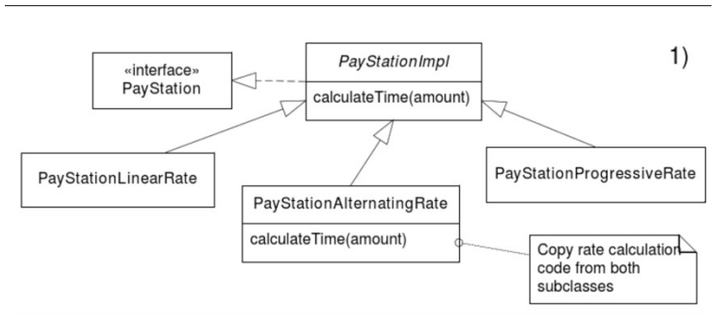



Other options:

1. Direct subclass of PayStationImpl
   - Duplicated code
2. Sub-subclass of PayStationLinearRate

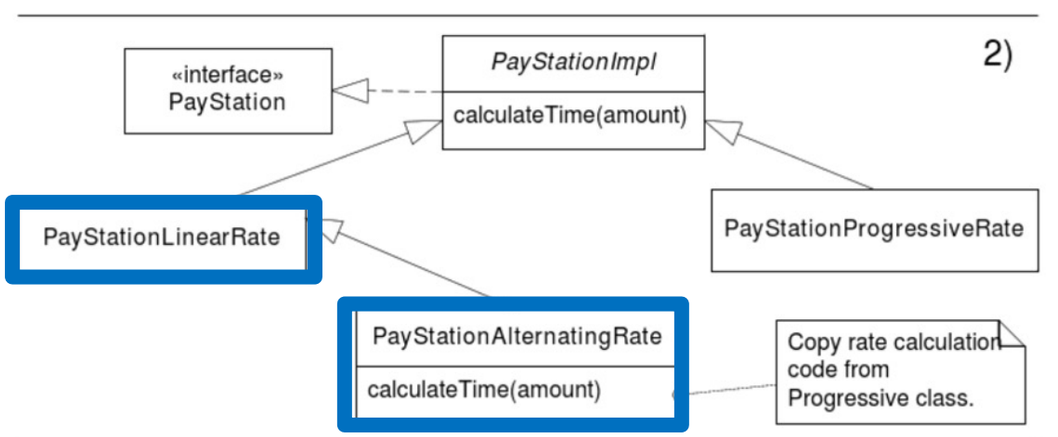


# Polymorphic Solution Proposal




Other options:

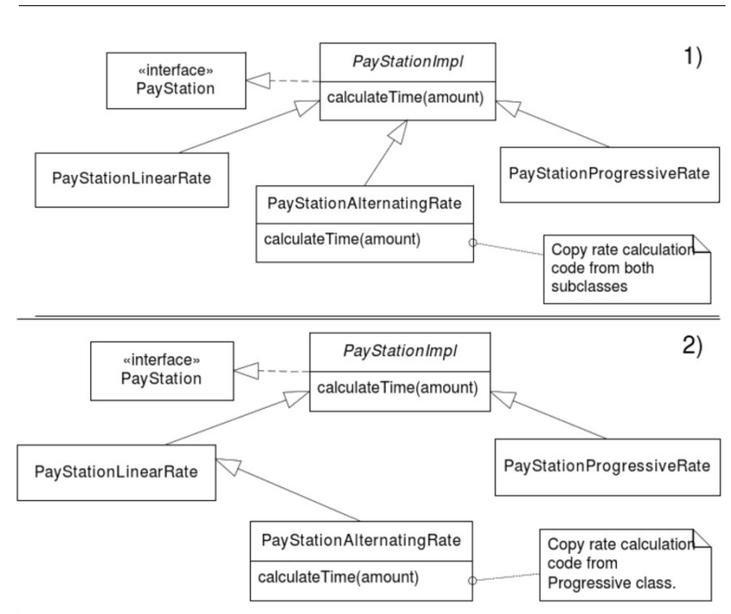
1. Direct subclass of PayStationImpl
   - Duplicated code
2. Sub-subclass of PayStationLinearRate
   - Avoids some duplication (but not all), and creates confusing asymmetry

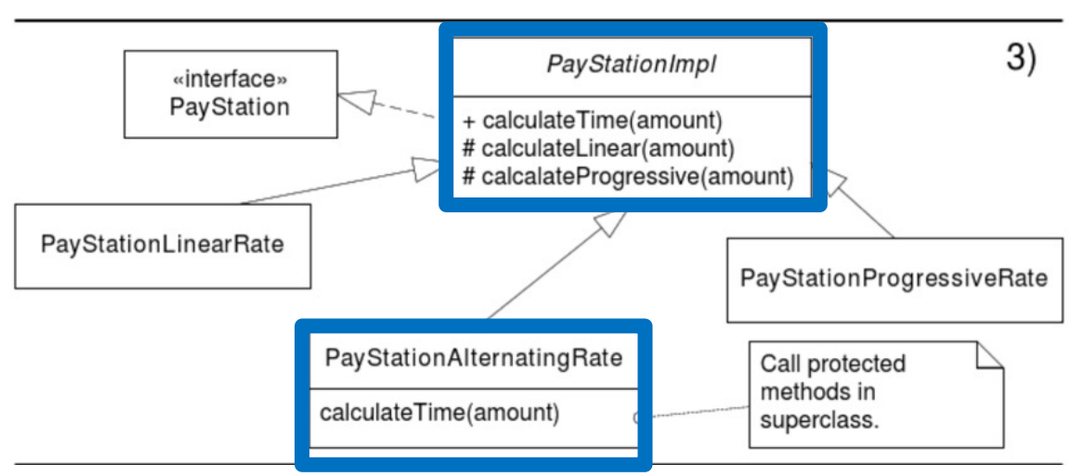


# Polymorphic Solution Proposal

Other options:

