Pattern Fragility

Several ways to destroy Strategy

Advice

Forbidden city, Beijing.

Why Patterns?

Design patterns organize and structure code in a particular way.

- Static: Arrangement of classes/interfaces
- Dynamic: Assignment of responsibility, interaction patterns

Why:

- Because I get some benefits from doing so

Bottom line:

- Patterns are *means to a goal, not the goal itself*

Patterns Are Code

However, patterns are defined in code, and if I code wrong I "amputate" the pattern. I get all the liabilities and none of the advantages.

Pattern Fragility	Pattern fragility is the property of design patterns that their benefits can only be fully utilized if the pattern's object structure and protocol is implemented correctly.

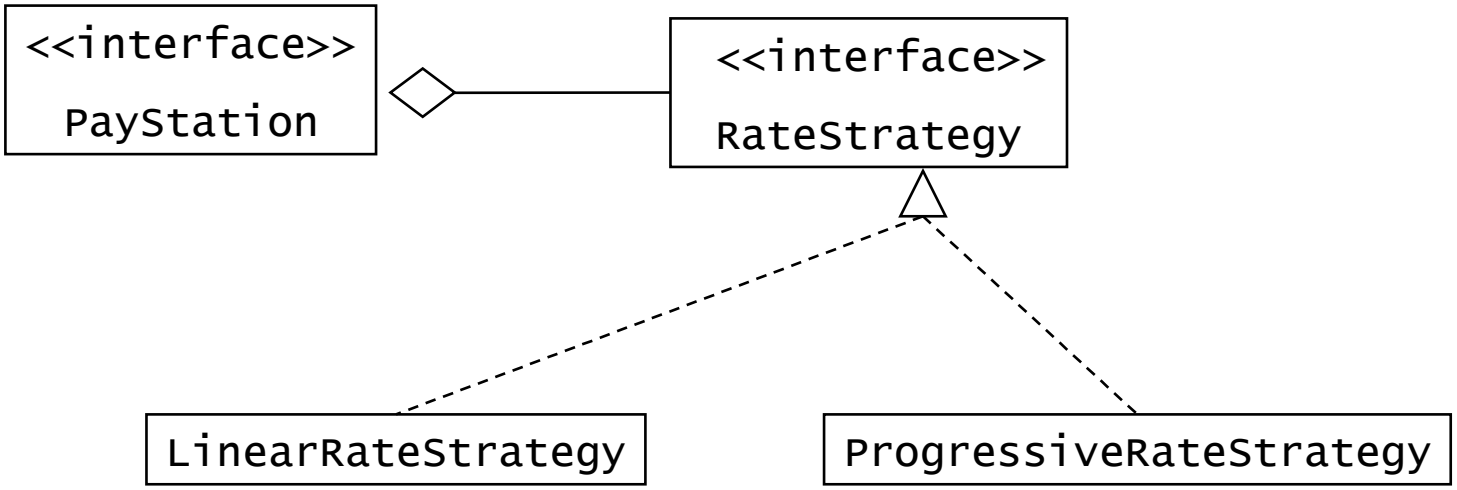
Warstory:

- COT case: Reusable search component's deadline was forced. Additional staff added. A design pattern based, highly decoupled, design was utterly destroyed in a week.

Example: Strategy

Strategy

Responsibilities must be served by concrete behaviour in objects...



Pitfall 1: Declaration of Delegates

1. Do not even think of using class names in declarations!

Why is the following change a disaster

```
public class PayStationImpl implements PayStation {
  [...]

  /** the strategy for rate calculations */
  private ProgressiveRateStrategy rateStrategy;

  [...]
}
```

Proverb 1

Declare object references that play part in a design pattern by their interface type, never by their concrete class type.

