Lecture 15

ECE 1145: Software Construction and Evolution

Multi-Dimensional Variance
HotCiv Compositional Design
(CH 17, 18)

Announcements

- Relevant Exercises: 17.5, 18.4
- Midterm Survey on Canvas
- Iteration 5: Test Stubs, State, Abstract Factory due Oct. 31

Multi-Dimensional Variation

New Customer Requirement!

- AlphaTown wants the pay station to display the **time when parking expires** (instead of the number of minutes, as previously)
  - 4-digit display => display expiration time in 24 hour format

Multi-Dimensional Variation

New Customer Requirement!

- AlphaTown wants the pay station to display the **time when parking expires** (instead of the number of minutes, as previously)
  - 4-digit display => display expiration time in 24 hour format

How does this add to our current dimensions of variation?

Multi-Dimensional Variation

New Customer Requirement!

- AlphaTown wants the pay station to display the **time when parking expires** (instead of the number of minutes, as previously)
  - 4-digit display => display expiration time in 24 hour format

How does this add to our current dimensions of variation?

→**Rate calculation**: linear, progressive, alternating

→**Receipt format:** standard, barcode

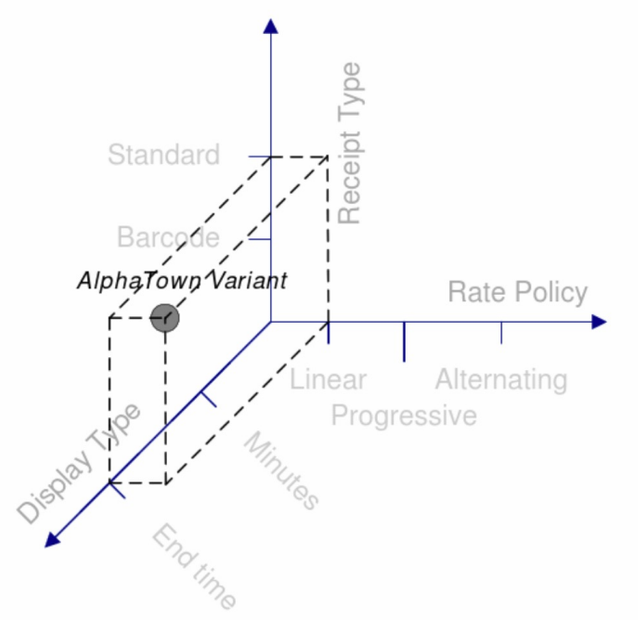
→**Display output**: minutes of parking time, time when parking ends (new!)

Multi-Dimensional Variation

Configuration table:

	Variability points		
Product	**Rate**	**Receipt**	**Display**
Alphatown	Linear	Standard	End time
Betatown	Progressive	Barcode	Minutes
Gammatown	Alternating	Standard	Minutes

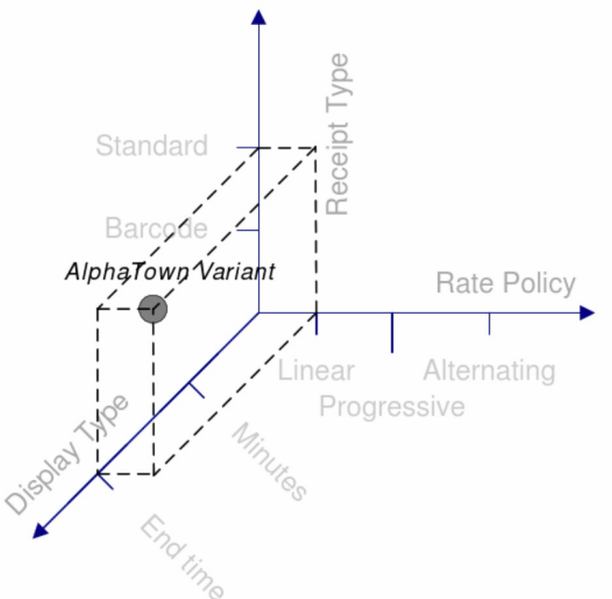
→ 3 variability dimensions



Multi-Dimensional Variation

Configuration table:

	Variability points		
Product	**Rate**	**Receipt**	**Display**
Alphatown	Linear	Standard	End time
Betatown	Progressive	Barcode	Minutes
Gammatown	Alternating	Standard	Minutes



→ 3 variability dimensions

Dimensions are independent!

→ **All combinations are valid**

- 3 rate policies x 2 receipt types x 2 display options = 12 variants
- More policies could lead to combinatorial explosion

Exploring Design Proposals

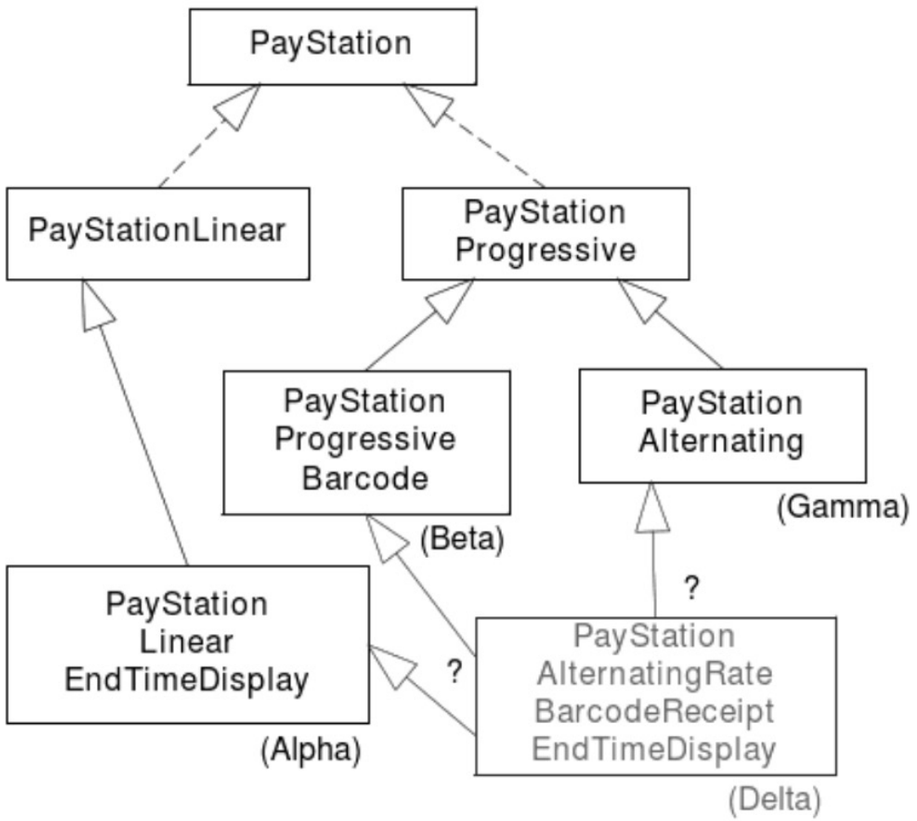
	Variability points		
Product	**Rate**	**Receipt**	**Display**
Alphatown	Linear	Standard	End time
Betatown	Progressive	Barcode	Minutes
Gammatown	Alternating	Standard	Minutes

1. Polymorphic (including previous variations)
2. Compositional

Polymorphic Proposal

	Variability points		
Product	**Rate**	**Receipt**	**Display**
Alphatown	Linear	Standard	End time
Betatown	Progressive	Barcode	Minutes
Gammatown	Alternating	Standard	Minutes

As we might guess, the polymorphic approach does not handle multi-dimensional variability well!

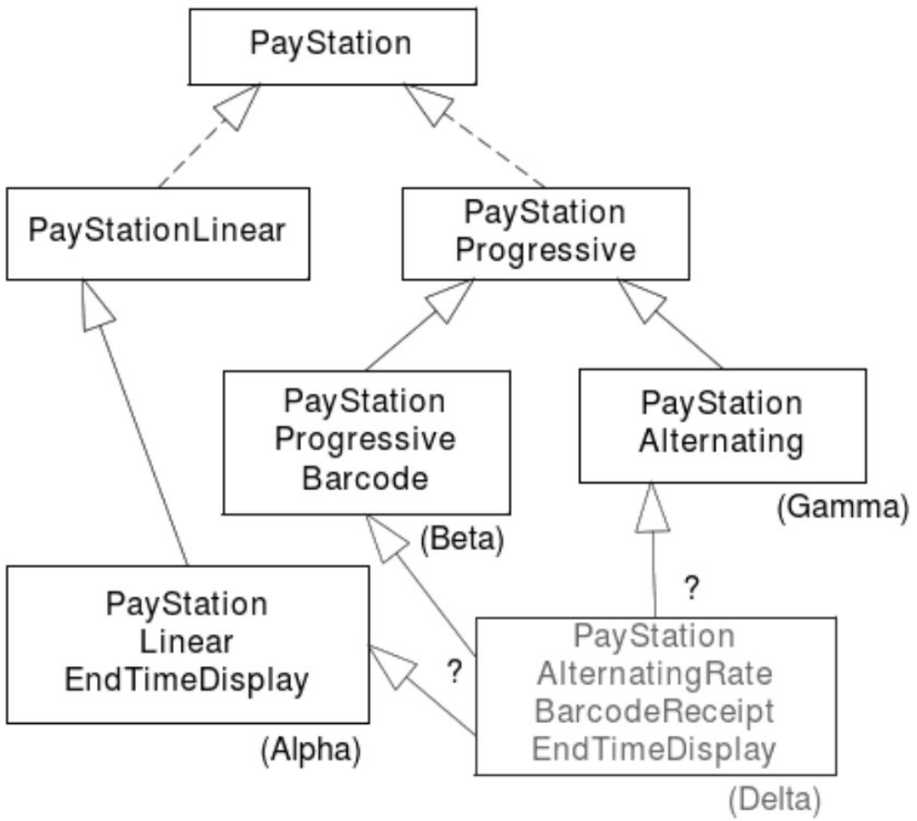


Polymorphic Proposal

	Variability points		
Product	**Rate**	**Receipt**	**Display**
Alphatown	Linear	Standard	End time
Betatown	Progressive	Barcode	Minutes
Gammatown	Alternating	Standard	Minutes

As we might guess, the polymorphic approach does not handle multi-dimensional variability well!