1. Direct subclass of PayStationImpl
   - Duplicated code
2. Sub-subclass of PayStationLinearRate
   - Avoids some duplication (but not all), and creates confusing asymmetry
3. Superclass rate calculation

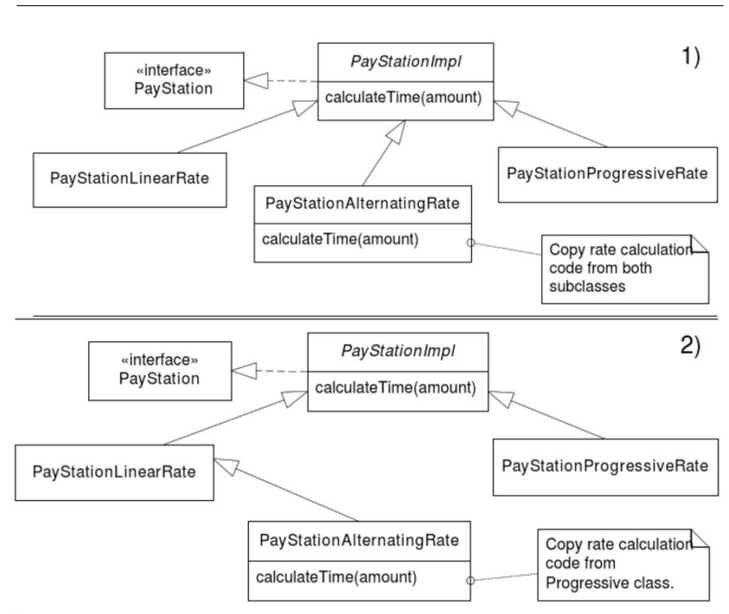


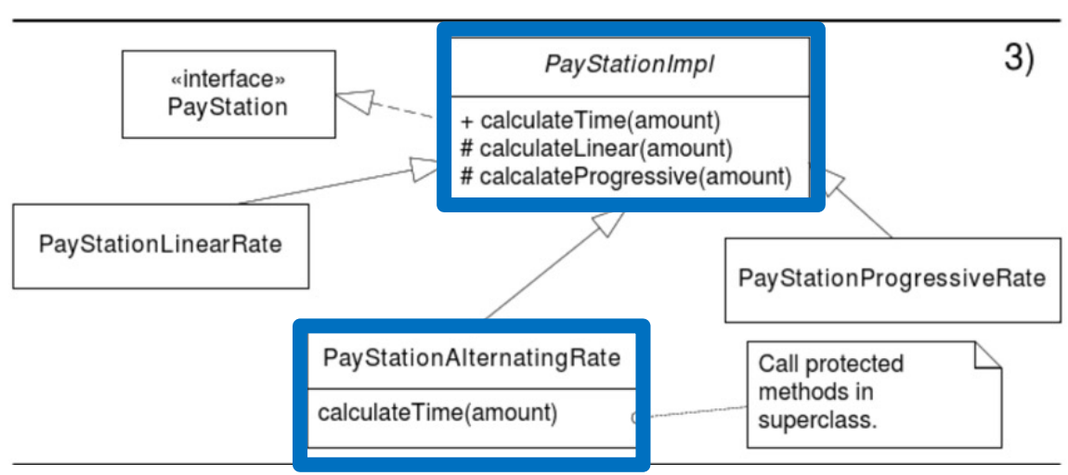



# Polymorphic Solution Proposal

Other options:

1. Direct subclass of PayStationImpl
   - Duplicated code
2. Sub-subclass of PayStationLinearRate
   - Avoids some duplication (but not all), and creates confusing asymmetry
3. Superclass rate calculation
   - Avoids duplication, but does not scale and risks change by modification for future rate policies