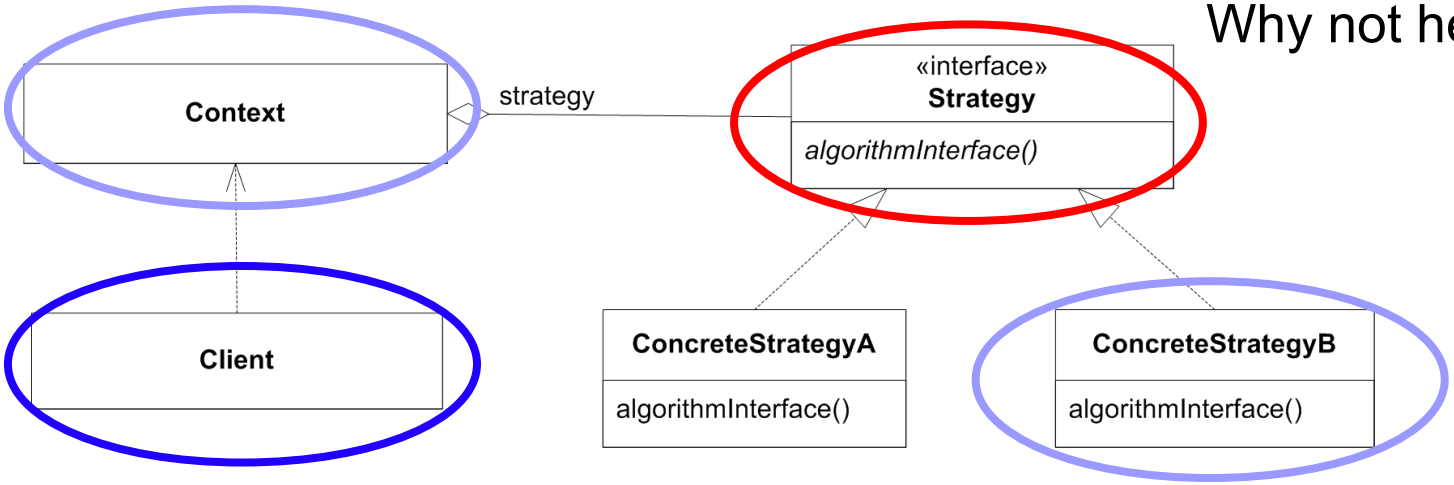
Pitfall 2: Binding in the Right Place

Loose coupling is fine, but we have to couple the objects together eventually.

It is important that the binding is made

- *in the right place*
- *as few places as possible (optimally 1!)*

Many possibilities for Strategy:



Binding in the Wrong Place

Binding in the Context object:

```
public class PayStationImpl implements PayStation {
  [...]
  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;
    RateStrategy rateStrategy = new LinearRateStrategy();
    timeBought = rateStrategy.calculateTime(insertedSoFar);
  }
  [...]
}
```

Will *not* break any tests for Alphatown!

Consequence

What is the consequence?

I got *all* the pattern's liabilities

- more interfaces and classes and objects to overview

And *none* of the pattern's benefits

- high coupling
- no variability at all!

Advice

Proverb

Object should be created and coupled in the production code units whose responsibility are explicitly configuration and binding.

In Strategy, this is normally the Client role.

Note again:

- Automated tests that test the full suite of products *will* detect this defect.
- A manual testing effort much focused on a specific product variant will probably not...

Relation to Other Patterns

Abstract Factory is a *creational pattern*. Its purpose in life is to define bindings. Thus, the factory is often *the right place* to make bindings.

In State it is actually often the ConcreteState objects that define the ‘next state’ of the state machine. Thus it is more common that ConcreteState objects directly create state objects.

Concealed Parameterization

Assume: Previous binding survived.

Later: *"Why does Betatown not work any more?*

I need to fix it, and fix it fast!"

```
public class PayStationImpl implements PayStation {
  [...]
  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;
    RateStrategy rateStrategy;
    if ( town == Town.ALPHATOWN ) {
      rateStrategy = new LinearRateStrategy();
    } else if ( town == Town.BETATOWN ) {
      rateStrategy = new ProgressiveRateStrategy();
    }
    timeBought = rateStrategy.calculateTime(insertedSoFar);
  }
  [...]
}
```

Advice

Proverb:

Decide on a design strategy to handle a given variability and stick to it.

Responsibility Erosion

Software changes its own requirement.

New (weird) request

– Gammatown: Explain rate policy.

```
public class AlternatingRateStrategy implements RateStrategy {
  [...]
  public int calculateTime( int amount ) {
    if ( decisionStrategy.isWeekend() ) {
      currentState = weekendStrategy;
    } else {
      currentState = weekdayStrategy;
    }
    return currentState.calculateTime( amount );
  }

  public String explanationText() {
    if ( currentState == weekdayStrategy ) {
      return [the explanation for weekday];
    } else {
      return [the explanation for weekend];
    }
  }
}
```

Consequences

Now, however, this strategy does not conform to the contract by the interface.

```
if ( rateStrategy instanceof AlternatingRateStrategy ) {
  AlternatingRateStrategy rs =
    (AlternatingRateStrategy) rateStrategy;
  String theExplanation = rs.explanationText();
  [use it somehow]
}
```

Solution: Move the method up into the RateStrategy interface.

But: I have now added a new responsibility. One that may not be very cohesive.

Advice

Proverb:

Carefully analyse new requirements to avoid responsibility erosion and bloating interfaces with in-cohesive methods.

The polymorphic wrapping trap

From the mandatory hand-in

```
public class AlphaCivGame implements Game {...}

public class BetaCivGame extends AlphaCivGame{

    public BetaCivGame(){
        super();
    }

    @Override
    public Player getWinner() {
        return new BetaWinnerStrategy().getWinner(this);
    }

}
```

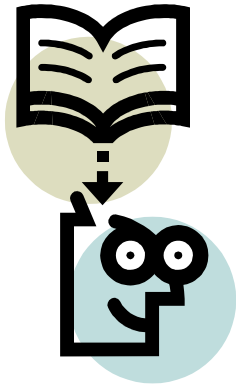
Summary

Take care at the implementation level!!!

It only takes a few “slip-ups” to completely destroy the intended benefits of a pattern!

Corollary: You do not learn patterns by reading a book or listening to me!

CODE! and reflect!



Summary

All your programmers must deeply understand the roles and protocols embodied in design patterns in order to keep the design intact.