What if we need to add a fourth town variant? Where do we subclass?



Polymorphic Proposal

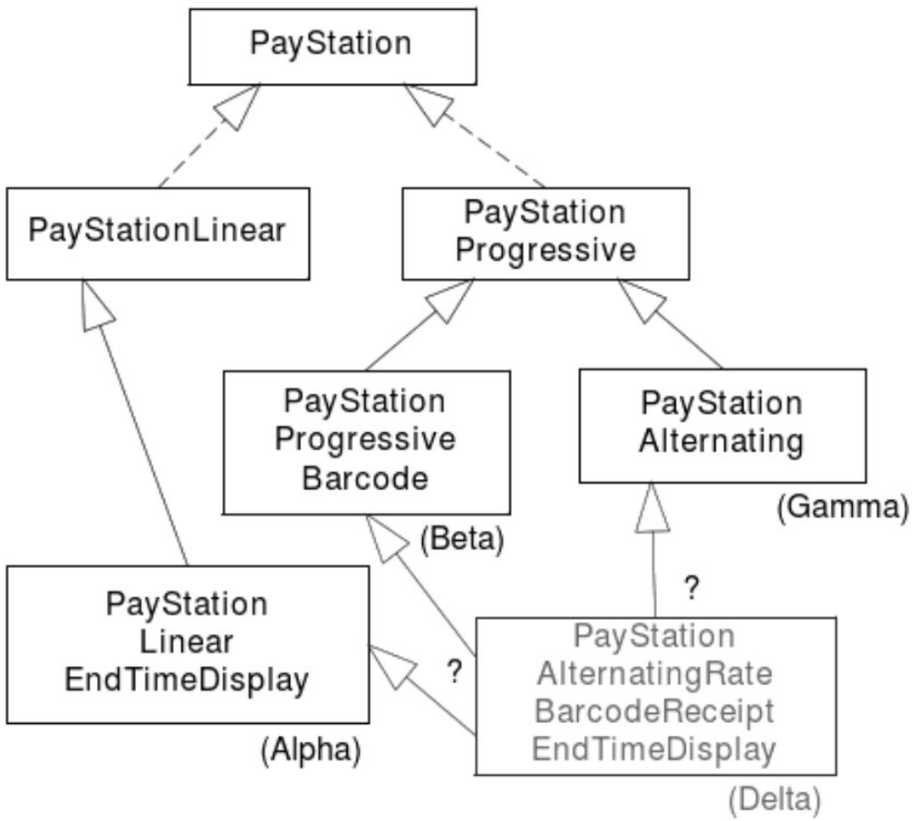
	Variability points		
Product	**Rate**	**Receipt**	**Display**
Alphatown	Linear	Standard	End time
Betatown	Progressive	Barcode	Minutes
Gammatown	Alternating	Standard	Minutes

As we might guess, the polymorphic approach does not handle multi-dimensional variability well!

What if we need to add a fourth town variant? Where do we subclass?

- Either duplicate code, or the root class becomes a pile of methods only relevant for a few subclasses

→ **Bad cohesion and analyzability**



Polymorphic Proposal

	Variability points		
Product	**Rate**	**Receipt**	**Display**
Alphatown	Linear	Standard	End time
Betatown	Progressive	Barcode	Minutes
Gammatown	Alternating	Standard	Minutes

Inheritance is one-dimensional
(in Java and C#)

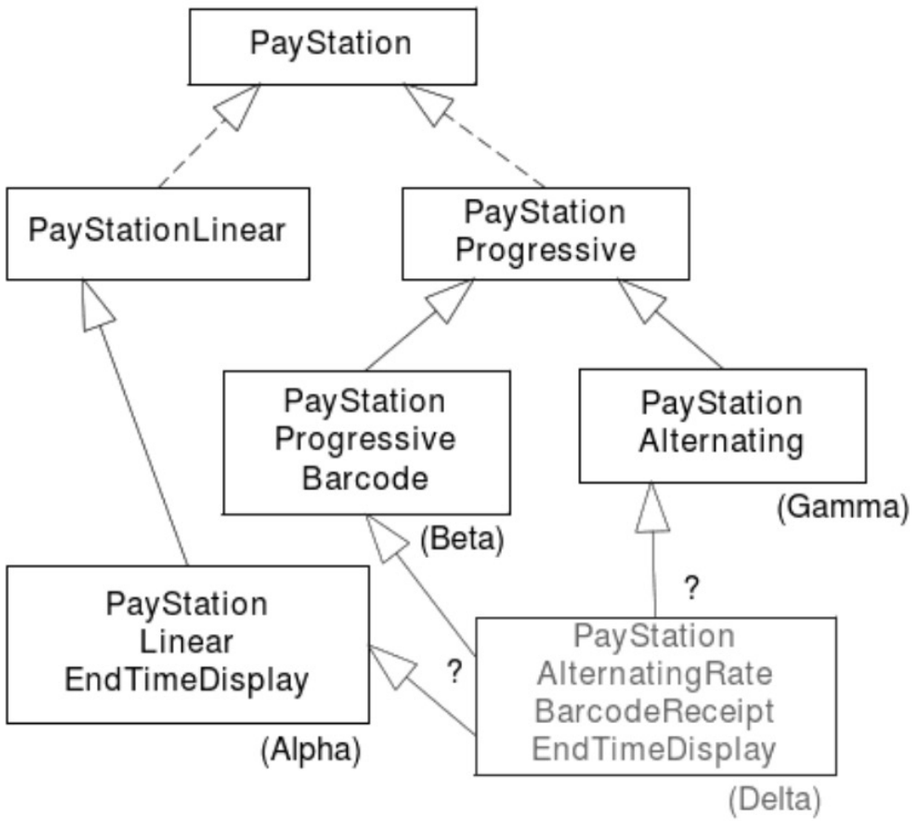
→Can only "extend" one class
(one superclass)

Multiple inheritance is allowed in C++

```
class Derived : public BaseA, public BaseB
```

But, it can create implementation conflicts.

- What if BaseA, BaseB have implemented/overridden the same method?

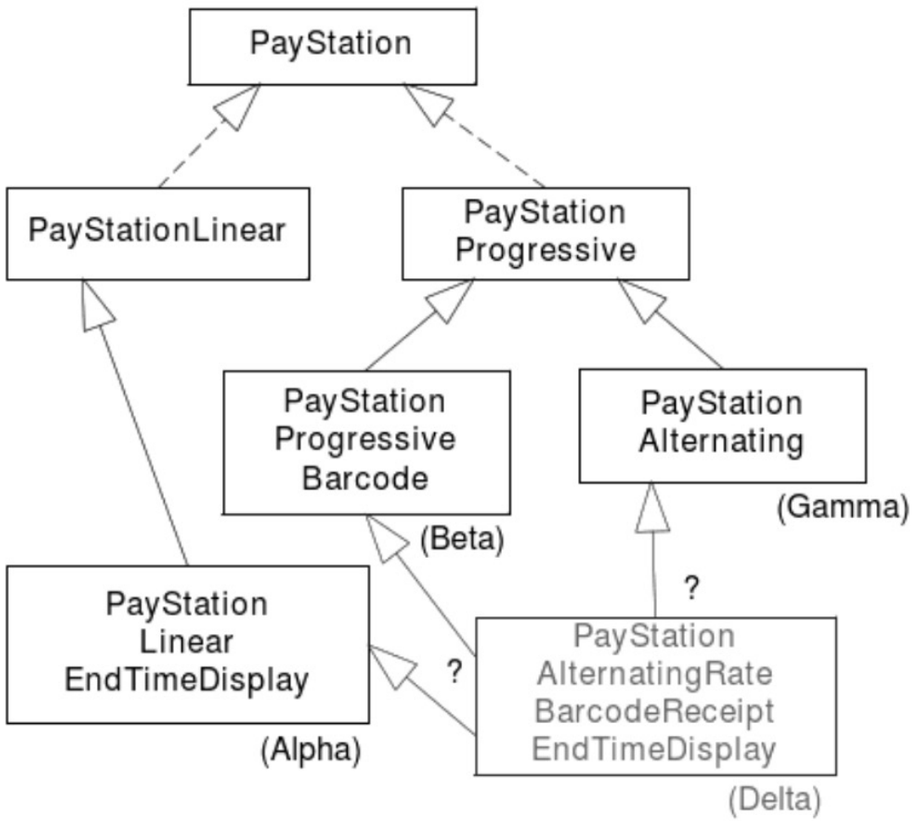


Polymorphic Proposal

	Variability points		
Product	**Rate**	**Receipt**	**Display**
Alphatown	Linear	Standard	End time
Betatown	Progressive	Barcode	Minutes
Gammatown	Alternating	Standard	Minutes

Inheritance is one-dimensional
(in Java and C#)

→Can only "extend" one class
(one superclass)



Multiple inheritance is allowed in C++

```
class Derived : public BaseA, public BaseB
```

But, it can create implementation conflicts.

- What if BaseA, BaseB have implemented/overridden the same method?

Key point: Do not use inheritance to handle multi-dimensional variation

Compositional Proposal (3-1-2)

3. **What behavior varies?**

Compositional Proposal (3-1-2)

3. **What behavior varies?** → Display output (PayStation readDisplay)

Compositional Proposal (3-1-2)

3. What behavior varies? → Display output (PayStation readDisplay)

1. **Program to an interface:** Strategy pattern

Compositional Proposal (3-1-2)

3. What behavior varies? → Display output (PayStation readDisplay)

1. **Program to an interface:** Strategy pattern

Introduce DisplayStrategy:

```
package paystation.domain;
/** The strategy for calculating the output for the display.
*/

public interface DisplayStrategy {
  /** return the output to present at the pay station's
      display
      @param minutes the minutes parking time
      bought so far.
   */
  public int calculateOutput( int minutes );
}
```

Compositional Proposal (3-1-2)

3. What behavior varies? → Display output (PayStation readDisplay)

1. **Program to an interface:** Strategy pattern

Introduce DisplayStrategy:

```
package paystation.domain;
/** The strategy for calculating the output for the display.
*/

public interface DisplayStrategy {
  /** return the output to present at the pay station's
      display
      @param minutes the minutes parking time
      bought so far.
   */
  public int calculateOutput( int minutes );
}
```

Refactor pay station to use DisplayStrategy:

```
  public int readDisplay() {
    return displayStrategy.calculateOutput(timeBought);
  }
```

Compositional Proposal (3-1-2)

3. What behavior varies? → Display output (PayStation readDisplay)

1. Program to an interface: Strategy pattern

Introduce DisplayStrategy:

```
package paystation.domain;
/** The strategy for calculating the output for the display.
*/

public interface DisplayStrategy {
  /** return the output to present at the pay station's
      display
      @param minutes the minutes parking time
      bought so far.
   */
  public int calculateOutput( int minutes );
}
```

Refactor pay station to use DisplayStrategy:

```
  public int readDisplay() {
    return displayStrategy.calculateOutput(timeBought);
  }
```

2. **Favor object composition**

Compositional Proposal (3-1-2)

3. What behavior varies? → Display output

1. Program to an interface: Strategy pattern

Introduce DisplayStrategy:

```
package paystation.domain;
/** The strategy for calculating the output for the display.
*/

public interface DisplayStrategy {
  /** return the output to present at the pay station's
      display
      @param minutes the minutes parking time
      bought so far.
   */
  public int calculateOutput( int minutes );
}
```

We can make all 12 variants by configuring the set of delegate objects that PayStationImpl should use.