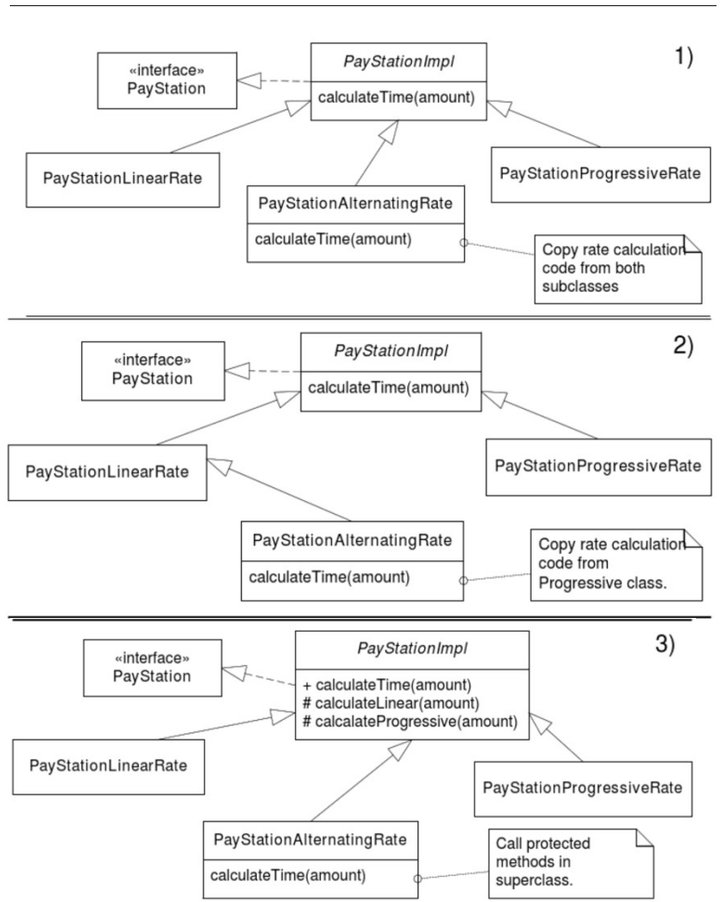


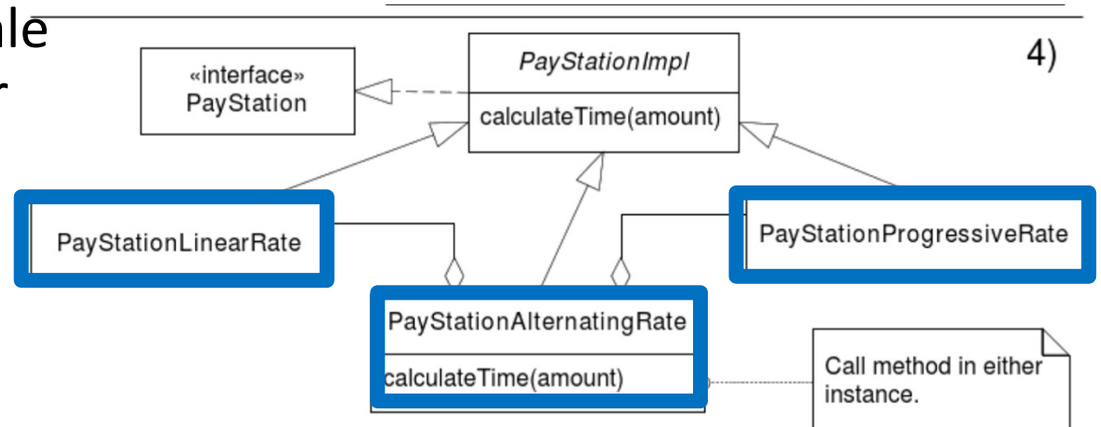



# Polymorphic Solution Proposal

Other options:

1. Direct subclass of PayStationImpl
   - Duplicated code
2. Sub-subclass of PayStationLinearRate
   - Avoids some duplication (but not all), and creates confusing asymmetry
3. Superclass rate calculation
   - Avoids duplication, but does not scale and risks change by modification for future rate policies
4. Pay stations within pay station

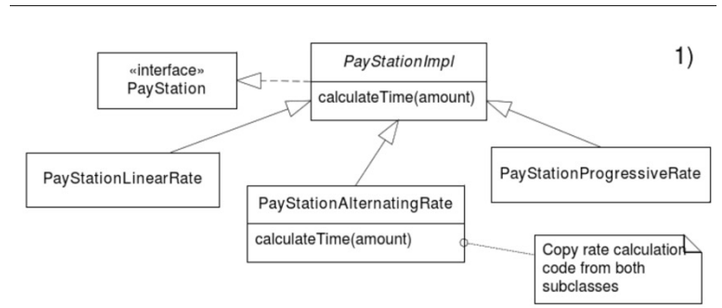


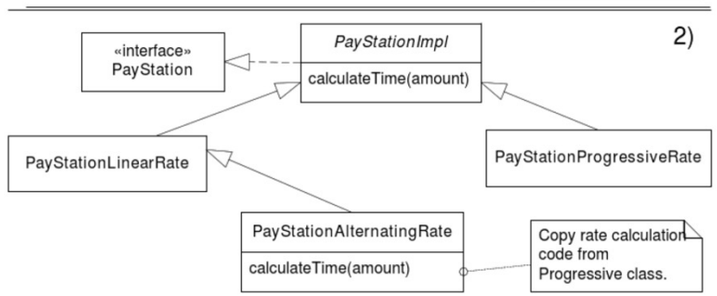



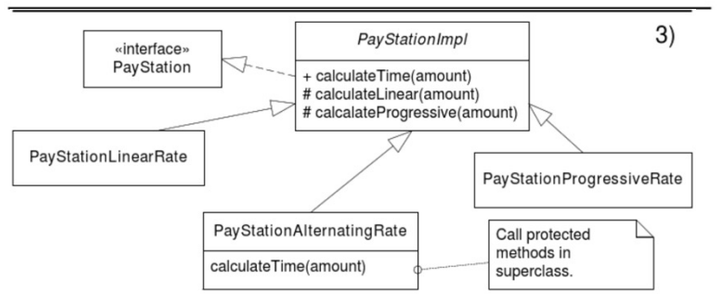
# Polymorphic Solution Proposal

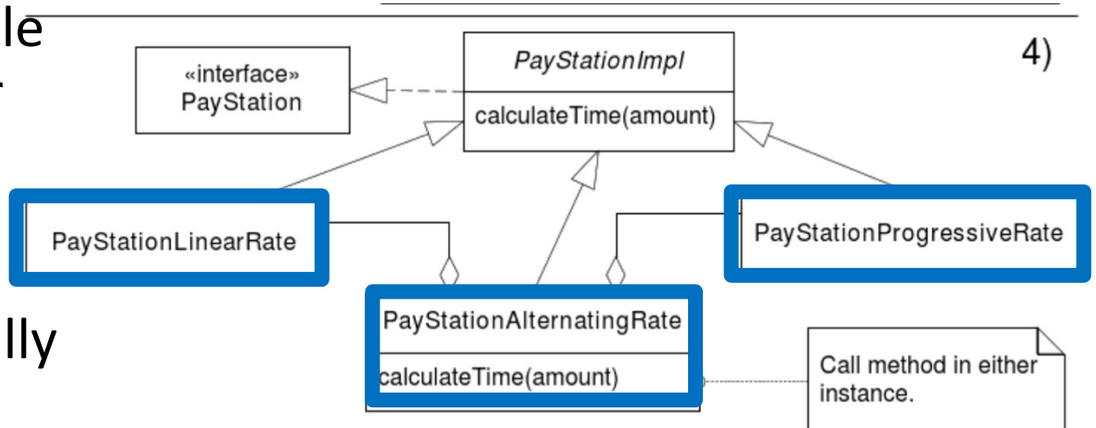
Other options:

1. Direct subclass of PayStationImpl
   - Duplicated code
2. Sub-subclass of PayStationLinearRate
   - Avoids some duplication (but not all), and creates confusing asymmetry
3. Superclass rate calculation
   - Avoids duplication, but does not scale and risks change by modification for future rate policies
4. Pay stations within pay station
   - Avoids duplication, but is conceptually confusing



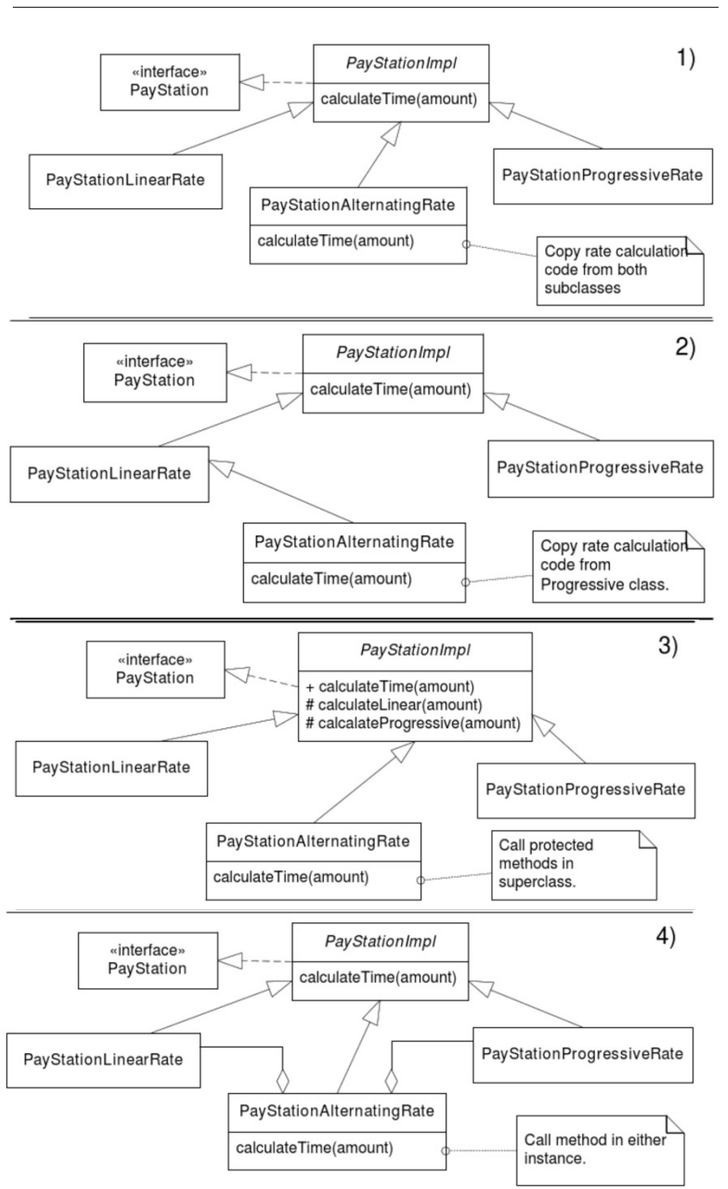



# Polymorphic Solution Proposal

How are our polymorphic options?
→ Not great!

Let's return to compositional design



# Compositional + Parametric Proposal

Option: Conditional for delegating to a rate strategy

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

# Compositional + Parametric Proposal

Option: Conditional for delegating to a rate strategy

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

Testing AlphaTown:

```
public void setUp() {
  ps = new PayStationImpl( new LinearRateStrategy(),
                           new LinearRateStrategy() );
}
```

# Compositional + Parametric Proposal

Option: Conditional for delegating to a rate strategy

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

Testing AlphaTown:

```java
public void setUp() {
  ps = new PayStationImpl( new LinearRateStrategy(),
                           new LinearRateStrategy() );
}
```

- Modifies code
- Strange constructor for AlphaTown and BetaTown (weak cohesion)
- Adds pay station responsibility

# Compositional Proposal

Recall 3-1-2 process:

**(3): Identify behavior that varies**

**(1): State a responsibility that covers the behavior and express it as an interface**

**(2): Compose the behavior by delegating**

# Compositional Proposal

Recall 3-1-2 process:

**(3): Identify behavior that varies**

→ Rate calculation

**(1): State a responsibility that covers the behavior and express it as an interface**

→ RateStrategy (already exists)

**(2): Compose the behavior by delegating**

→ Implement GammaTown behavior by combining existing rate calculations

# Compositional Proposal

Recall 3-1-2 process:

**(3): Identify behavior that varies**

→ Rate calculation

**(1): State a responsibility that covers the behavior and express it as an interface**

→ RateStrategy (already exists)

**(2): Compose the behavior by delegating**

→ Implement GammaTown behavior by combining existing rate calculations

→ Proposal: A "coordinator" delegates rate calculation to specialized "workers" (already-implemented rate calculations)

# Compositional Proposal

**Key Point: Object collaborations define compositional designs**

*When designing software compositionally, you make objects collaborate to achieve complex behavior.*

