Lecture 06

ECE 1145: Software Construction and Evolution

Design Patterns Introduction

Strategy Pattern

(CH 7, 8, 9)

Announcements

- Iteration 2 due Sept. 26
- Relevant Exercises: **7.5** 7.6 9.1 9.3
- Starcraft patterns: https://www.youtube.com/playlist?list=PL6A29DA14366FF6C3
    - Strategy pattern: https://www.youtube.com/watch?v=MOEsKHqLiBM
- Design patterns book (go to “Sign in” and enter your Pitt email for free access): https://learning.oreilly.com/library/view/design-patterns-elements/0201633612/

Questions for Today

How do we handle variability in software?

How do we apply patterns to design software?

Review

Case: PayStation

* ~~accept legal coin~~
* ~~reject illegal coin, exception~~
* ~~5 cents should give 2 minutes parking time.~~
* ~~readDisplay~~
* ~~buy for 40 cents produces valid receipt~~
* ~~cancel resets pay station~~
* ~~25 cents = 10 minutes~~
* ~~enter a 10 and 25 coin~~
* ~~receipt can store values~~
* ~~buy for 100 cents~~
* ~~clearing after a buy operation~~

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

  public void addPayment( int coinValue )
          throws IllegalCoinException {
    switch ( coinValue ) {
    case 5: break;
    case 10: break;
    case 25: break;
    default:
      throw new IllegalCoinException("Invalid coin: "+coinValue);
    }
    insertedSoFar += coinValue;
    timeBought = insertedSoFar / 5 * 2;
  }
  public int readDisplay() {
    return timeBought;
  }
  public Receipt buy() {
    Receipt r = new ReceiptImpl(timeBought);
    reset();
    return r;
  }
  public void cancel() {
    reset();
  }
  private void reset() {
    timeBought = insertedSoFar = 0;
  }
}
```

The Nightmare: Success!

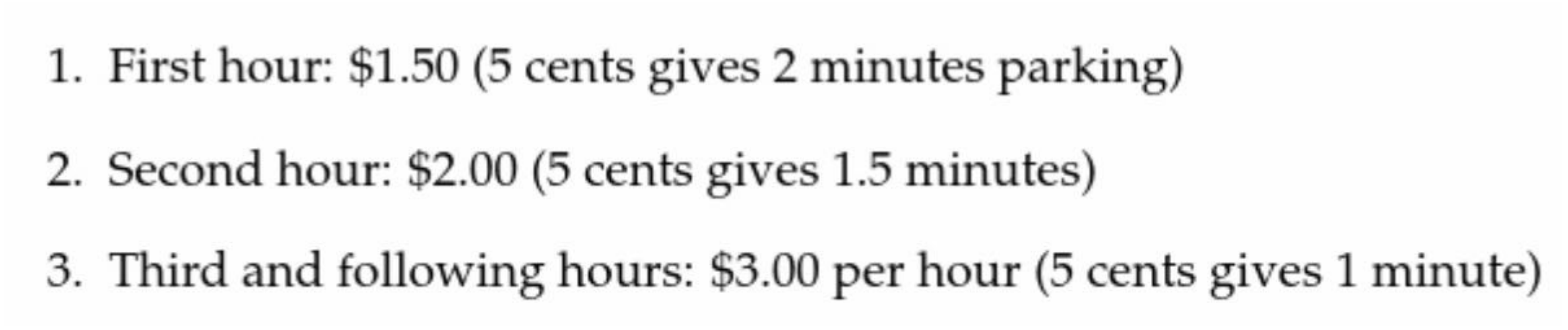
AlphaTown is very satisfied!

Word has spread far and wide of our reliable software!

Now BetaTown has inquired about our parking machine software – but, they have a special request…

BetaTown: New Customer Requirements!

Progressive price model



1. First hour: \$1.50 (5 cents gives 2 minutes parking)
2. Second hour: \$2.00 (5 cents gives 1.5 minutes)
3. Third and following hours: \$3.00 per hour (5 cents gives 1 minute)

Maybe we will have more requests for different pricing models ???

How can we handle these two products (and future variants)?

Variability

Problem (formulation 1): We need to develop and maintain two variants of the software system in a way that introduces the least cost and defects.

```
public void addPayment( int coinValue )
        throws IllegalCoinException {
  switch ( coinValue ) {
  case 5: break;
  case 10: break;
  case 25: break;
  default:
    throw new IllegalCoinException("Invalid coin: "+coinValue);
  }
  insertedSoFar += coinValue;
  timeBought = insertedSoFar / 5 * 2;
}
```

Variability

Problem (formulation 1): We need to develop and maintain two variants of the software system in a way that introduces the least cost and defects.

```
public void addPayment( int coinValue )
        throws IllegalCoinException {
  switch ( coinValue ) {
  case 5: break;
  case 10: break;
  case 25: break;
  default:
    throw new IllegalCoinException("Invalid coin: "+coinValue);
  }
  insertedSoFar += coinValue;
  timeBought = insertedSoFar / 5 * 2;
}
```

Definition: Variability point

A variability point is a well defined section of the production code whose behavior it should be possible to vary.

Variability

Problem (formulation 2): How do I introduce having two different behaviors of the rate calculation variability point such that both the cost and the risk of introducing defects are low?