Refactor pay station to use DisplayStrategy:

```
  public int readDisplay() {
    return displayStrategy.calculateOutput(timeBought);
  }
```

2. **Favor object composition**

Compositional Proposal (3-1-2)

3. What behavior varies? → Display output

1. Program to an interface: Strategy pattern

Introduce DisplayStrategy:

```
package paystation.domain;
/** The strategy for calculating the output for the display.
*/

public interface DisplayStrategy {
  /** return the output to present at the pay station's
      display
      @param minutes the minutes parking time
      bought so far.
   */
  public int calculateOutput( int minutes );
}
```

Refactor pay station to use DisplayStrategy:

```
  public int readDisplay() {
    return displayStrategy.calculateOutput(timeBought);
  }
```

2. Favor object composition

We can make all 12 variants by configuring the set of delegate objects that PayStationImpl should use.

We have a **community** of interacting objects in which each object has a **role** to play

Compositional Proposal (3-1-2)

3. What behavior varies? → Display output

1. Program to an interface: Strategy pattern

Introduce DisplayStrategy:

```
package paystation.domain;
/** The strategy for calculating the output for the display.
*/

public interface DisplayStrategy {
  /** return the output to present at the pay station's
      display
      @param minutes the minutes parking time
      bought so far.
   */
  public int calculateOutput( int minutes );
}
```

Refactor pay station to use DisplayStrategy:

```
  public int readDisplay() {
    return displayStrategy.calculateOutput(timeBought);
  }
```

2. Favor object composition

We can make all 12 variants by configuring the set of delegate objects that PayStationImpl should use.

We have a **community** of interacting objects in which each object has a **role** to play

→ Rate calculator
→ Receipt creator
→ Display output calculator
→ Coordinator (PayStation)

Compositional Proposal (3-1-2)

3. What behavior varies? → Display output

1. Program to an interface: Strategy pattern

Introduce DisplayStrategy:

```
package paystation.domain;
/** The strategy for calculating the output for the display.
*/

public interface DisplayStrategy {
  /** return the output to present at the pay station's
      display
      @param minutes the minutes parking time
      bought so far.
   */
  public int calculateOutput( int minutes );
}
```

We can introduce more variability through **addition**, **without modification** of existing strategies.

Refactor pay station to use DisplayStrategy:

```
  public int readDisplay() {
    return displayStrategy.calculateOutput(timeBought);
  }
```

2. Favor object composition

Compositional Proposal (3-1-2)

3. What behavior varies? → Display output

1. Program to an interface: Strategy pattern

Introduce DisplayStrategy:

```
package paystation.domain;
/** The strategy for calculating the output for the display.
*/

public interface DisplayStrategy {
  /** return the output to present at the pay station's
      display
      @param minutes the minutes parking time
      bought so far.
   */
  public int calculateOutput( int minutes );
}
```

Refactor pay station to use DisplayStrategy:

```
  public int readDisplay() {
    return displayStrategy.calculateOutput(timeBought);
  }
```

2. Favor object composition

- Independence of variability points may complicate transfer of information.
- Need lots of classes (naming is important for analyzability)

Compositional Proposal (3-1-2)

3. What behavior varies? → Display output

1. Program to an interface: Strategy pattern

Introduce DisplayStrategy:

```
package paystation.domain;
/** The strategy for calculating the output for the display.
*/

public interface DisplayStrategy {
  /** return the output to present at the pay station's
      display
      @param minutes the minutes parking time
      bought so far.
   */
  public int calculateOutput( int minutes );
}
```

Refactor pay station to use DisplayStrategy:

```
  public int readDisplay() {
    return displayStrategy.calculateOutput(timeBought);
  }
```

2. Favor object composition

Couldn't we still end up with 12 variants and 12 factory classes? Is this better than 12 subclasses?

Compositional Proposal (3-1-2)

3. What behavior varies? → Display output

1. Program to an interface: Strategy pattern

Introduce DisplayStrategy:

```
package paystation.domain;
/** The strategy for calculating the output for the display.
*/

public interface DisplayStrategy {
  /** return the output to present at the pay station's
      display
      @param minutes the minutes parking time
      bought so far.
   */
  public int calculateOutput( int minutes );
}
```

Refactor pay station to use DisplayStrategy:

```
  public int readDisplay() {
    return displayStrategy.calculateOutput(timeBought);
  }
```

2. Favor object composition

Couldn't we still end up with 12 variants and 12 factory classes? Is this better than 12 subclasses?

With strategies/factory we have less duplicated code, and each factory create method is basically just one "new" statement.

Compositional Proposal (3-1-2)

3. What behavior varies? → Display output

1. Program to an interface: Strategy pattern

Introduce DisplayStrategy:

```
package paystation.domain;
/** The strategy for calculating the output for the display.
*/

public interface DisplayStrategy {
  /** return the output to present at the pay station's
      display
      @param minutes the minutes parking time
      bought so far.
   */
  public int calculateOutput( int minutes );
}
```

Refactor pay station to use DisplayStrategy:

```
  public int readDisplay() {
    return displayStrategy.calculateOutput(timeBought);
  }
```

2. Favor object composition

Couldn't we still end up with 12 variants and 12 factory classes? Is this better than 12 subclasses?

With strategies/factory we have less duplicated code, and each factory create method is basically just one "new" statement.

Use configuration files if we really need all 12 variants; file is used by one implementation class to determine the proper delegates.

Design Patterns as Roles

Recall: Strategy and State patterns, interface, classes, and relationships

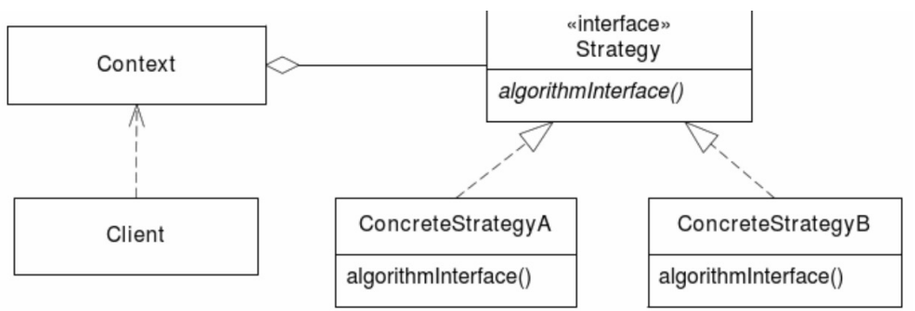


Figure 18.1: STRATEGY pattern structure in UML.

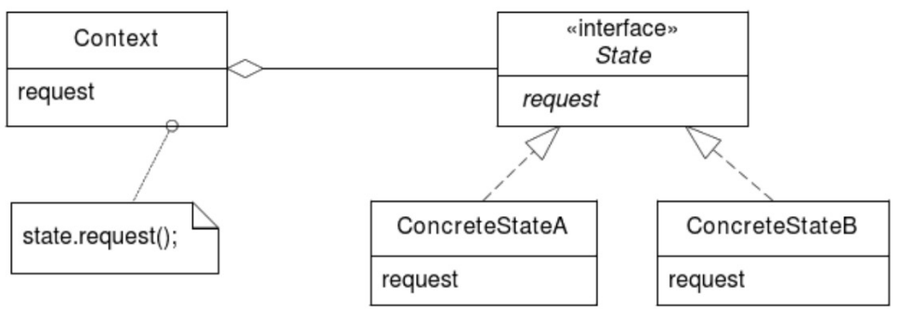


Figure 18.2: STATE pattern structure in UML.

Design Patterns as Roles

Recall: Strategy and State patterns, interface, classes, and relationships

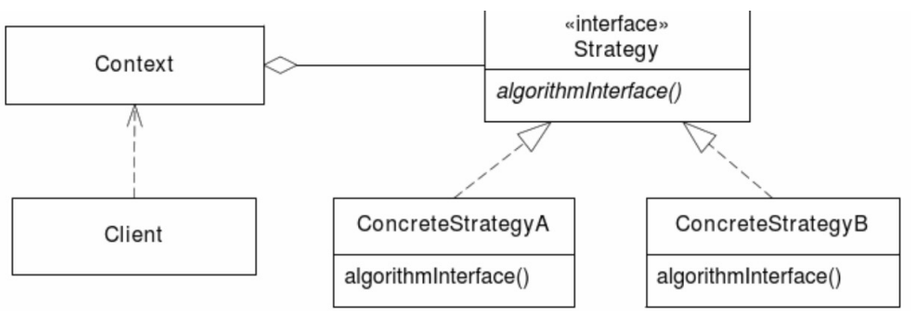


Figure 18.1: STRATEGY pattern structure in UML.