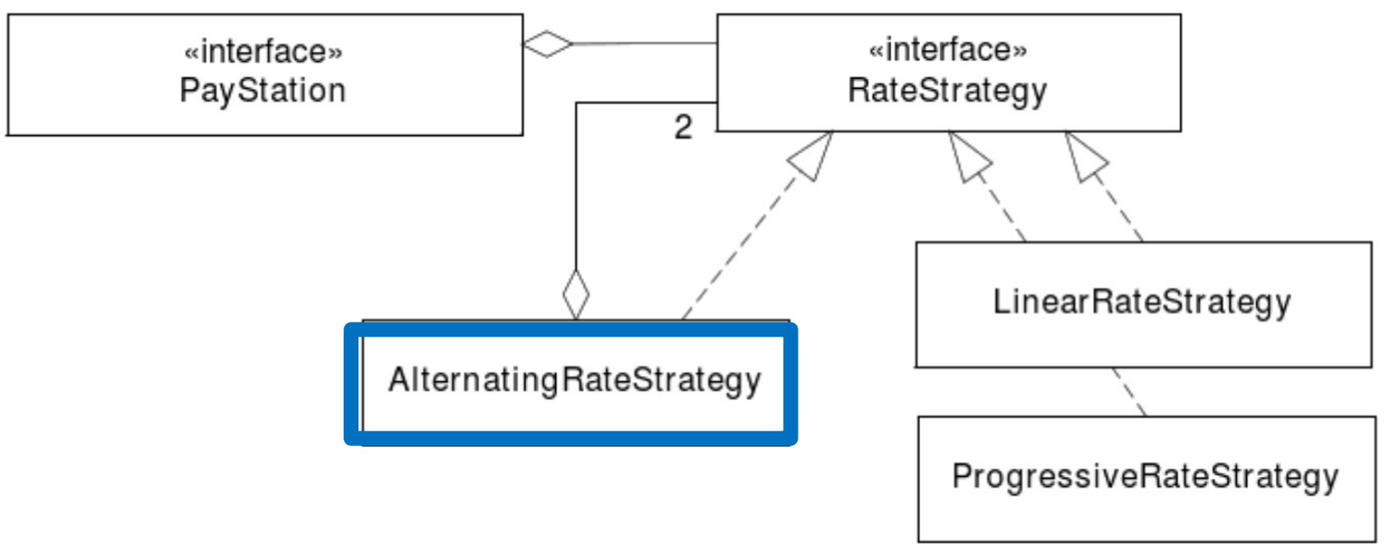



Figure 11.3: Rate calculation as a combined effort.

# Compositional Proposal

**Key Point: Object collaborations define compositional designs**

*When designing software compositionally, you make objects collaborate to achieve complex behavior.*

Change by **addition**

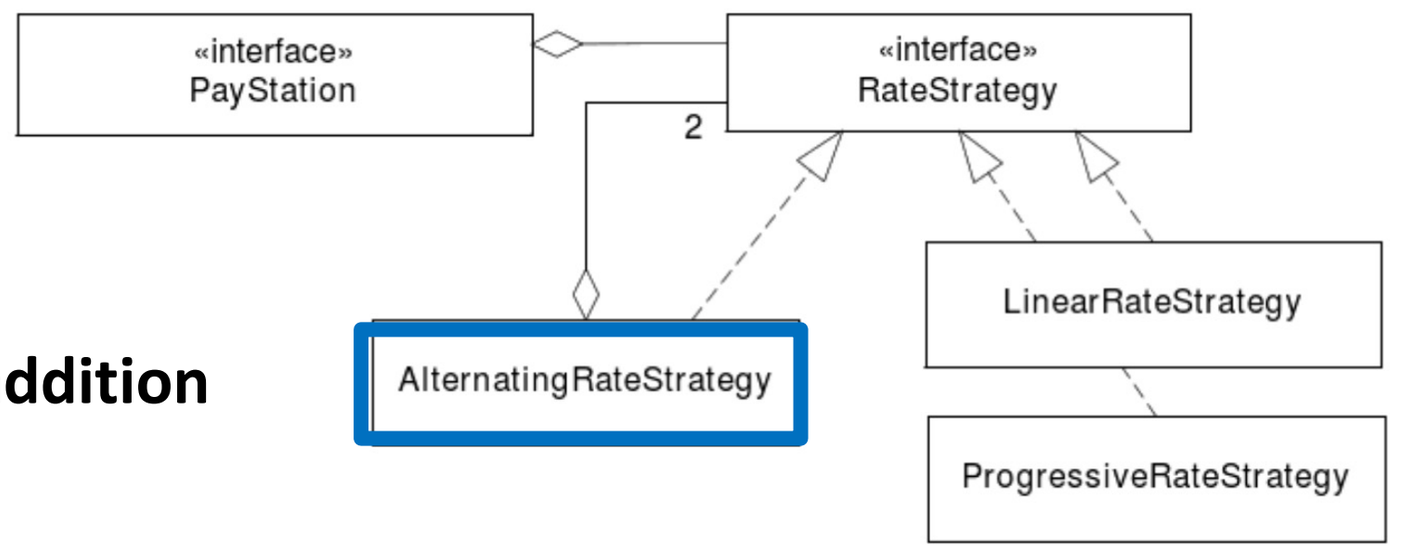



Figure 11.3: Rate calculation as a combined effort.

# TDD: Compositional Design

For TDD, how do we write a test for this?

# TDD: Compositional Design

For TDD, how do we write a test for this?

AlphaTown

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

# TDD: Compositional Design

For TDD, how do we write a test for this?

AlphaTown

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

GammaTown

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

# TDD: Compositional Design

For TDD, how do we write a test for this?

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

AlphaTown

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

GammaTown

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

# TDD: Compositional Design

Suppose we define:

```
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);
}
```

Do we need to run this on different days to test?

# TDD: Compositional Design

Suppose we define:

```
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);
}
```

Do we need to run this on different days to test?

This is an **indirect input parameter:**

It is not an instance variable of the object

It is not a parameter to the method

**It cannot be set by our test code ☹**

# TDD: Compositional Design

Suppose we define:

```
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);
}
```

Do we need to run this on different days to test?