```java
public void addPayment( int coinValue )
        throws IllegalCoinException {
  switch ( coinValue ) {
  case 5: break;
  case 10: break;
  case 25: break;
  default:
    throw new IllegalCoinException("Invalid coin: "+coinValue);
  }
  insertedSoFar += coinValue;
  timeBought = insertedSoFar / 5 * 2;
}
```

Definition: Variability point

A variability point is a well defined section of the production code whose behavior it should be possible to vary.

Variability

How can we handle this?

Consider:

- Most of the code is the same in the two products
- What about **real** success? 20 product variants?

Variability

Model 1: **Copy**

- Make a copy of the source tree - one copy for AlphaTown, one for BetaTown

Model 2: **Parameterization**

- Throw in some ‘if’ statements to check whether AlphaTown or BetaTown

Model 3: **Polymorphism**

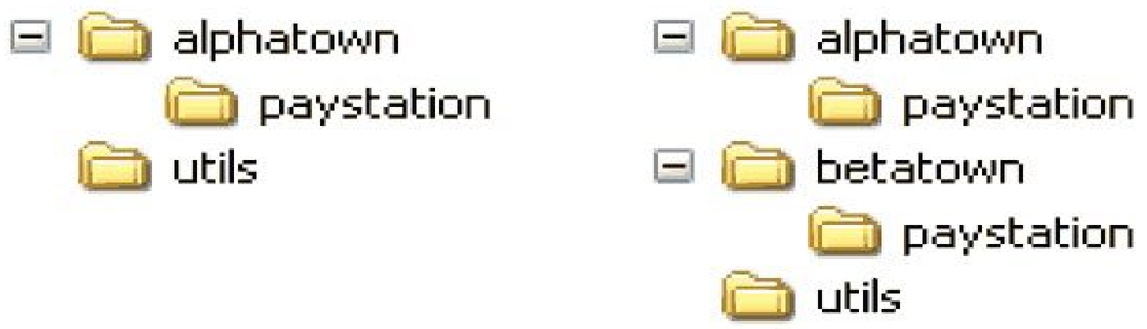
- Use inheritance, subclass PayStationImpl, override the rate calculation

Model 4: **Composition**

- Factor out rate calculation responsibility into an interface, create multiple concrete rate calculation classes, pay station delegates to the appropriate rate calculation object

Model 1: Copy Source Tree

Copy and paste!



Also copy test cases! And rewrite any tests that are dependent on the variability point.

Code the new variant by replacing the code at the variability point.

Model 1: Copy Source Tree

Pros:

- Simple!
  - No special skill set required in developer team
  - Easy to explain to new developers
- Fast!
  - Copy, paste, modify
- Decoupled!
  - Defects introduced in variant 2 does **not** reduce reliability of variant 1
  - Easy to distinguish variants

Cons:

- **Multiple maintenance problem**
  - Changes in common code must be propagated to all copies
- Example:
  - 4 pay station variants (different rate policies) = 4 source trees
  - Feature request: pay station keeps track of earnings ☹
  - Fix the same bug in nearly identical production code bases at the same time ☹
- Typically, variants drift apart, becoming different products instead of variants...

Model 1: Copy Source Tree

Pros:

- Simple!
  - No special skill set required in developer team
  - Easy to explain to new developers
- Fast!
  - Copy, paste, modify
- Decoupled!
  - Defects introduced in variant 2 does **not** reduce reliability of variant 1
  - Easy to distinguish variants

Cons:

- **Multiple maintenance problem**
  - Changes in common code must be propagated to all copies
- Example:
  - 4 pay station variants (different rate policies) = 4 source trees
  - Feature request: pay station keeps track of earnings ☹
  - Fix the same bug in nearly identical production code bases at the same time ☹
- Typically, variants drift apart, becoming different products instead of variants...

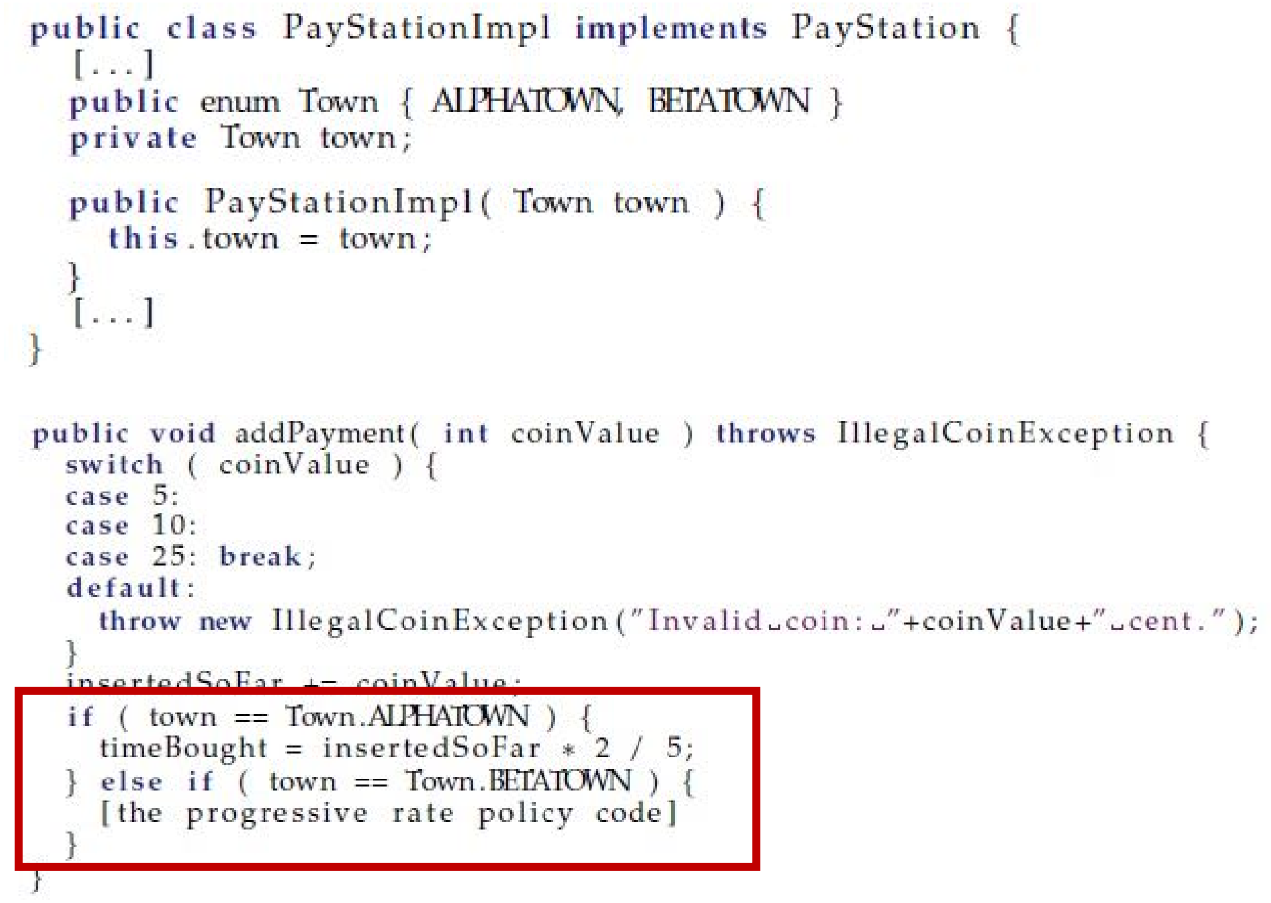
Summary: Quick but Dangerous! Gets worse at scale

Model 2: Parametric Solution

Variability point is in a single “behavioural unit” in the addPayment method.

Make a conditional statement there

- Introduce a “town” parameter
- Switch on the parameter each time town-specific behaviour is needed



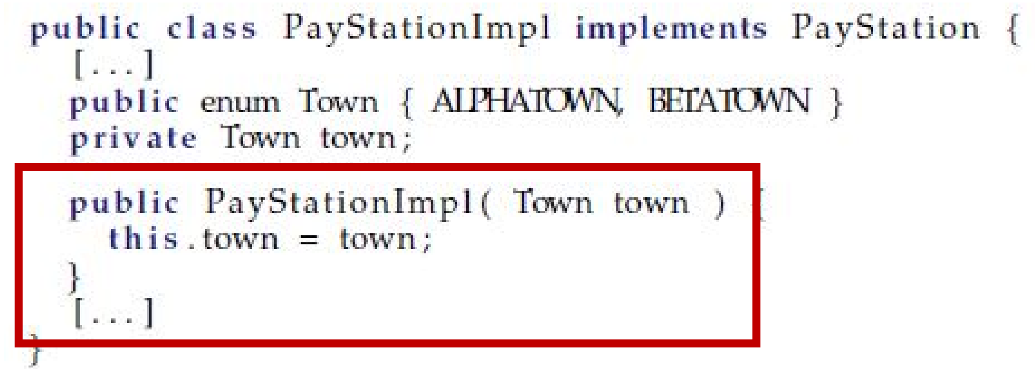
```
public class PayStationImpl implements PayStation {
  [...]
  public enum Town { ALPHATOWN, BETATOWN }
  private Town town;

  public PayStationImpl( Town town ) {
    this.town = town;
  }
  [...]
}

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;
  if ( town == Town.ALPHATOWN ) {
    timeBought = insertedSoFar * 2 / 5;
  } else if ( town == Town.BETATOWN ) {
    [the progressive rate policy code]
  }
}
```

Model 2: Parametric Solution

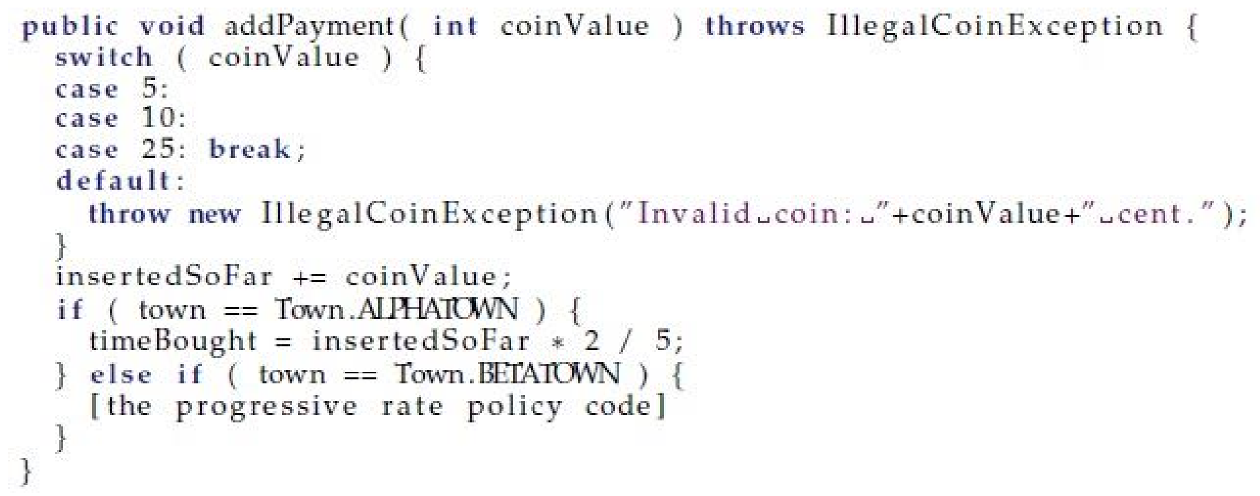
How can we specify which variant of the pay station to use?

One option: in the constructor parameters:

```
PayStation ps = new PayStationImpl(
Town.ALPHATOWN );
```



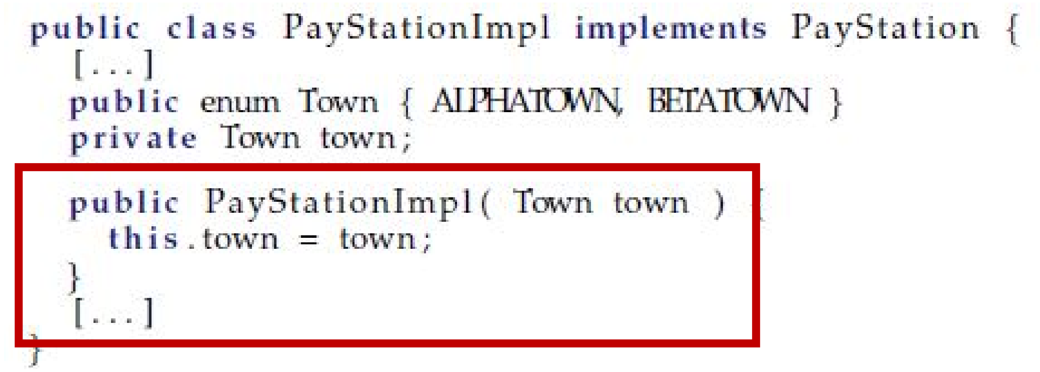
```
public class PayStationImpl implements PayStation {
  [...]
  public enum Town { ALPHATOWN, BETATOWN }
  private Town town;