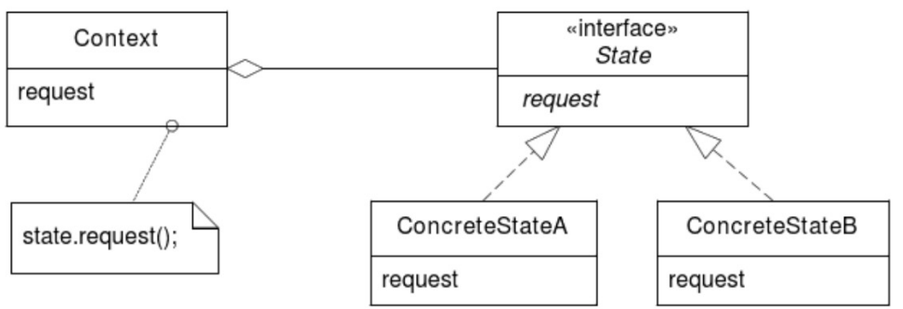


Figure 18.2: STATE pattern structure in UML.

GammaTown
alternating rate strategy:

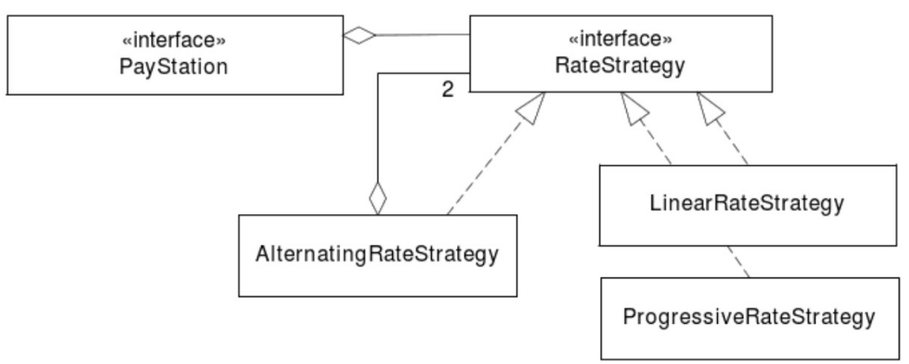


Figure 18.3: The combination of STRATEGY and STATE.

Design Patterns as Roles

Recall: Strategy and State patterns, interface, classes, and relationships

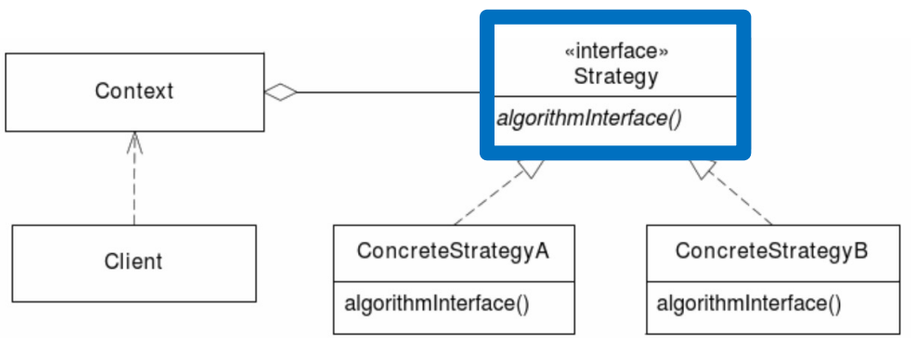


Figure 18.1: STRATEGY pattern structure in UML.

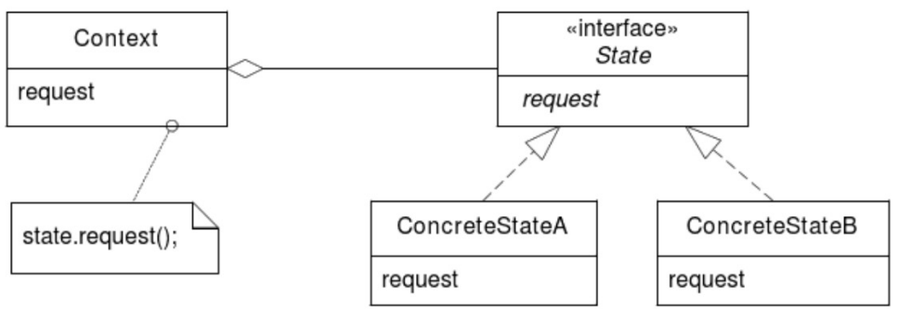


Figure 18.2: STATE pattern structure in UML.

GammaTown
alternating rate strategy:

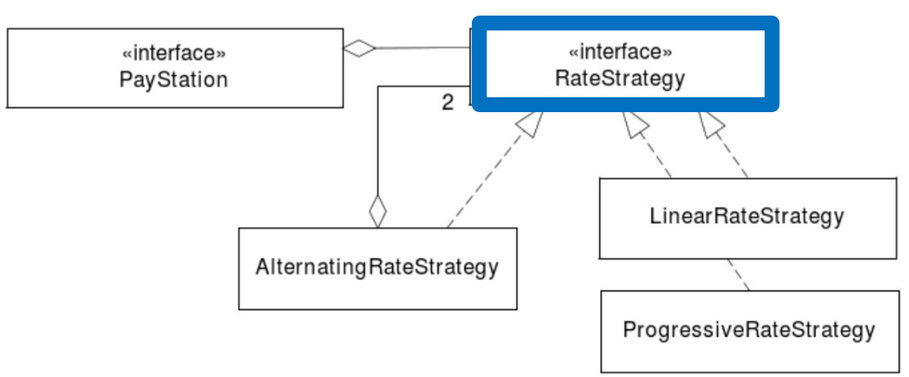


Figure 18.3: The combination of STRATEGY and STATE.

Design Patterns as Roles

Recall: Strategy and State patterns, interface, classes, and relationships

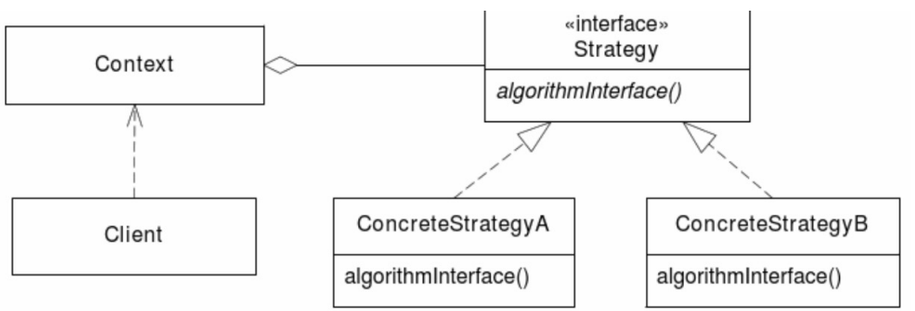


Figure 18.1: STRATEGY pattern structure in UML.

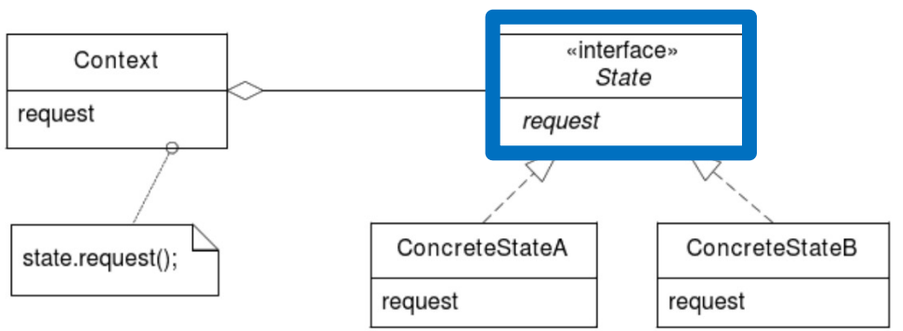


Figure 18.2: STATE pattern structure in UML.

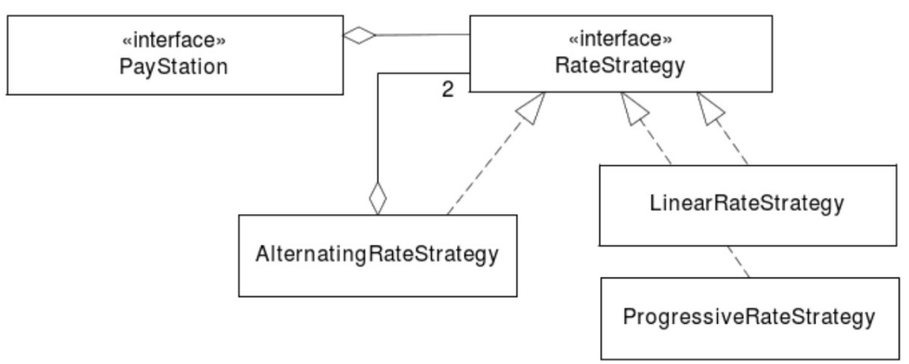


Figure 18.3: The combination of STRATEGY and STATE.

GammaTown
alternating rate strategy:

Where is state?

Design Patterns as Roles

Definition: Design pattern (Role view)

A design pattern is defined by a set of roles, each role having a specific set of responsibilities, and by a well defined protocol between these roles.

In the pattern diagrams, think of the Strategy interface as a Strategy **role,** and the State interface as a State **role**

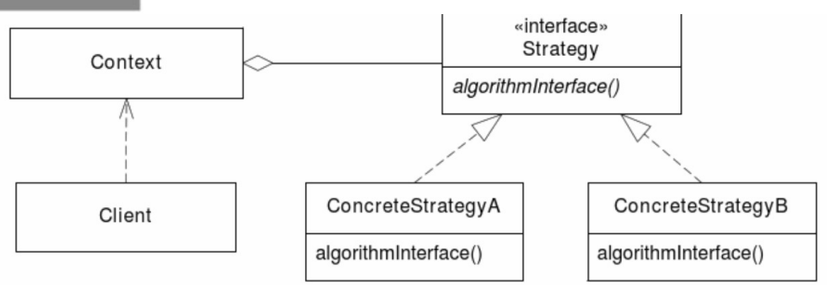


Figure 18.1: STRATEGY pattern structure in UML.

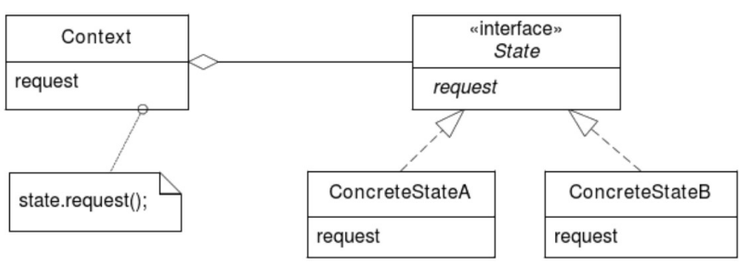


Figure 18.2: STATE pattern structure in UML.