This is an **indirect input parameter:**

It is not an instance variable of the object
It is not a parameter to the method
**It cannot be set by our test code** ☹

External resources are a more general problem that we will discuss more later; for now, we'll assume manual testing

# Compositional Design

AlternatingRateStrategy.java

```java
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 );
  }
```

```java
  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);
  }
}
```

# Compositional Design

AlternatingRateStrategy.java

```java
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 );
  }
```

```java
  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);
  }
}
```

# Compositional Design

AlternatingRateStrategy.java

```java
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 );
  }
```

Decide **state** of RateStrategy

```java
  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);
  }
}
```

Delegate calculation

# Compositional Design

AlternatingRateStrategy.java

```java
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 );
  }
```

This is the **State** design pattern

Decide **state** of RateStrategy

```java
  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);
  }
}
```

Delegate calculation

# Compositional Design: Analysis

AlternatingRateStrategy.java

```java
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 );
  }
```

```java
  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);
  }
}
```

TestAlternatingRate.java

```java
public void setUp() {
  RateStrategy rs =
    new AlternatingRateStrategy( new LinearRateStrategy(),
                                 new ProgressiveRateStrategy() );

  ps = new PayStationImpl( rs );
}
```

# Compositional Design: Analysis

- **Reliability**: this will not affect AlphaTown and BetaTown implementations
  - Return later to automated testing

AlternatingRateStrategy.java

```java
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 );
  }
```

```java
  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);
  }
}
```

TestAlternatingRate.java

```java
public void setUp() {
  RateStrategy rs =
    new AlternatingRateStrategy( new LinearRateStrategy(),
                                 new ProgressiveRateStrategy() );

  ps = new PayStationImpl( rs );
}
```

# Compositional Design: Analysis

- **Reliability**: this will not affect AlphaTown and BetaTown implementations
  - Return later to automated testing
- **Maintainability**: simple and straightforward implementation

AlternatingRateStrategy.java

```
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);
  }
}
```

TestAlternatingRate.java

```
public void setUp() {
  RateStrategy rs =
    new AlternatingRateStrategy( new LinearRateStrategy(),
                                 new ProgressiveRateStrategy() );

  ps = new PayStationImpl( rs );
}
```

# Compositional Design: Analysis

- **Reliability**: this will not affect AlphaTown and BetaTown implementations
  - Return later to automated testing
- **Maintainability**: simple and straightforward implementation
- **Client's interface is consistent**: pay station use of rate calculations has not changed

AlternatingRateStrategy.java

```
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);
  }
}
```

TestAlternatingRate.java

```
public void setUp() {
  RateStrategy rs =
    new AlternatingRateStrategy( new LinearRateStrategy(),
                                 new ProgressiveRateStrategy() );

  ps = new PayStationImpl( rs );
}
```

# Compositional Design: Analysis

- **Reliability**: this will not affect AlphaTown and BetaTown implementations
  - Return later to automated testing
- **Maintainability**: simple and straightforward implementation
- **Client's interface is consistent**: pay station use of rate calculations has not changed
- **Reuse**: can be applied to any future customers wanting weekday/weekend variation

AlternatingRateStrategy.java

```java
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 );
  }
```

```java
  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);
  }
}
```

TestAlternatingRate.java

```java
public void setUp() {
  RateStrategy rs =
    new AlternatingRateStrategy( new LinearRateStrategy(),
                                 new ProgressiveRateStrategy() );

  ps = new PayStationImpl( rs );
}
```

# Compositional Design: Analysis

- **Reliability**: this will not affect AlphaTown and BetaTown implementations
  - Return later to automated testing
- **Maintainability**: simple and straightforward implementation
- **Client's interface is consistent**: pay station use of rate calculations has not changed
- **Reuse**: can be applied to any future customers wanting weekday/weekend variation
- **Flexibility**: can change by addition for new requirements

AlternatingRateStrategy.java

```java
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 );
  }
```

```java
  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);
  }
}
```

TestAlternatingRate.java

```java
public void setUp() {
  RateStrategy rs =
    new AlternatingRateStrategy( new LinearRateStrategy(),
                                 new ProgressiveRateStrategy() );

  ps = new PayStationImpl( rs );
}
```

# Compositional Design: Analysis

- **Reliability**: this will not affect AlphaTown and BetaTown implementations
  - Return later to automated testing
- **Maintainability**: simple and straightforward implementation
- **Client's interface is consistent**: pay station use of rate calculations has not changed
- **Reuse**: can be applied to any future customers wanting weekday/weekend variation
- **Flexibility**: can change by addition for new requirements
- **Cohesion**: state-specific behavior is localized to AlternatingRateStrategy

AlternatingRateStrategy.java

```java
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 );
  }
```

```java
  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);
  }
}
```

TestAlternatingRate.java

```java
public void setUp() {
  RateStrategy rs =
    new AlternatingRateStrategy( new LinearRateStrategy(),
                                 new ProgressiveRateStrategy() );

  ps = new PayStationImpl( rs );
}
```

# Compositional Design: Analysis

- **Reliability**: this will not affect AlphaTown and BetaTown implementations
  - Return later to automated testing
- **Maintainability**: simple and straightforward implementation
- **Client's interface is consistent**: pay station use of rate calculations has not changed
- **Reuse**: can be applied to any future customers wanting weekday/weekend variation
- **Flexibility**: can change by addition for new requirements
- **Cohesion**: state-specific behavior is localized to AlternatingRateStrategy

- Increased number of objects

AlternatingRateStrategy.java

```java
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 );
  }
```

```java
  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);
  }
}
```

TestAlternatingRate.java

```java
public void setUp() {
  RateStrategy rs =
    new AlternatingRateStrategy( new LinearRateStrategy(),
                                 new ProgressiveRateStrategy() );

  ps = new PayStationImpl( rs );
}
```