  public PayStationImpl( Town town ) {
    this.town = town;
  }
  [...]
}
```



```
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;
  if ( town == Town.ALPHATOWN ) {
    timeBought = insertedSoFar * 2 / 5;
  } else if ( town == Town.BETATOWN ) {
    [the progressive rate policy code]
  }
}
```

Model 2: Parametric Solution

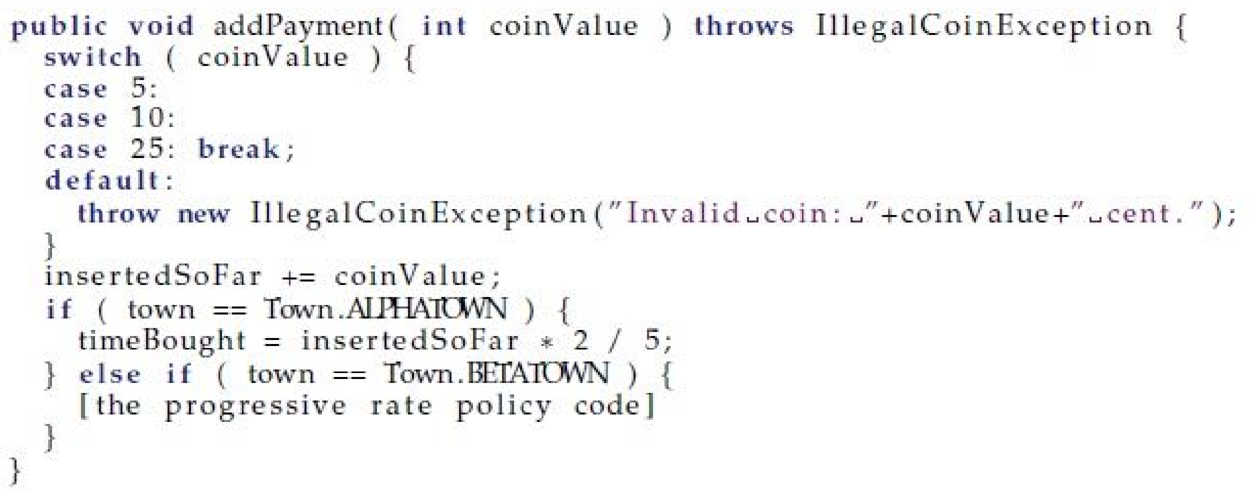
How can we specify which variant of the pay station to use?



```
public class PayStationImpl implements PayStation {
  [...]
  public enum Town { ALPHATOWN, BETATOWN }
  private Town town;

  public PayStationImpl( Town town ) {
    this.town = town;
  }
  [...]
}
```

One option: in the constructor parameters:

```
PayStation ps = new PayStationImpl(
Town.ALPHATOWN );
```



```
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;
  if ( town == Town.ALPHATOWN ) {
    timeBought = insertedSoFar * 2 / 5;
  } else if ( town == Town.BETATOWN ) {
    [the progressive rate policy code]
  }
}
```

Other options:

- Town is an input argument to addPayment?
- Town is defined in a property file?
- Town is defined in a database?
- Etc.

Model 2: Parametric Solution

Pros:

- Simple!
  - A conditional statement is easy to understand for any skill level developer team
- Avoids multiple maintenance problem!
  - One code base
  - Easier to fix defects in common behavior
  - Easier to add common features

Cons:

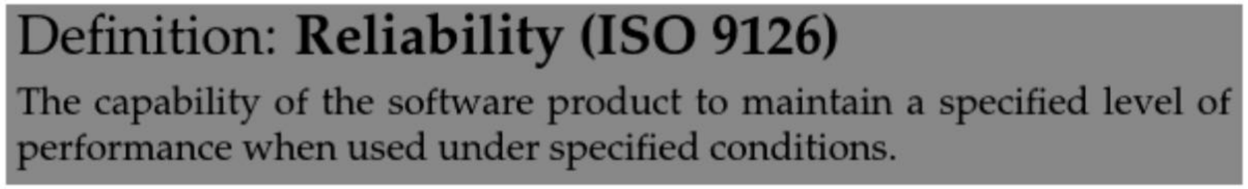
- Reliability concerns
- Analyzability concerns
- Responsibility erosion
- Composition problem

Model 2: Parametric Solution

Reliability/quality concerns

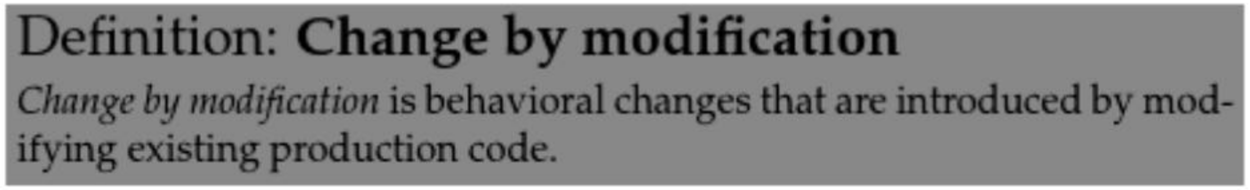
- Risk of **change by modification**
  - Don’t want to cause problems with AlphaTown!
  - Tests can help, but aren’t foolproof
- Need to modify the existing PayStationImpl for any new rate model (e.g., in a new town)
- Requires reviewing and re-running all tests of all product variants
- Gets worse with more conditionals!

Recall:



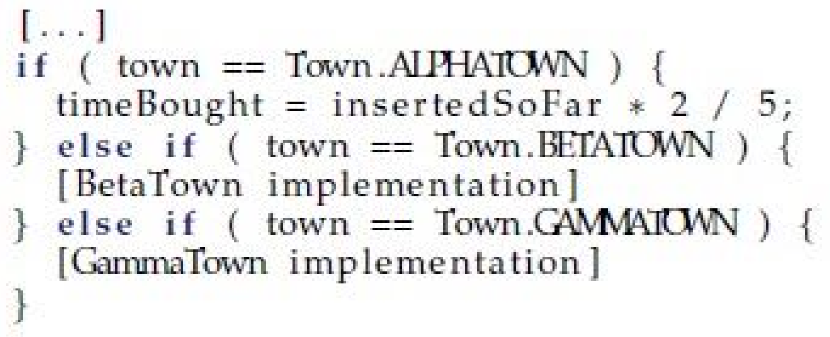
Definition: **Reliability (ISO 9126)**

The capability of the software product to maintain a specified level of performance when used under specified conditions.



Definition: **Change by modification**

Change by modification is behavioral changes that are introduced by modifying existing production code.



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

Model 2: Parametric Solution

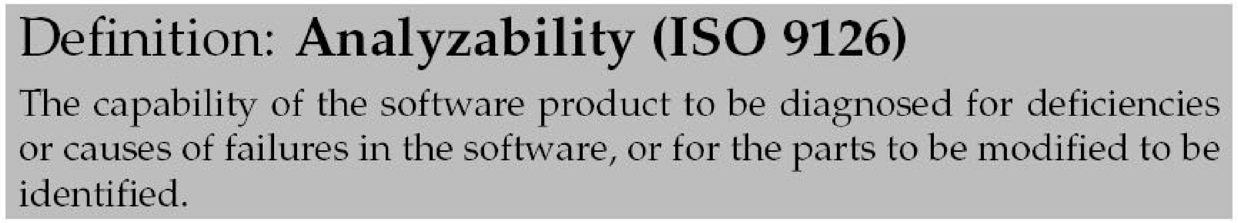
Analyzability concerns:
Code bloat

- With more variants, the switching code becomes longer and longer…

Switch creep

- One “if” often leads to more “if”s, worse with more variability dimensions

Recall:



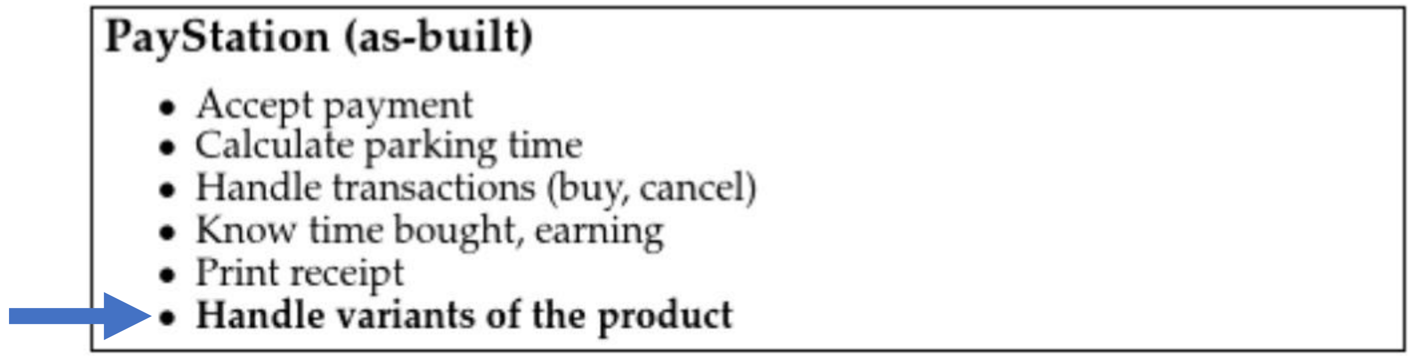
Definition: **Analyzability (ISO 9126)**

The capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified.