Design Patterns as Roles

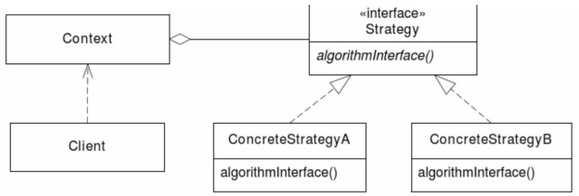


Figure 18.1: STRATEGY pattern structure in UML.

Definition: Design pattern (Role view)

A design pattern is defined by a set of roles, each role having a specific set of responsibilities, and by a well defined protocol between these roles.

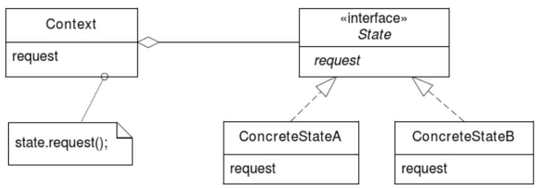


Figure 18.2: STATE pattern structure in UML.

In the pattern diagrams, think of the Strategy interface as a Strategy **role,** and the State interface as a State **role**

The **RateStrategy** interface serves the **strategy role** in the Strategy pattern, as well as the **state role** in the State pattern

A concrete instance of LinearRateStrategy may play **both** the Strategy:ConcreteStrategy and State:ConcreteState roles

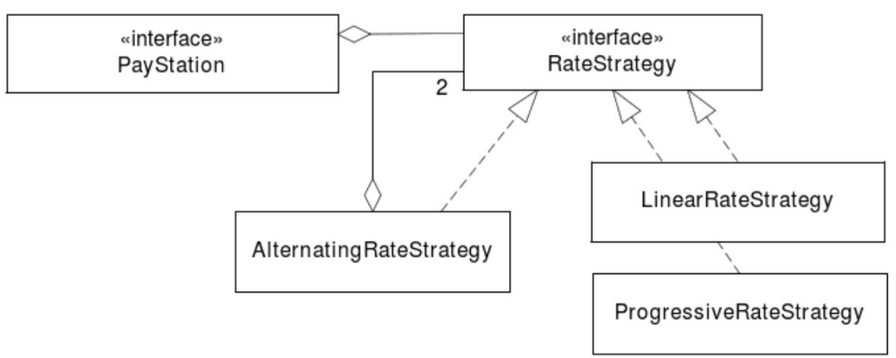
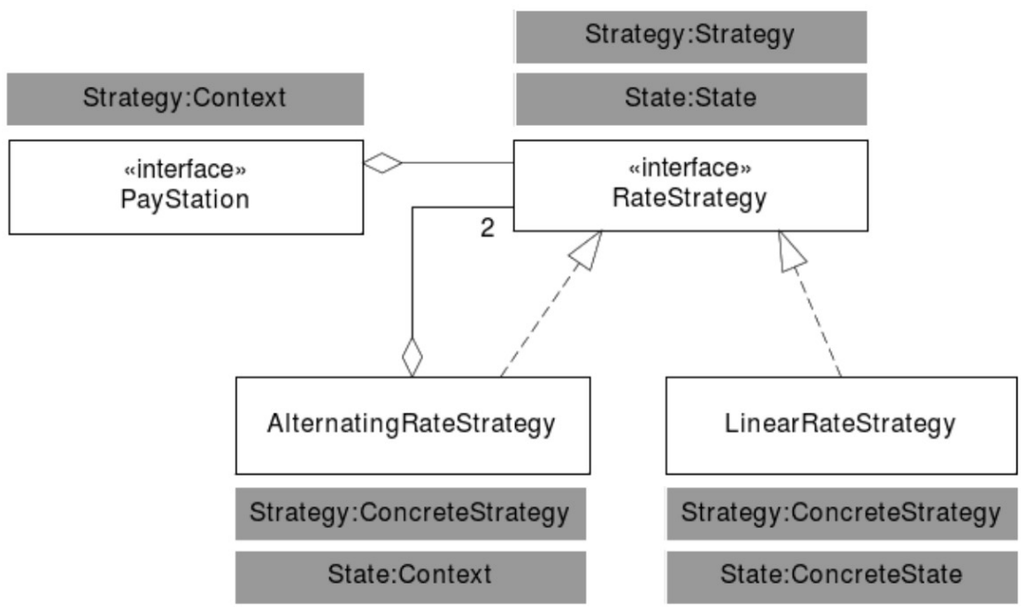


Figure 18.3: The combination of STRATEGY and STATE.

Design Patterns as Roles

Definition: Role diagram

A role diagram is a UML class diagram in which gray boxes either above or below each interface or class describe the abstraction's role in a particular pattern. The role is described by **pattern-name:role-name**.



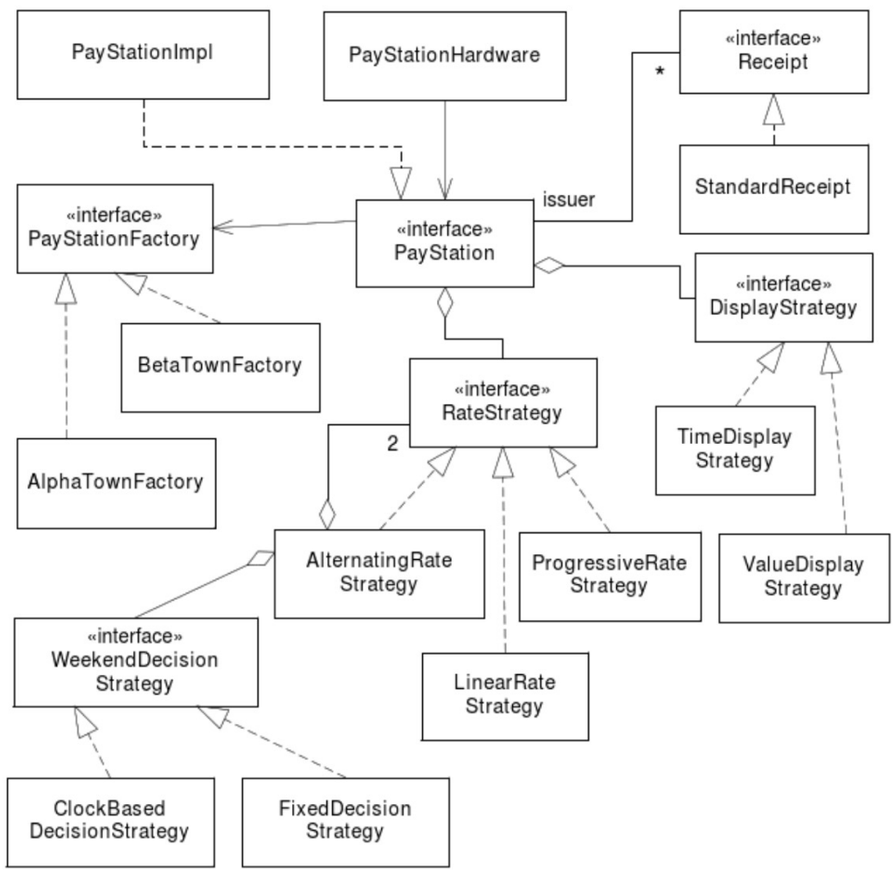
The **RateStrategy** interface serves the **strategy role** in the Strategy pattern, as well as the **state role** in the State pattern

A concrete instance of LinearRateStrategy may play **both** the Strategy:ConcreteStrategy and State:ConcreteState roles

Maintaining Compositional Designs

The pay station compositional design is **flexible** – but is it maintainable?

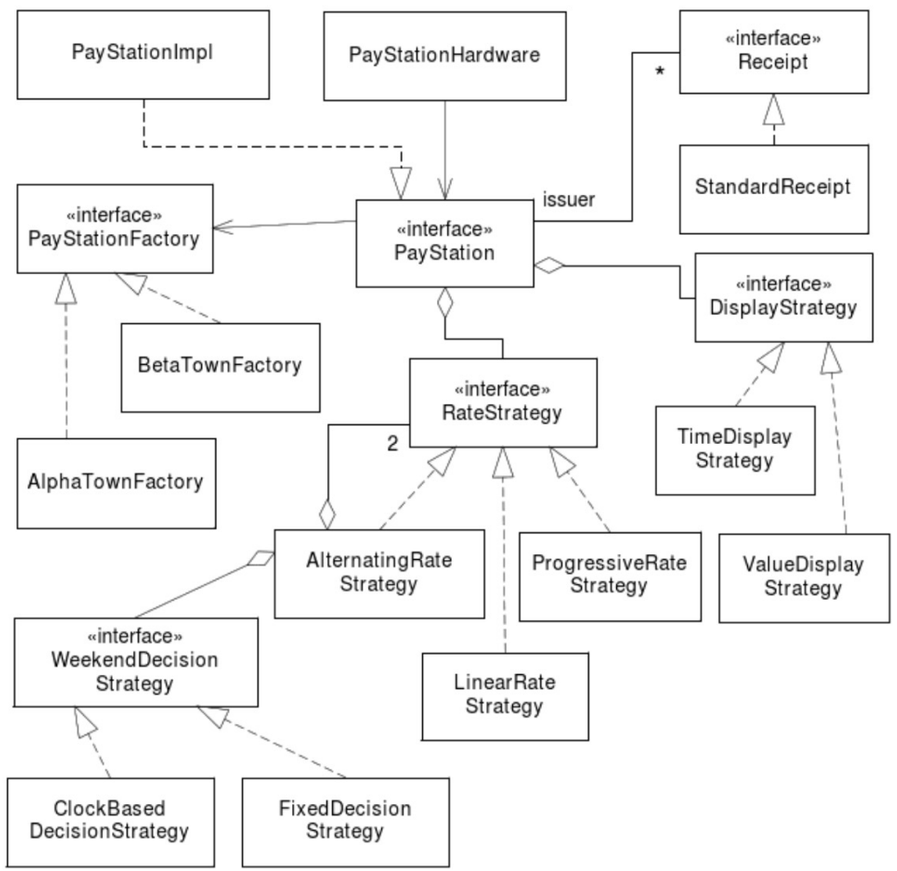
Recall: Maintainability is the capability of the software product to be **modified** (corrected, improved, adapted)



Maintaining Compositional Designs

The pay station compositional design is **flexible** – but is it maintainable?