# State Pattern

**State pattern:** allow an object to alter its behavior when its internal state changes

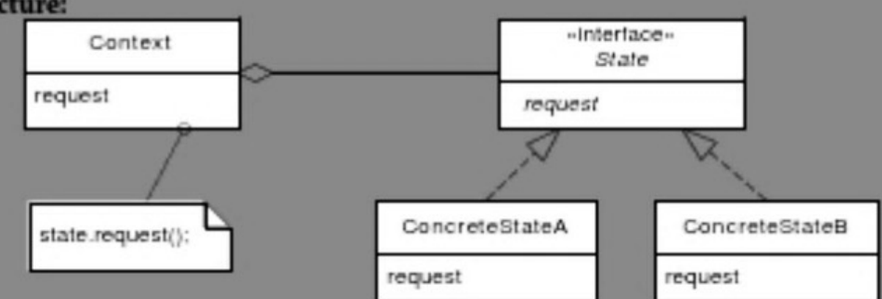
**[11.1] Design Pattern: State**

**Intent** Allow an object to alter its behavior when its internal state changes.

**Problem** Your product's behavior varies at run-time depending upon some internal state.

**Solution** Describe the responsibilities of the dynamically varying behavior in an interface and implement the concrete behavior associated with each unique state in an object, the state object, that implements this interface. The context object delegates to its current state object. When internal state changes occur, the current state object reference is changed to refer to the corresponding state object.

**Structure:**




**Roles** **State** specifies the responsibilities and interface of the varying behavior associated with a state, and **ConcreteState** objects define the specific behavior associated with each specific state. The **Context** object delegates to its current state object. The state object reference is changed whenever the context changes its internal state.

**Cost - Benefit** *State specific behavior is localized* as all behavior associated with a specific state is in a single class. It *makes state transitions explicit* as assigning the current state object is the only way to change state. A liability is the *increased number of objects and interactions* compared to a state machine based upon conditional statements in the context object.

# State Pattern

**State pattern:** allow an object to alter its behavior when its internal state changes

- **Context** object delegates requests to the state object

- Internal state changes change the concrete **state object**

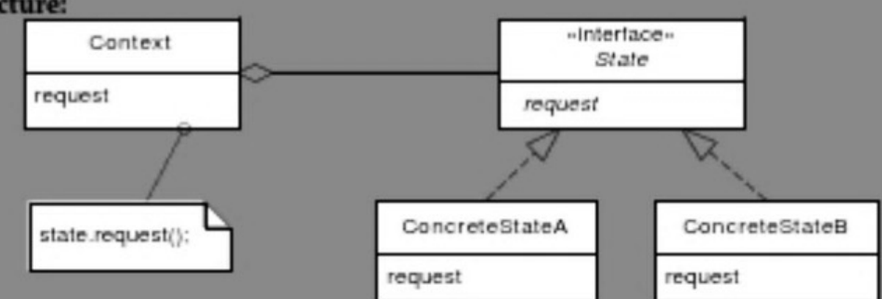
**[11.1] Design Pattern: State**

**Intent** Allow an object to alter its behavior when its internal state changes.

**Problem** Your product's behavior varies at run-time depending upon some internal state.

**Solution** Describe the responsibilities of the dynamically varying behavior in an interface and implement the concrete behavior associated with each unique state in an object, the state object, that implements this interface. The context object delegates to its current state object. When internal state changes occur, the current state object reference is changed to refer to the corresponding state object.

**Structure:**




**Roles** **State** specifies the responsibilities and interface of the varying behavior associated with a state, and **ConcreteState** objects define the specific behavior associated with each specific state. The **Context** object delegates to its current state object. The state object reference is changed whenever the context changes its internal state.

**Cost - Benefit** *State specific behavior is localized* as all behavior associated with a specific state is in a single class. It *makes state transitions explicit* as assigning the current state object is the only way to change state. A liability is the *increased number of objects and interactions* compared to a state machine based upon conditional statements in the context object.

# State Pattern

**State pattern:** allow an object to alter its behavior when its internal state changes

- **Context** object delegates requests to the state object
  - ?
- Internal state changes change the concrete **state object**
  - ?

```
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 );
  }
```

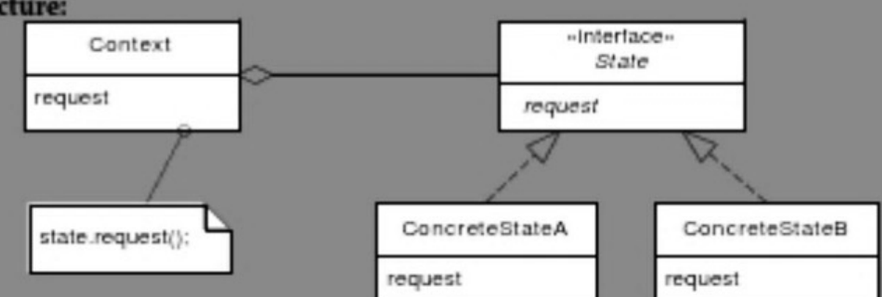
**[11.1] Design Pattern: State**

**Intent** Allow an object to alter its behavior when its internal state changes.

**Problem** Your product's behavior varies at run-time depending upon some internal state.

**Solution** Describe the responsibilities of the dynamically varying behavior in an interface and implement the concrete behavior associated with each unique state in an object, the state object, that implements this interface. The context object delegates to its current state object. When internal state changes occur, the current state object reference is changed to refer to the corresponding state object.

**Structure:**




**Roles** **State** specifies the responsibilities and interface of the varying behavior associated with a state, and **ConcreteState** objects define the specific behavior associated with each specific state. The **Context** object delegates to its current state object. The state object reference is changed whenever the context changes its internal state.

**Cost - Benefit** *State specific behavior is localized* as all behavior associated with a specific state is in a single class. It *makes state transitions explicit* as assigning the current state object is the only way to change state. A liability is the *increased number of objects and interactions* compared to a state machine based upon conditional statements in the context object.