```
if ( Town == ALPHATOWN ) {
   if ( databaseServer == ORACLE && optimizingOn ){
      if ( DEBUG ) { System.out.println( "..." ); }
      ...
   } else { if ( isMobilePayment() ) {
      discountFactor = 0.9;
      XXX
} else { ... }
```

Model 2: Parametric Solution

Responsibility erosion (“feature creep”)



PayStation (as-built)

- Accept payment
- Calculate parking time
- Handle transactions (buy, cancel)
- Know time bought, earning
- Print receipt
- **Handle variants of the product**

Later … parsing XML files, printing debug statements in the console, updating a database, handling transactions over a network …

Beware “The Blob”

Model 2: Parametric Solution

Composition problem

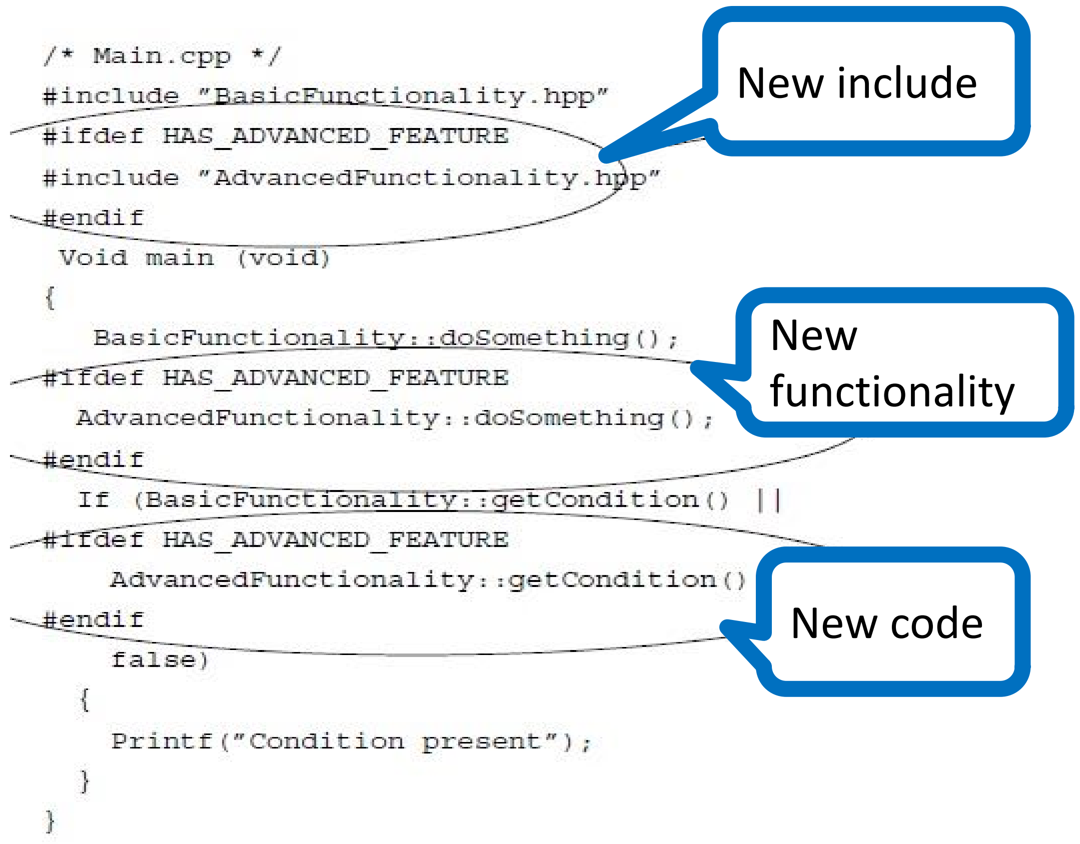
- A rate model that is a combination of existing ones leads to code duplication
- Example of much worse situation will be dealt with later...

Model 2: Parametric Solution

Conditional Compilation:

- In C and C++ you may alternatively use #ifdef's
- The analysis is basically the same as for parameterization, except that there is no performance penalty
  - But choice of which model to be used cannot be made at run-time.

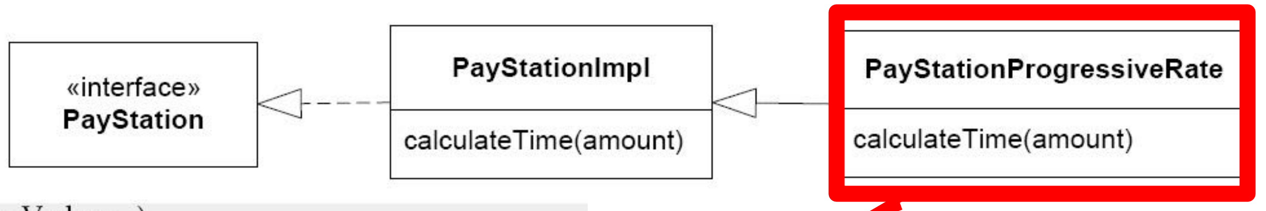
For embedded software where the memory footprint of code is important this may be the best solution!

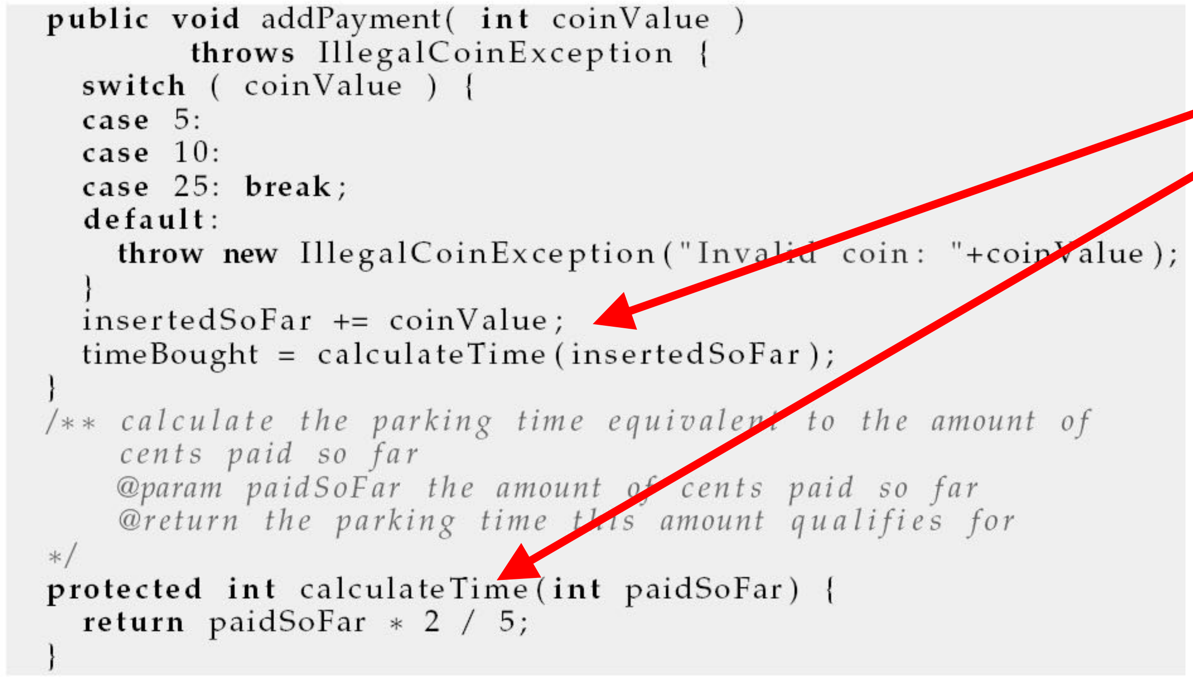


```
/* Main.cpp */
#include "BasicFunctionality.hpp"
#ifdef HAS_ADVANCED_FEATURE
#include "AdvancedFunctionality.hpp"
#endif
 Void main (void)
{
   BasicFunctionality::doSomething();
#ifdef HAS_ADVANCED_FEATURE
  AdvancedFunctionality::doSomething();
#endif
  If (BasicFunctionality::getCondition() ||
#ifdef HAS_ADVANCED_FEATURE
    AdvancedFunctionality::getCondition()
#endif
    false)
  {
    Printf("Condition present");
  }
}
```

Model 3: Polymorphic Solution

Subclass and override!



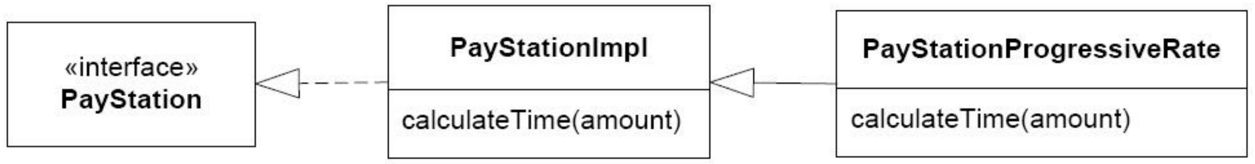


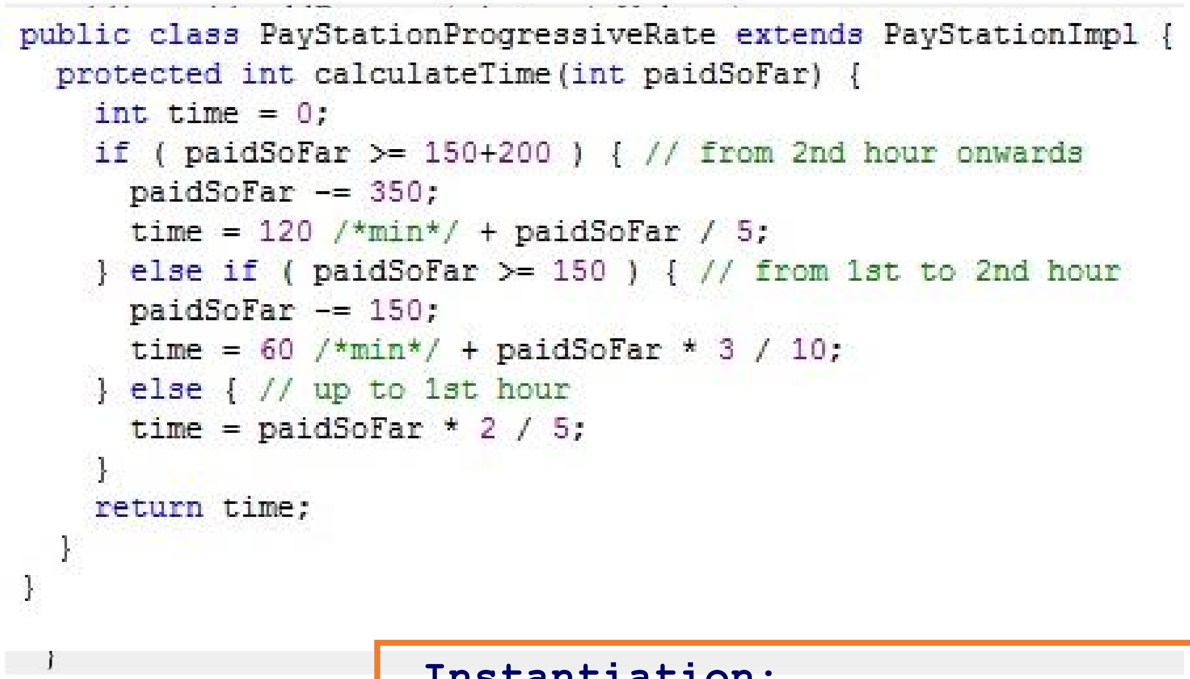
```
public void addPayment( int coinValue )
        throws IllegalCoinException {
  switch ( coinValue ) {
  case 5:
  case 10:
  case 25: break;
  default:
    throw new IllegalCoinException("Invalid coin: "+coinValue);
  }
  insertedSoFar += coinValue;
  timeBought = calculateTime(insertedSoFar);
}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