Recall: Maintainability is the capability of the software product to be **modified** (corrected, improved, adapted)



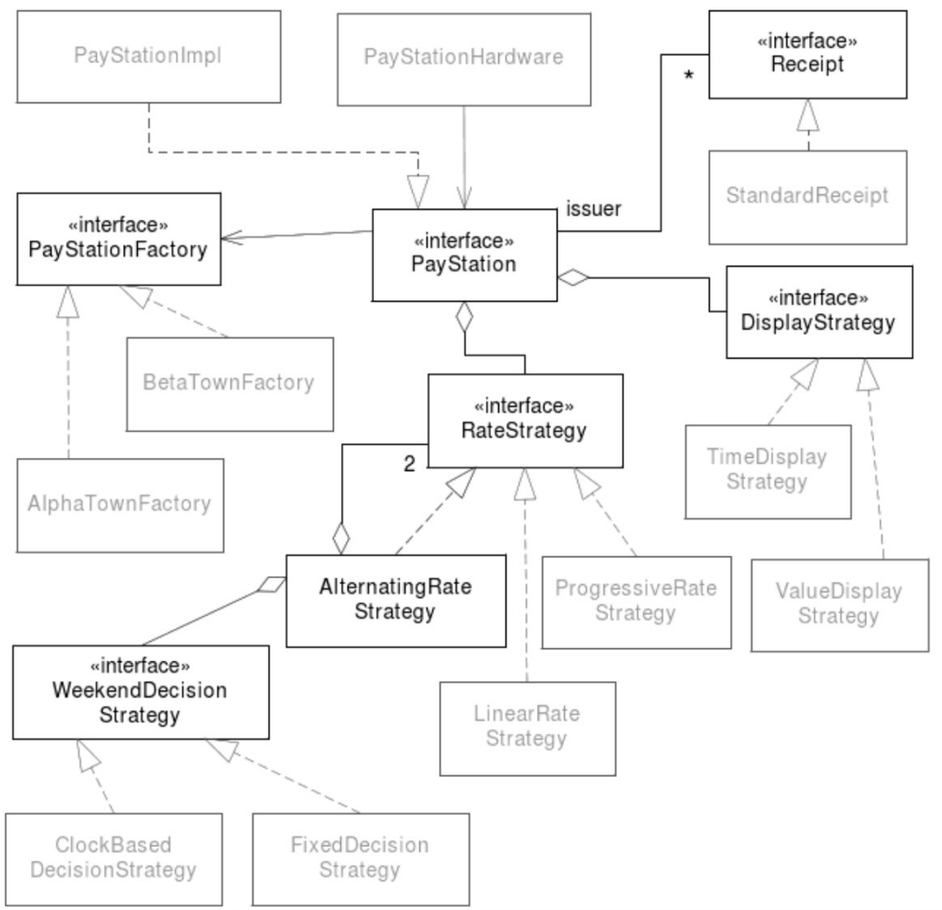
- This is a complex design!
- How would we communicate this design to new developers? Make them take this class?

Maintaining Compositional Designs

Solution: Know our patterns, and **document** them in our design

Most importantly, document **central roles:** State, Strategy, Factory

(Concrete implementations are implied)



Maintaining Compositional Designs

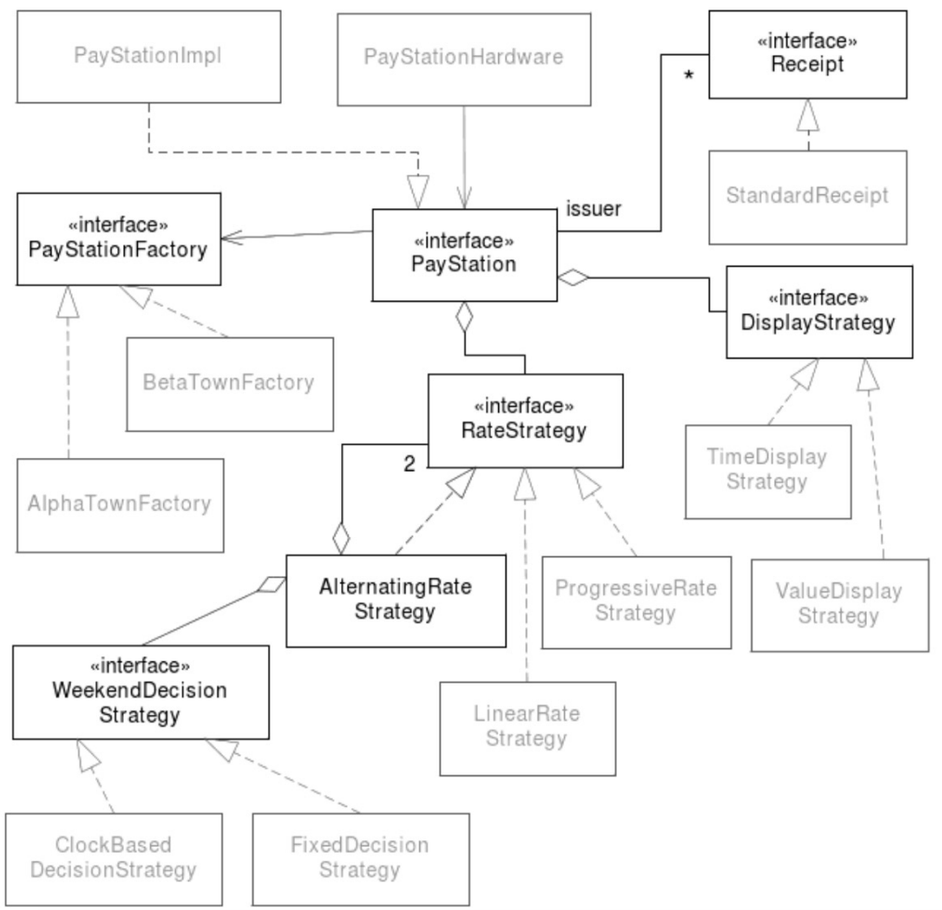
Solution: Know our patterns, and **document** them in our design

Most importantly, document **central roles:** State, Strategy, Factory

(Concrete implementations are implied)

Definition: Design pattern (Roadmap view)

Design patterns structure, document, and provide overview of the roles and protocols in complex, compositional, designs. A design pattern serves as a roadmap of a part of the design.



Maintaining Compositional Designs

Solution: Know our patterns, and **document** them in our design

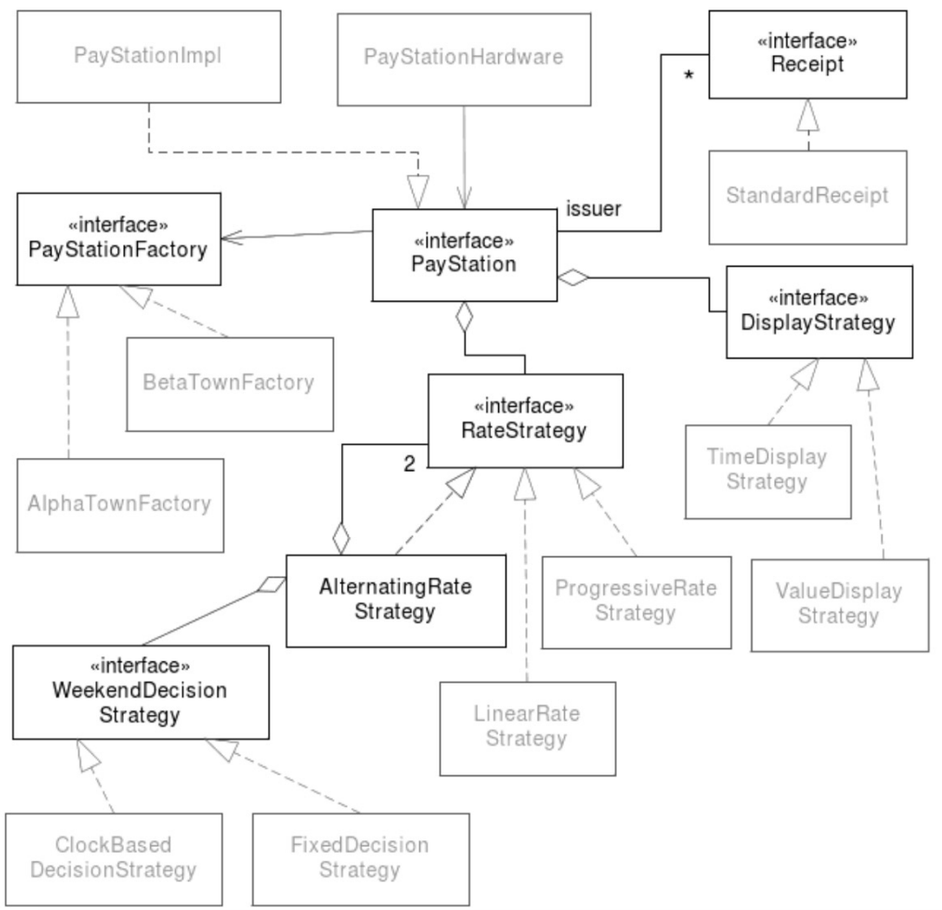
Most importantly, document **central roles:** State, Strategy, Factory

(Concrete implementations are implied)

Definition: Design pattern (Roadmap view)

Design patterns structure, document, and provide overview of the roles and protocols in complex, compositional, designs. A design pattern serves as a roadmap of a part of the design.