# State Pattern

**State pattern:** allow an object to alter its behavior when its internal state changes

- **Context** object delegates requests to the state object
  - AlternatingRateStrategy
- Internal state changes change the concrete **state object**
  - LinearRateStrategy
  - ProgressiveRateStrategy

```
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 );
  }
```

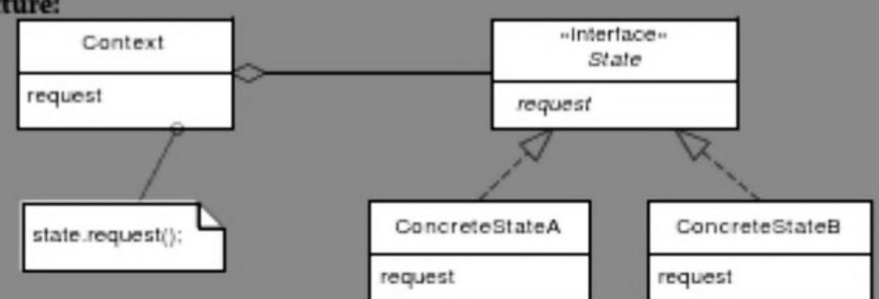
**[11.1] Design Pattern: State**

**Intent** Allow an object to alter its behavior when its internal state changes.

**Problem** Your product's behavior varies at run-time depending upon some internal state.

**Solution** Describe the responsibilities of the dynamically varying behavior in an interface and implement the concrete behavior associated with each unique state in an object, the state object, that implements this interface. The context object delegates to its current state object. When internal state changes occur, the current state object reference is changed to refer to the corresponding state object.

**Structure:**




**Roles** **State** specifies the responsibilities and interface of the varying behavior associated with a state, and **ConcreteState** objects define the specific behavior associated with each specific state. The **Context** object delegates to its current state object. The state object reference is changed whenever the context changes its internal state.

**Cost - Benefit** *State specific behavior is localized* as all behavior associated with a specific state is in a single class. It *makes state transitions explicit* as assigning the current state object is the only way to change state. A liability is the *increased number of objects and interactions* compared to a state machine based upon conditional statements in the context object.

# State Pattern

**State pattern:** allow an object to alter its behavior when its internal state changes

- **Context** object delegates requests to the state object
  - AlternatingRateStrategy
- Internal state changes change the concrete **state object**
  - LinearRateStrategy
  - ProgressiveRateStrategy

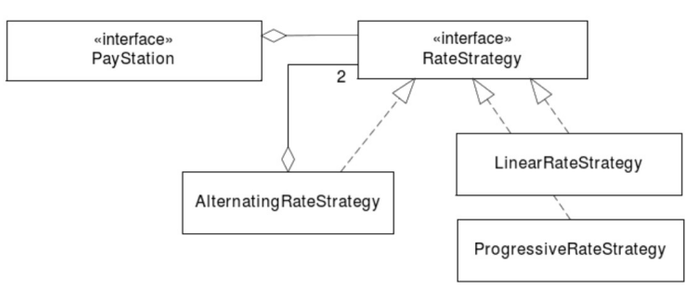



Figure 11.3: Rate calculation as a combined effort.

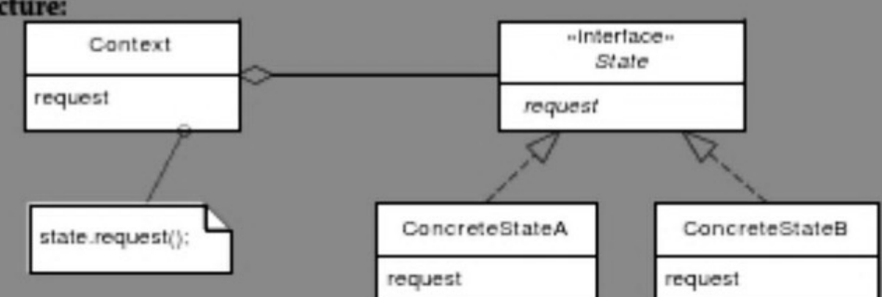
**[11.1] Design Pattern: State**

**Intent** Allow an object to alter its behavior when its internal state changes.

**Problem** Your product's behavior varies at run-time depending upon some internal state.

**Solution** Describe the responsibilities of the dynamically varying behavior in an interface and implement the concrete behavior associated with each unique state in an object, the state object, that implements this interface. The context object delegates to its current state object. When internal state changes occur, the current state object reference is changed to refer to the corresponding state object.

**Structure:**




**Roles** **State** specifies the responsibilities and interface of the varying behavior associated with a state, and **ConcreteState** objects define the specific behavior associated with each specific state. The **Context** object delegates to its current state object. The state object reference is changed whenever the context changes its internal state.

**Cost - Benefit** *State specific behavior is localized* as all behavior associated with a specific state is in a single class. It *makes state transitions explicit* as assigning the current state object is the only way to change state. A liability is the *increased number of objects and interactions* compared to a state machine based upon conditional statements in the context object.

Same structure as Strategy pattern!

# State Pattern

Design patterns are defined by the **problems** they solve

→ The characteristics of the problem will define the proper pattern to apply

# State Pattern

Design patterns are defined by the **problems** they solve

→ The characteristics of the problem will define the proper pattern to apply

- **State pattern:** provides behavior that varies according to an object's internal state
- **Strategy pattern**: handles variability of business rules or algorithms

# State Pattern

Design patterns are defined by the **problems** they solve

→ The characteristics of the problem will define the proper pattern to apply

- **State pattern:** provides behavior that varies according to an object's internal state
- **Strategy pattern**: handles variability of business rules or algorithms

PatternCraft: State
https://www.youtube.com/watch?v=yZt7mUVDijU&list=PL8B19C3040F6381A2&index=1

Next Time: How do we test the alternating rate? → Test Stubs