Refactor to introduce calculateTime method

Model 3: Polymorphic Solution

Subclass and override!





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

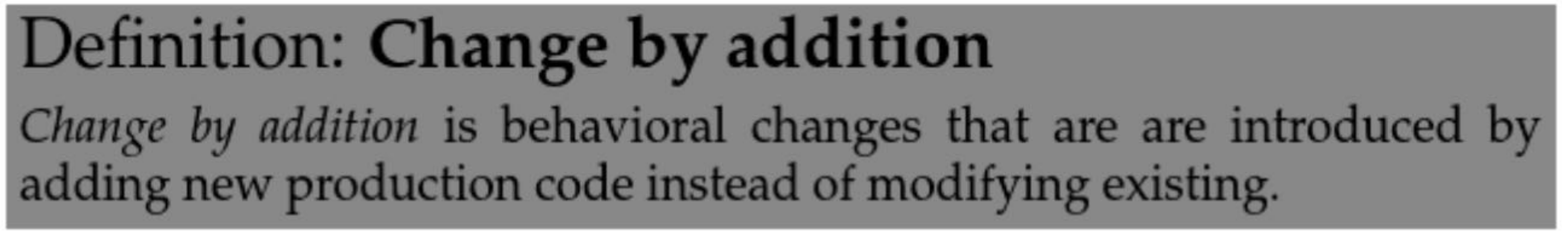
```
Instantiation:
PayStation ps =
   new PayStationProgressiveRate();
```

Model 3: Polymorphic Solution

☺

Reliability

- The first time adding a new rate policy, we *change by modification*
- **But**
  - All following new requirements (regarding rate policies) can be handled by **adding** new subclasses, without **modifying** existing classes
  - **Change by addition, not by modification**



Definition: **Change by addition**

Change by addition is behavioral changes that are are introduced by adding new production code instead of modifying existing.

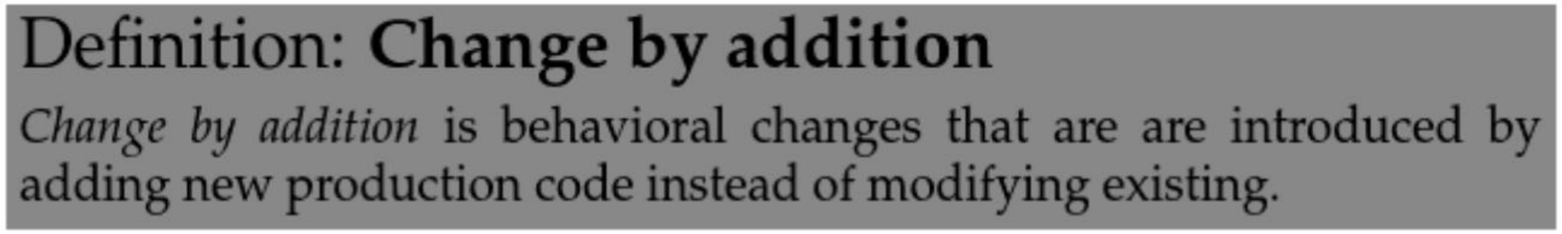
"open for extension, closed for modification"

Model 3: Polymorphic Solution

☺

Reliability

- The first time adding a new rate policy, we **change by modification**
- **But** any new rate policies can be handled by **adding** new subclasses **without modifying** existing classes
    - **Change by addition, not by modification**



Definition: **Change by addition**

Change by addition is behavioral changes that are are introduced by adding new production code instead of modifying existing.

"open for extension, closed for modification"

Analyzability

- No code bloat from conditional statements
- New classes instead!

Model 3: Polymorphic Solution

☹

Increased number of classes

- New subclass for each new rate policy
- i.e., instead of 43 if statements in one class, 43 subclasses

Inheritance is spent on a single type of variation

- With single inheritance, we have “wasted” it on rate policy!
  - What is next? “PayStationProgressiveRateButLinearOnWeekendsWithOracleDataBaseAccessDebuggingVersionAndBothCoinAndMobilePayPaymentOptions” ???
  - Not great for multiple types of variants

- We will discuss this problem in detail later...

Model 3: Polymorphic Solution

☹

Inheritance is compile time binding

- You cannot change the rate model except by rewriting code!
- Impossible to dynamically change rate policy

Reuse across variants is difficult

- GammaTown Preview: “We want a rate policy similar to Alphatown during weekdays but similar to Betatown during weekends.”
  - With some code in one superclass and some in another subclass, combining them will lead to a pretty odd design
  - Refactor into an abstract superclass that contains the rate policies?
    - But do they make sense there?

Model 3: Polymorphic Solution

Pros:

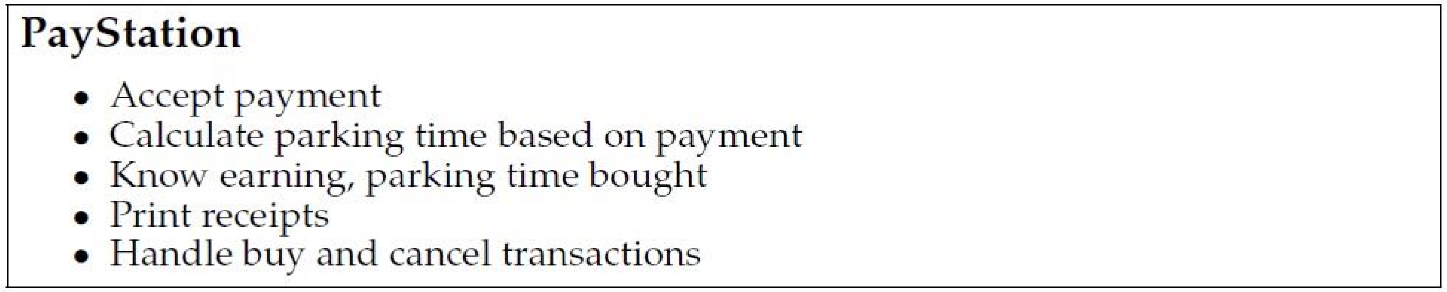
- Avoid multiple maintenance
- Avoid reliability concerns
- Code analyzability

Cons:

- Increased number of classes
- Inheritance spent on one type of variation
- Compile-time binding
- Reuse across variants is difficult

Model 4: Composition

Rule of thumb: No abstraction should have too many responsibilities (e.g., 3 max)


PayStation

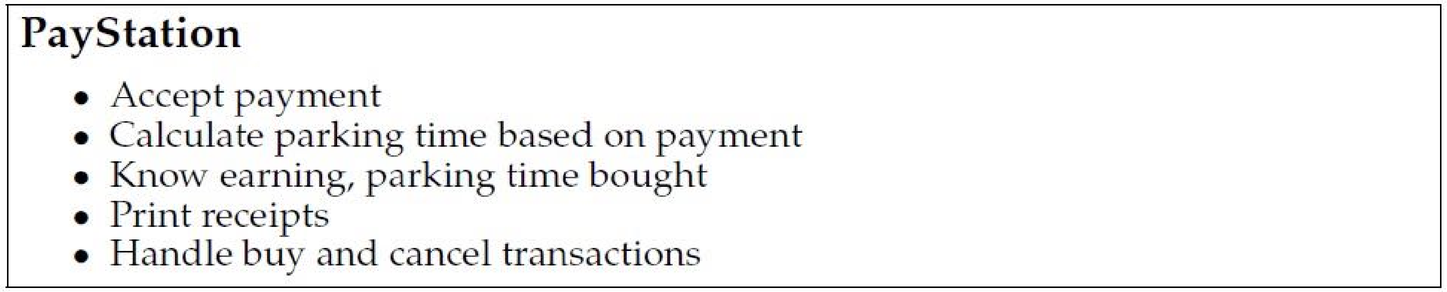
- Accept payment
- Calculate parking time based on payment
- Know earning, parking time bought
- Print receipts
- Handle buy and cancel transactions

The reason that we have to **modify** code to handle the new requirement is because the change involves a responsibility (calculate parking time) that is **buried** within an abstraction (PayStation) **and mixed up with other responsibilities** (print receipt, handle buy, etc.) !!!