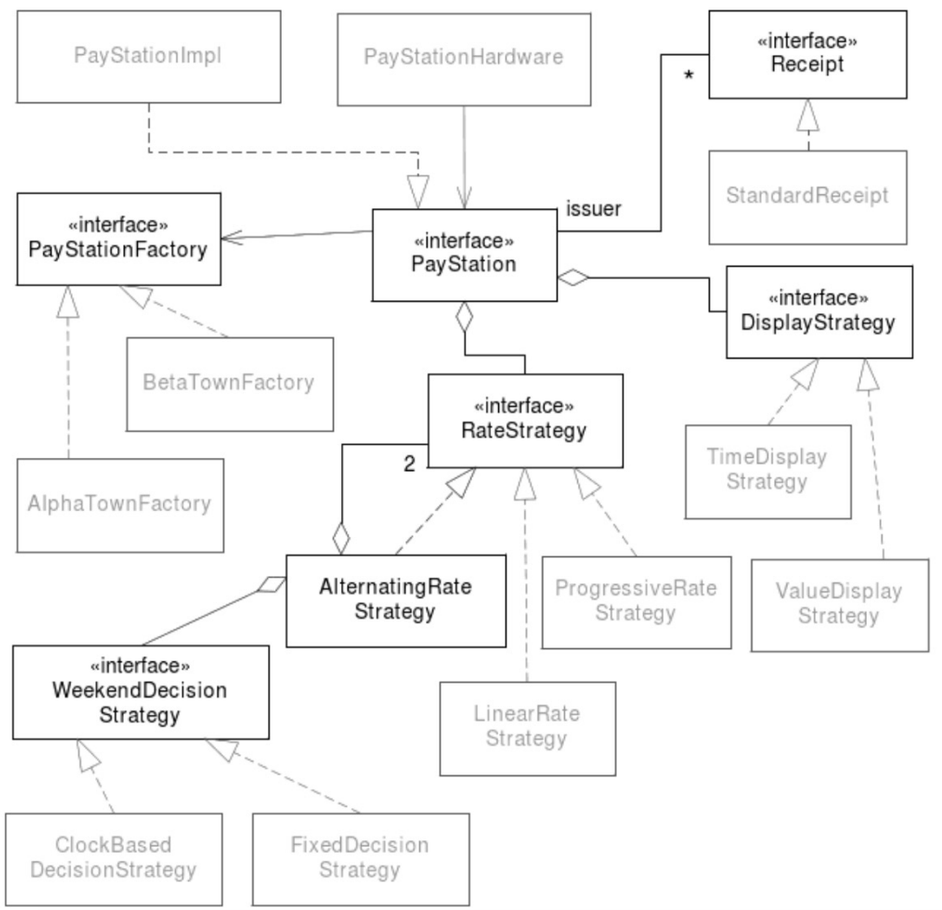
Developers must be trained in patterns to understand and maintain complex designs (even good ones!)



Maintaining Compositional Designs

"Is Design Dead?" – Martin Fowler
https://martinfowler.com/articles/designDead.html

- Invest time in learning about patterns
- Concentrate on when to apply the pattern (not too early)
- Concentrate on how to implement the pattern in its simplest form first, then add complexity later.
- If you put a pattern in, and later realize that it isn't pulling its weight - don't be afraid to take it out again.



Strategy Pattern Review: HotCiv

Refactor game setup to use concrete WinnerStrategy (using IntelliJ features):

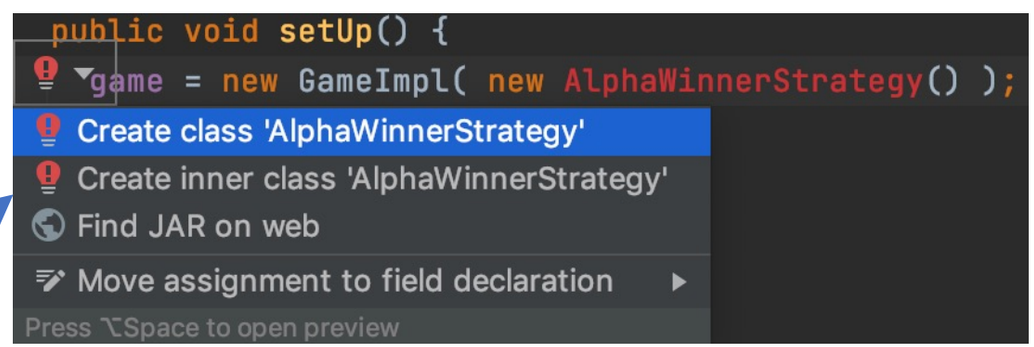
GameImpl

```
private WinnerStrategy winnerStrategy;

1 related problem
public GameImpl( WinnerStrategy winnerStrategy ) {
  this.winnerStrategy = winnerStrategy;
```

TestAlphaCiv

```
@Before
public void setUp() {
  game = new GameImpl( new AlphaWinnerStrategy() );
}
```

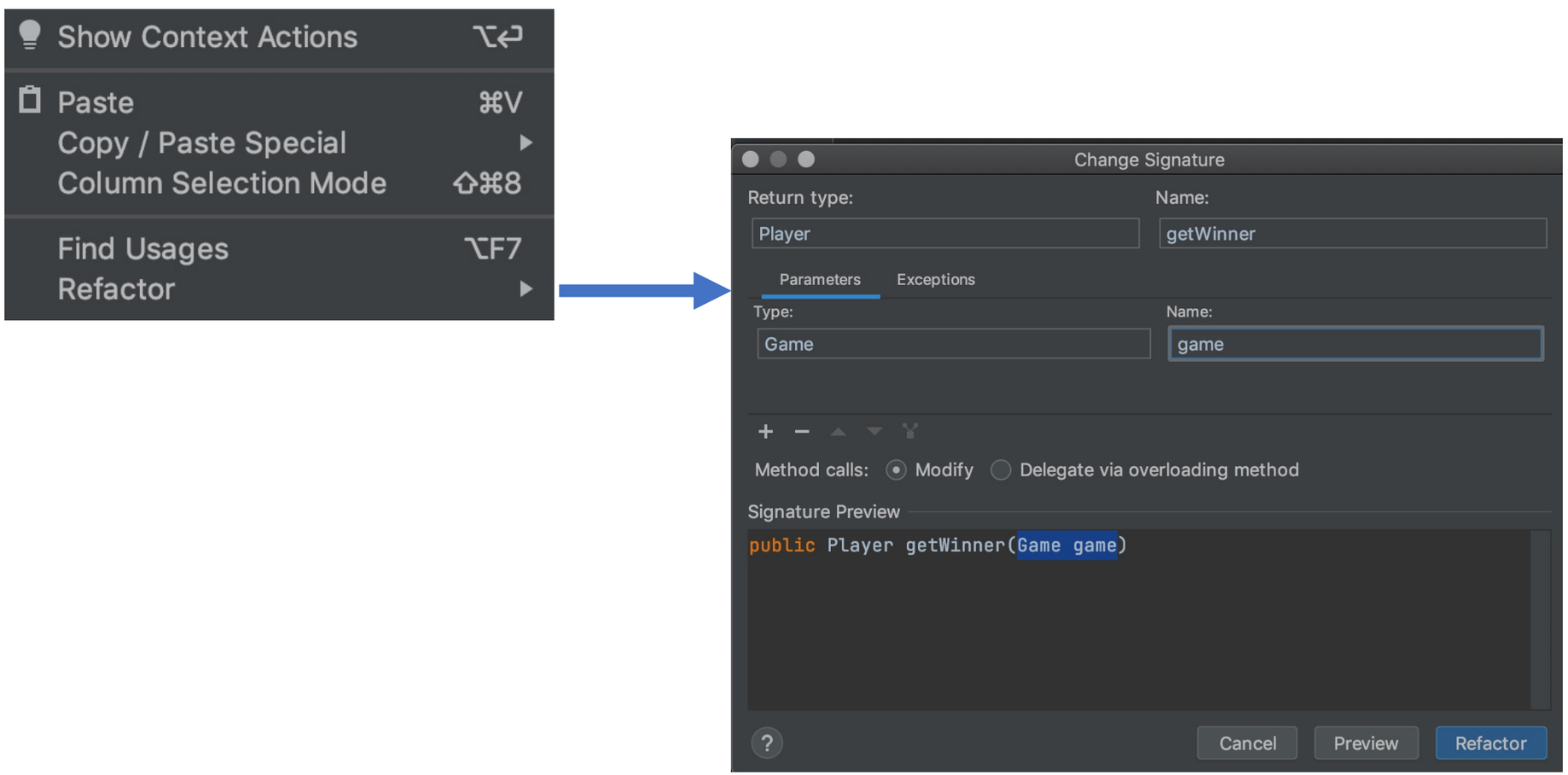


AlphaWinnerStrategy

```
public class AlphaWinnerStrategy implements WinnerStrategy {
    public Player getWinner(int age) {
        return null;
    }
}
```

Strategy Pattern Review: HotCiv

Refactoring in IntelliJ



Strategy Pattern Review: HotCiv

Keep GameImpl, UnitImpl, CityImpl, … **closed for modification**

- New concrete strategies can be created and passed to GameImpl constructor (or as a factory after Iteration 5)
- GameImpl should only use methods defined in strategy **interfaces**

```
public GameImpl(WinnerStrategy winnerStrategy, AgingStrategy agingStrategy,
                ActionStrategy actionStrategy, WorldLayoutStrategy worldLayoutStrategy) {
    this.winnerStrategy = winnerStrategy;
    this.agingStrategy = agingStrategy;
    this.actionStrategy = actionStrategy;
    this.worldLayoutStrategy = worldLayoutStrategy;

    // populate world with tiles, cities, units
    this.worldLayoutStrategy.createWorld( game: this);
}
```

```
public Player getWinner() {
    return winnerStrategy.getWinner( game: this);
}
```

Strategy Pattern Review: HotCiv

Keep GameImpl, UnitImpl, CityImpl, … **closed for modification**

- New concrete strategies can be created and passed to GameImpl constructor (or as a factory after Iteration 5)
- GameImpl should only use methods defined in strategy **interfaces**

```
public GameImpl(WinnerStrategy winnerStrategy, AgingStrategy agingStrategy,
                ActionStrategy actionStrategy, WorldLayoutStrategy worldLayoutStrategy) {
    this.winnerStrategy = winnerStrategy;
    this.agingStrategy = agingStrategy;
    this.actionStrategy = actionStrategy;
    this.worldLayoutStrategy = worldLayoutStrategy;

    // populate world with tiles, cities, units
    this.worldLayoutStrategy.createWorld( game: this);
}
```