So: What do we do???

Model 4: Composition

Rule of thumb: No abstraction should have too many responsibilities (e.g., 3 max)



PayStation

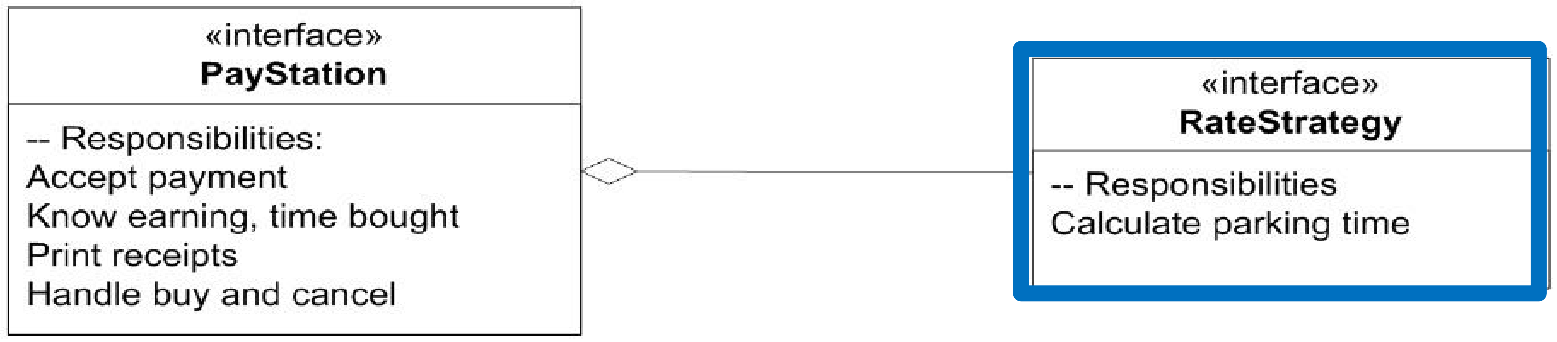
- Accept payment
- Calculate parking time based on payment
- Know earning, parking time bought
- Print receipts
- Handle buy and cancel transactions

The reason that we have to **modify** code to handle the new requirement is because the change involves a responsibility (calculate parking time) that is **buried** within an abstraction (PayStation) **and mixed up with other responsibilities** (print receipt, handle buy, etc.) !!!

So: What do we do???
→ Divide and conquer! (delegate)

Model 4: Composition

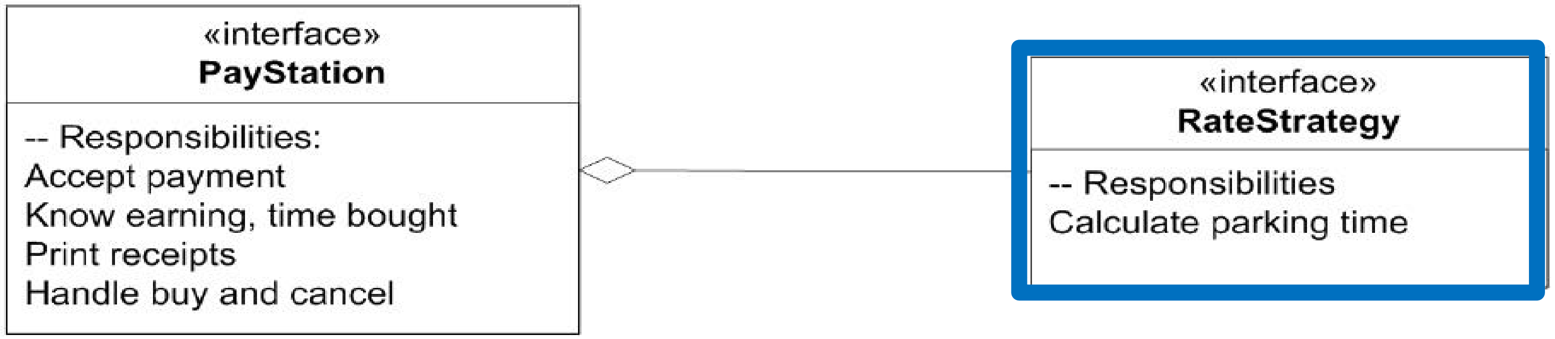
Method: Divide responsibilities!
→ Objects collaborate to provide the required behavior in combination



Put the responsibility in its own abstraction / object

Model 4: Composition

Method: Divide responsibilities!
→ Objects collaborate to provide the required behavior in combination



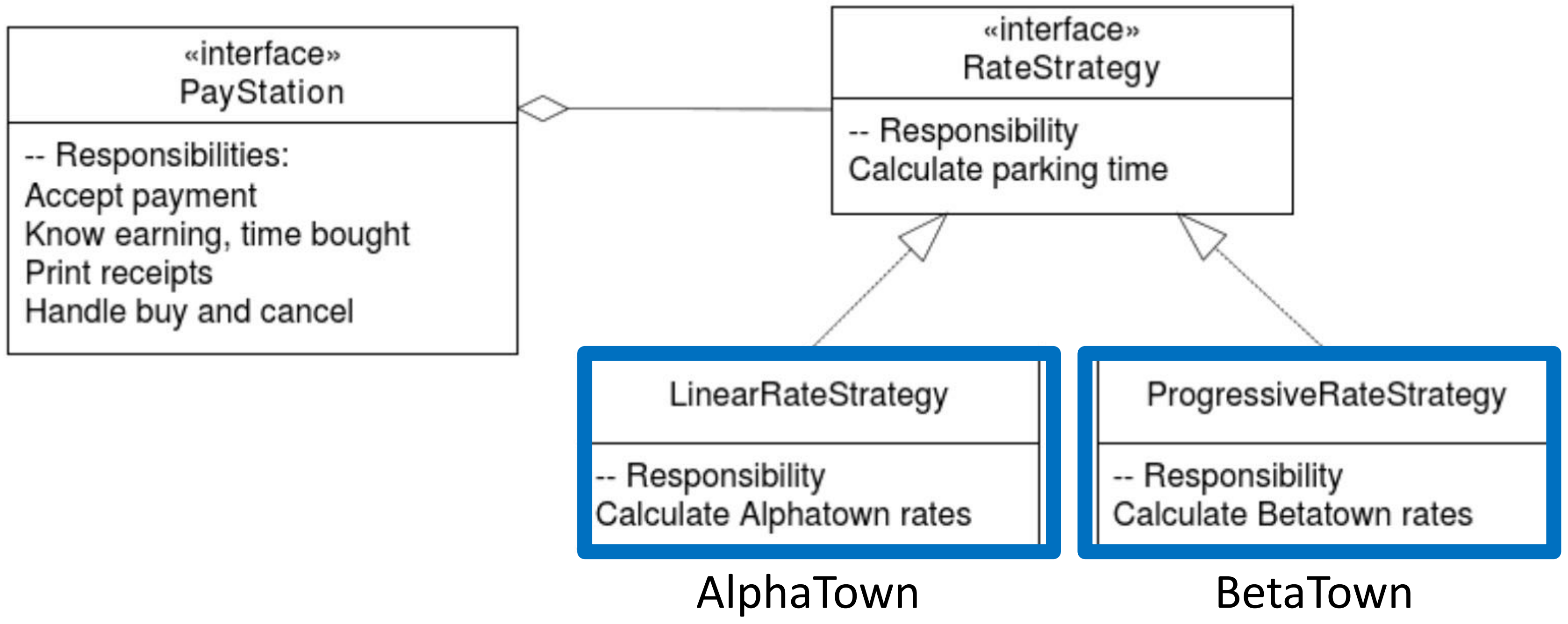
```
 [...]
insertedSoFar += coinValue;
timeBought = someOtherObject.calculateTime(insertedSoFar);
```