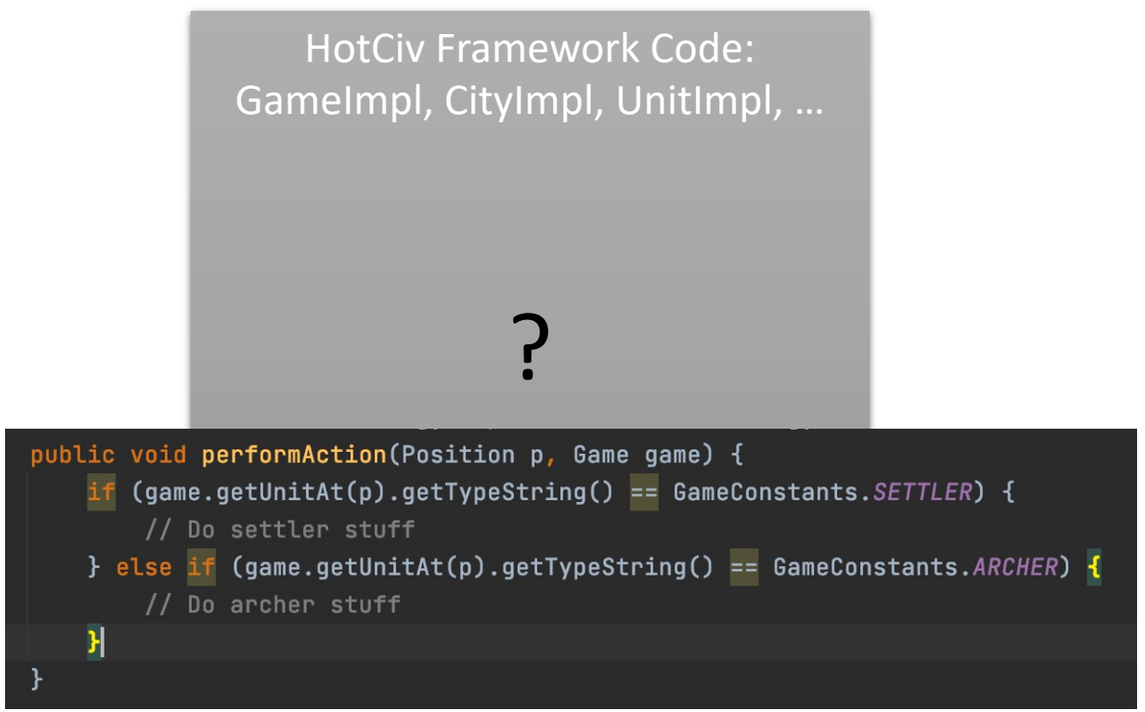
Lots of parameters, will be refactored to use factory

```
public Player getWinner() {
    return winnerStrategy.getWinner( game: this);
}
```

Strategy Pattern Review: HotCiv

Keep GameImpl, UnitImpl, CityImpl, … **closed for modification**

- Keep all variability switching code in **delegates** (not in the framework implementations, e.g., GameImpl)



Strategy Pattern Review: HotCiv

Keep GameImpl, UnitImpl, CityImpl, … **closed for modification**

- Keep all variability switching code in **delegates** (not in the framework implementations, e.g., GameImpl)

HotCiv Framework Code:
GameImpl, CityImpl, UnitImpl, …

?

```
public void performAction(Position p, Game game) {
    if (game.getUnitAt(p).getTypeString() == GameConstants.SETTLER) {
        // Do settler stuff
    } else if (game.getUnitAt(p).getTypeString() == GameConstants.ARCHER) {
        // Do archer stuff
    }
}
```

Bad! Now HotCiv has hard bindings to specific unit types.

Strategy Pattern Review: HotCiv

Keep GameImpl, UnitImpl, CityImpl, … **closed for modification**

- Keep all variability switching code in **delegates** (not in the framework implementations, e.g., GameImpl)

Strategy Pattern Review: HotCiv

Keep GameImpl, UnitImpl, CityImpl, … **closed for modification**

- Keep all variability switching code in **delegates** (not in the framework implementations, e.g., GameImpl)

Good! Avoids binding in GameImpl, concentrates GammaCiv logic in a clearly-named module (cohesive)

Strategy Pattern Review: HotCiv

Keep GameImpl, UnitImpl, CityImpl, … **closed for modification**

- Keep all variability switching code in **delegates** (not in the framework implementations, e.g., GameImpl)

HotCiv Framework Code:
GameImpl, CityImpl, UnitImpl, …

```
// age world
this.age = agingStrategy.advanceAge(this.age);
```

Keep game state information in Game (e.g., age)
Keep strategy pattern focused on algorithms

Strategy Pattern Review: HotCiv

Don’t overcomplicate, but triangulate when needs arise

Example: archer fortification - is this a *general feature* or GammaCiv *specific?*

```
public void performAction(Position p, Game game) {
    UnitImpl thisUnit = (UnitImpl)game.getUnitAt(p);
    if (thisUnit.getTypeString().equals(GameConstants.SETTLER)) {
        GameImpl theGame = (GameImpl)game;
        theGame.setCityAt(p, new CityImpl(thisUnit.getOwner()));
        theGame.clearUnitAt(p);
    } else if (thisUnit.getTypeString().equals(GameConstants.ARCHER)) {
        if (thisUnit.isMoveable()) {
            thisUnit.setDefensiveStrength(2 * thisUnit.getDefensiveStrength());
        } else {
            thisUnit.setDefensiveStrength((int)(0.5 * thisUnit.getDefensiveStrength()));
        }
        thisUnit.toggleMoveable();
    }
}
```

Changing unit moveability seems general

- Accessor isMoveable
- Mutator in implementation
- Check moveability in GameImpl moveUnit

Strategy Pattern Review: HotCiv

How to share/exchange state information among objects?

Example: Aging strategies require knowledge of game state (age or city owners)

Easy enough to pass “age”, but what about winning states that require more information from Game?

Strategy Pattern Review: HotCiv

How to share/exchange state information among objects?

Add methods to Game interface? (e.g., getOwners())

- Bloats the interface
- Lowers cohesion, responsibility erosion
- Only needed for BetaCiv

Strategy Pattern Review: HotCiv

How to share/exchange state information among objects?

Add methods to GameImpl (implementation)?

- Want to only pass an interface to getWinner, so requires casting to GameImpl in the strategy
- What if GameImpl isn't the only Game implementation?

Strategy Pattern Review: HotCiv

How to share/exchange state information among objects?

Use only Game interface methods?

- Iterate over every position, try getCity(), then getOwner()
- Seems overcomplicated

Passing the whole game works, but gives the strategies access to a lot of things they don't need…

Strategy Pattern Review: HotCiv

Pass the whole game?

GameImpl

```
public Player getWinner() {
    return winnerStrategy.getWinner( game: this);
}
```

This works, but gives the strategies access to a lot of things they don't need…

Passing Context Information

One possible solution: Context objects

```
public interface WinnerStrategy {
  public Player getWinner(WinnerStrategyContext context);
}
```

```
public interface WinnerStrategyContext {
  public int getAge();
  public Collection<Player> getOwners();
}
```

Passing Context Information

One possible solution: Context objects

```
public interface WinnerStrategy {
  public Player getWinner(WinnerStrategyContext context);
}
```

```
public interface WinnerStrategyContext {
  public int getAge();
  public Collection<Player> getOwners();
}
```

```
public class GameImpl implements Game {
  ...
  public Player getWinner() {
    return _winnerStrategy.getWinner(new WinnerContext() {
          public int getAge() {
            return GameImpl.this.getAge();
          }

          public Collection<Player> getOwners() {
            ArrayList<Player> result = new ArrayList<Player>();
            result .add(_redCity.getOwner());
            result .add(_blueCity.getOwner());
            return result;
          }
        });
  }
  ...
}
```

Passing Context Information

One possible solution: Context objects

```
public interface WinnerStrategyContext {
  public int getAge();
  public Collection<Player> getOwners();
}
```

```
public interface WinnerStrategy {
  public Player getWinner(WinnerStrategyContext context);
}
```

```
public class GameImpl implements Game {
  ...
  public Player getWinner() {
    return _winnerStrategy.getWinner(new WinnerContext() {
          public int getAge() {
            return GameImpl.this.getAge();
          }

          public Collection<Player> getOwners() {
            ArrayList<Player> result = new ArrayList<Player>();
            result .add(_redCity.getOwner());
            result .add(_blueCity.getOwner());
            return result;
          }
        });
  }
  ...
}
```

```
public class AlphaCivWinnerStrategy implements WinnerStrategy {
  private final int WINNING_AGE = -3000;

  @Override
  public Player getWinner(WinnerStrategyContext context) {
    if(context.getAge() >= WINNING_AGE)
      return Player.RED;
    return null;
  }
}
```

```
public class BetaCivWinnerStrategy implements WinnerStrategy {

  @Override
  public Player getWinner(WinnerContext context) {
      Player candidate = null;
      for(Player owner: context.getOwners()) {
          if(candidate == null)
              candidate = owner;
          if (owner != candidate)
              return null;
      }
      return candidate;
  }
}
```

Passing Context Information

One possible solution: Context objects

```
public interface WinnerStrategyContext {
  public int getAge();
  public Collection<Player> getOwners();
}
```

```
public interface WinnerStrategy {
  public Player getWinner(WinnerStrategyContext context);
}
```

```
public class GameImpl implements Game {
  ...
  public Player getWinner() {
    return _winnerStrategy.getWinner(new WinnerContext() {
          public int getAge() {
            return GameImpl.this.getAge();
          }

          public Collection<Player> getOwners() {
            ArrayList<Player> result = new ArrayList<Player>();
            result .add(_redCity.getOwner());
            result .add(_blueCity.getOwner());
            return result;
          }
        });
  }
  ...
}
```

Iteration 5:

Player wins after a certain number of attacks?

Strategy changes after a certain number of rounds?

Passing Context Information

One possible solution: Context objects

```
public interface WinnerStrategy {
  public Player getWinner(WinnerStrategyContext context);
}
```

```
public interface WinnerStrategyContext {
  public int getAge();
  public Collection<Player> getOwners();
}
```

```
public class GameImpl implements Game {
  ...
  public Player getWinner() {
    return _winnerStrategy.getWinner(new WinnerContext() {
          public int getAge() {
            return GameImpl.this.getAge();
          }

          public Collection<Player> getOwners() {
            ArrayList<Player> result = new ArrayList<Player>();
            result .add(_redCity.getOwner());
            result .add(_blueCity.getOwner());
            return result;
          }
        });
  }
  ...
}
```

Iteration 5:

Player wins after a certain number of attacks?
Strategy changes after a certain number of rounds?

Next time: Lots of patterns