Put the responsibility in its own abstraction / object

Model 4: Composition

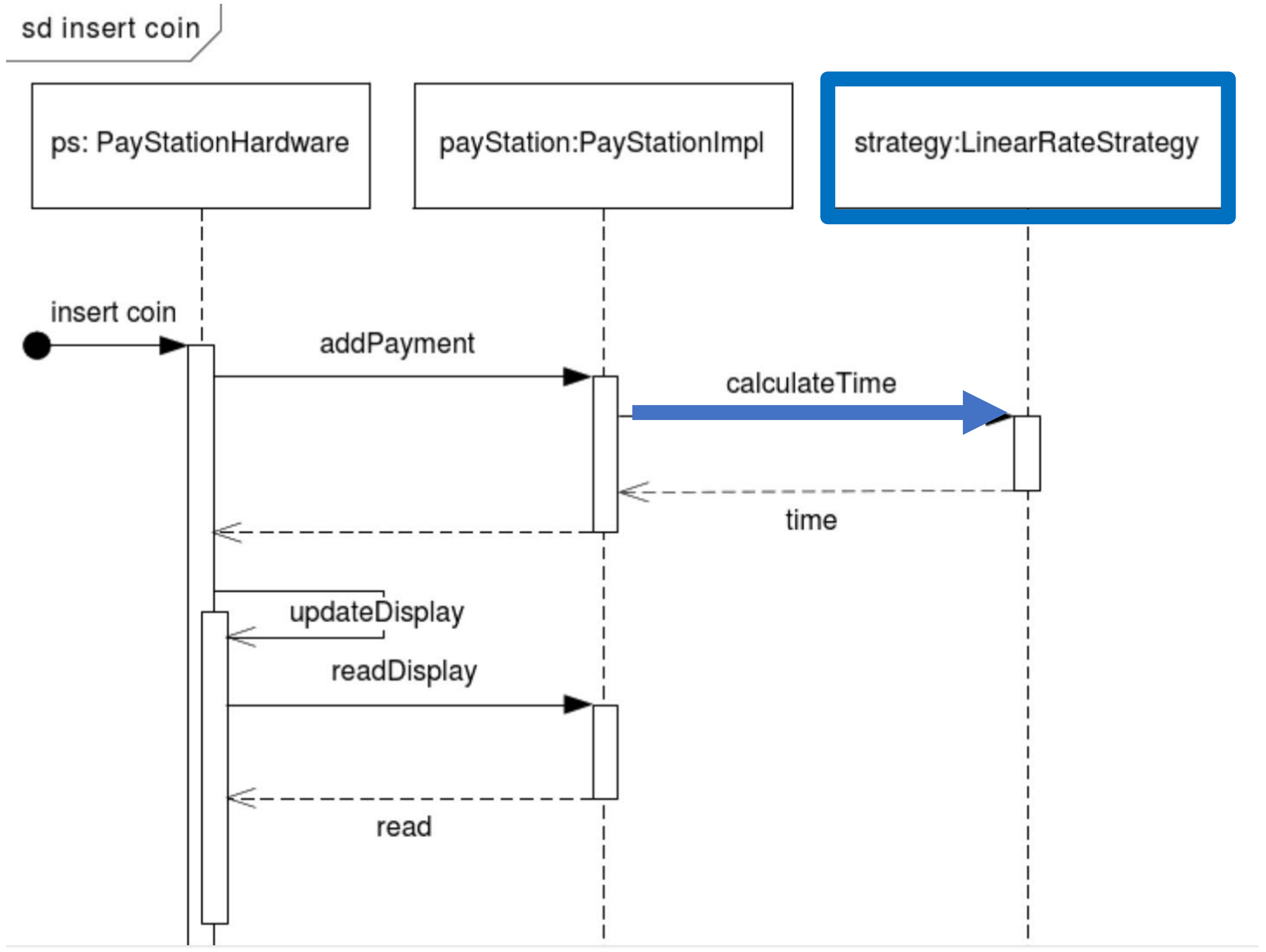
Definition: Delegation

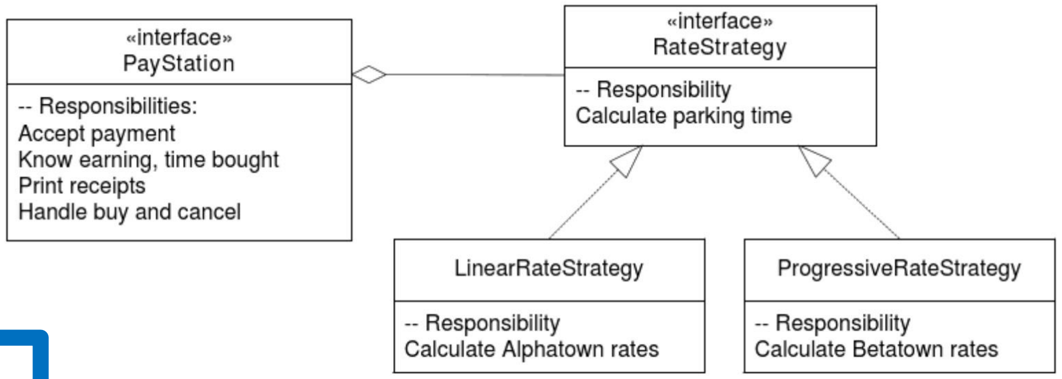
In delegation, two objects collaborate to satisfy a request or fulfill a responsibility. The behavior of the receiving object is partially handled by a subordinate object, called the **delegate**.



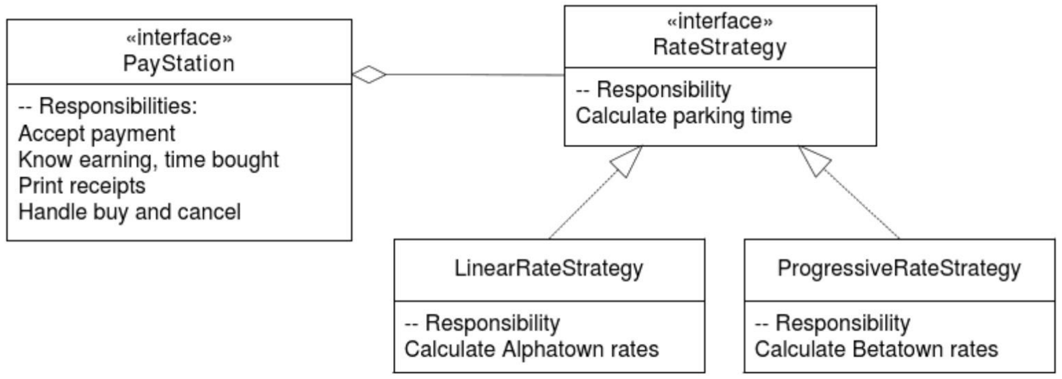
Model 4: Composition

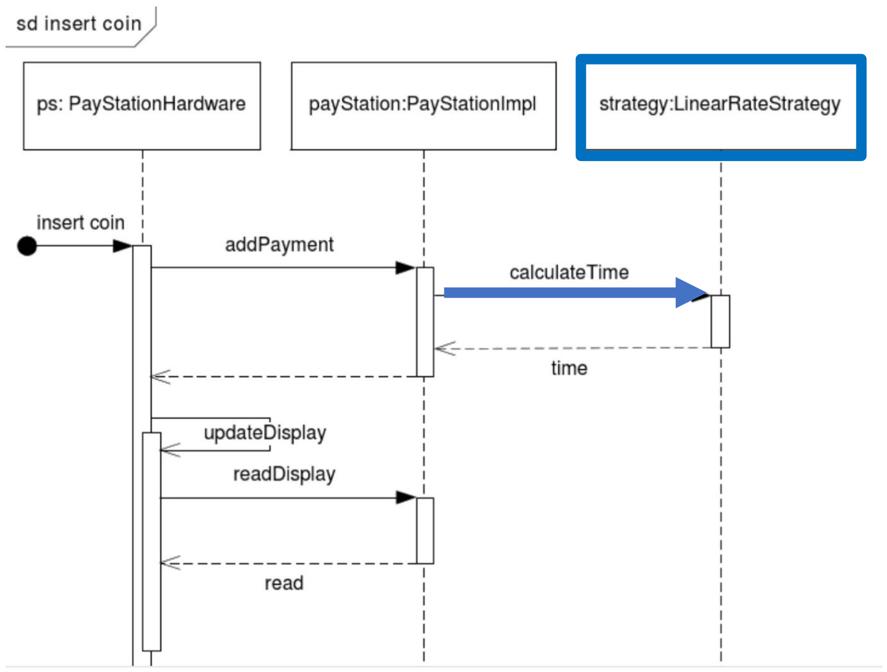
Sequence diagram





Model 4: Composition





In PayStationImpl:

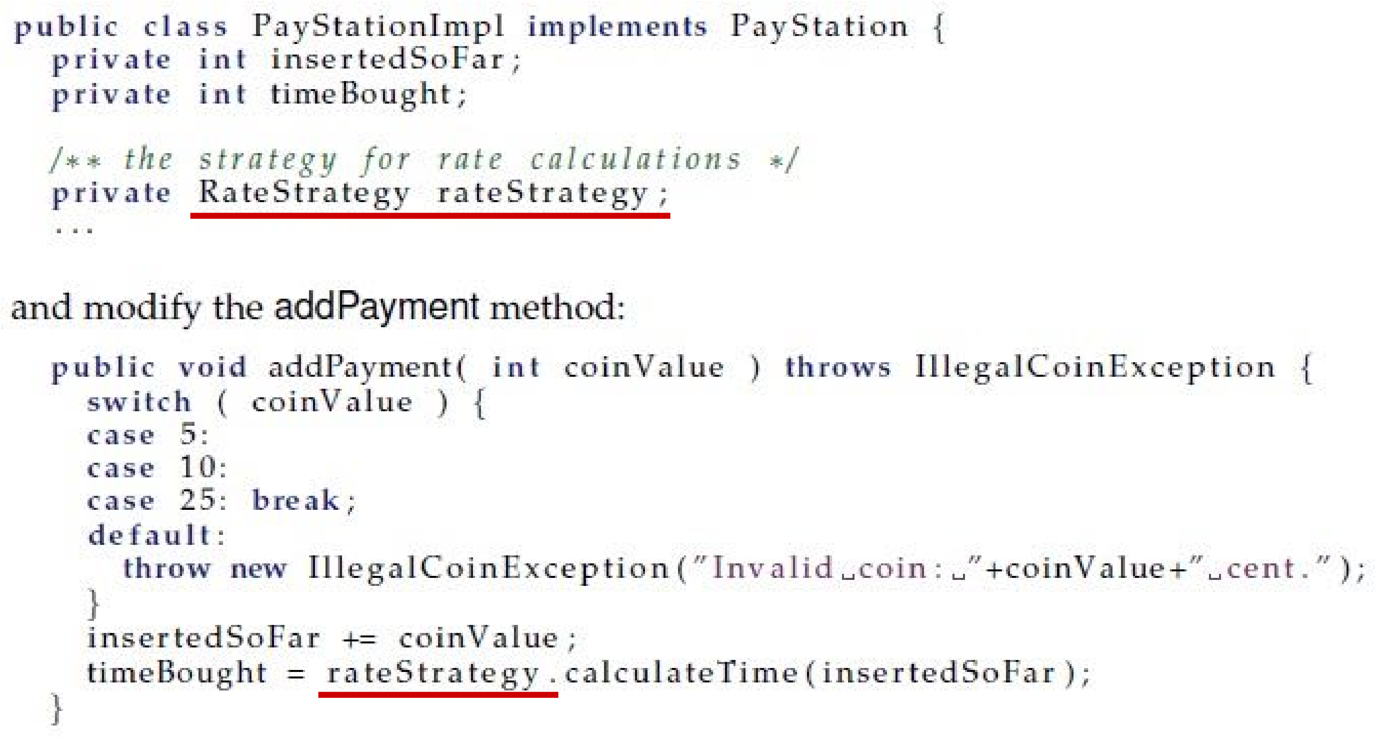
```
public class PayStationImpl implements PayStation {
  private int insertedSoFar;
  private int timeBought;

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

and modify the addPayment method:

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

Model 4: Composition

How does the pay station know which rate strategy object to use?



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

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

and modify the addPayment method:

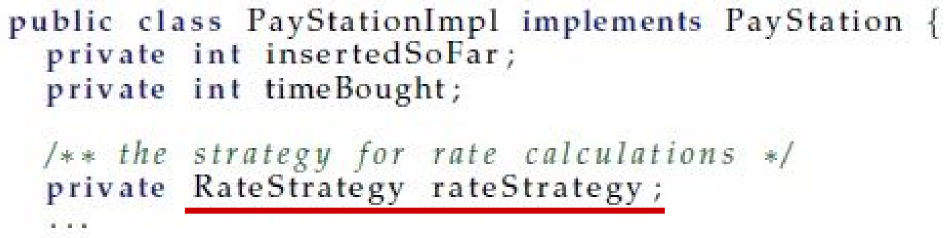
```
  public void addPayment( int coinValue ) throws IllegalCoinException {
    switch ( coinValue ) {
    case 5:
    case 10:
    case 25: break;
    default:
      throw new IllegalCoinException("Invalid coin: "+coinValue+" cent.");
    }
    insertedSoFar += coinValue;
    timeBought = rateStrategy.calculateTime(insertedSoFar);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

Model 4: Composition

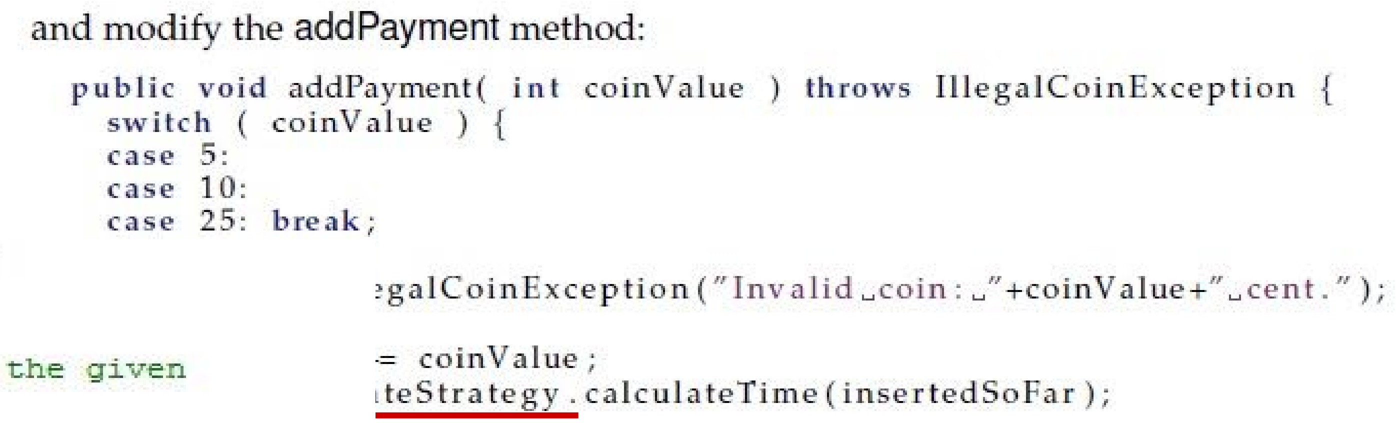
How does the pay station know which rate strategy object to use?

Several possibilities

- Constructor
- Set-method
- Creational patterns (later)

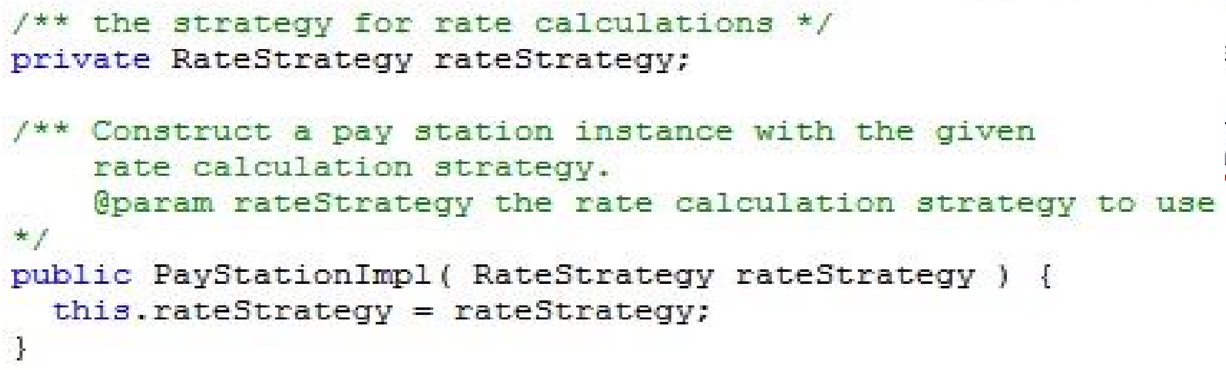


```
public class PayStationImpl implements PayStation {
  private int insertedSoFar;
  private int timeBought;

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



and modify the addPayment method:

```
  public void addPayment( int coinValue ) throws IllegalCoinException {
    switch ( coinValue ) {
    case 5:
    case 10:
    case 25: break;

egalCoinException("Invalid␣coin:␣"+coinValue+"␣cent.");

= coinValue;
teStrategy.calculateTime(insertedSoFar);
```



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

/** Construct a pay station instance with the given
    rate calculation strategy.
    @param rateStrategy the rate calculation strategy to use
*/
public PayStationImpl( RateStrategy rateStrategy ) {
  this.rateStrategy = rateStrategy;
}
```

Model 4: Composition

What are the benefits and liabilities of

Using the constructor to define the strategy?

Using a set-method to define the strategy?

Model 4: Composition

What are the benefits and liabilities of

Using the constructor to define the strategy?

- Compiler will tell you that you have forgotten to make it!
- Early binding - cannot be changed at run-time

Using a set-method to define the strategy?

- What if you forget to set it?
- ... but you can change your mind at run-time !

Model 4: Composition

☺

Responsibilities are clearly stated in interfaces

- Names make more sense:
  - **PayStation** and **RateStrategy**: The responsibilities
  - LinearRateStrategy ect: Concrete behavior fulfilling responsibilities

- Separating responsibilities = higher **cohesion** of code within each abstraction

- Easier testing, debugging

Model 4: Composition

☺

Variant selection is localized

- There is only one place in the code where we decide rate policy
  - (In the configuration/main code where we instantiate the pay station)
  - Contrast with the parametric solution
- **No variant handling code** in the PayStation!

Combinatorial

- We have not used inheritance, so we can still subclass (or use composition) to provide new behavior in other aspects – without interfering with the rate calculation!

- More on this later…

Model 4: Composition

Increased number of objects

- Similar to the polymorphic solution, we trade complexity within the code with complexity of structure

Client objects must be aware of rate strategies

- The client (the one instantiating the PayStation) must be aware of the particular strategy object to pass to the PayStation (Context)
  - Thus, this code creates a hard binding between the two…
- **Never** instantiate the strategies *inside* the Context object (here the PayStationImpl)
  - (unless you want all the liabilities of the parametric approach!)

Model 4: Composition

Pros:

- Separation of responsibilities
- Variant selection is localized
- Combinatorial
- **Reliability**
  - New classes for new rate policies
- **Analyzability**
  - No conditional statements
- **Run-time binding**
  - Can change rate policy while running

Cons:

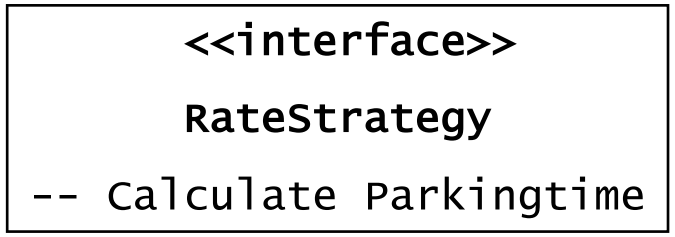
- Increased number of interfaces, objects
- Client objects must be aware of strategies

3-1-2 Process (More when we get to Compositional Design)

③ **Find Variability:** Identify some behavior that is likely to change…

- Rate policy

① **Use an Interface:** State a responsibility that covers this behavior and express it in an interface



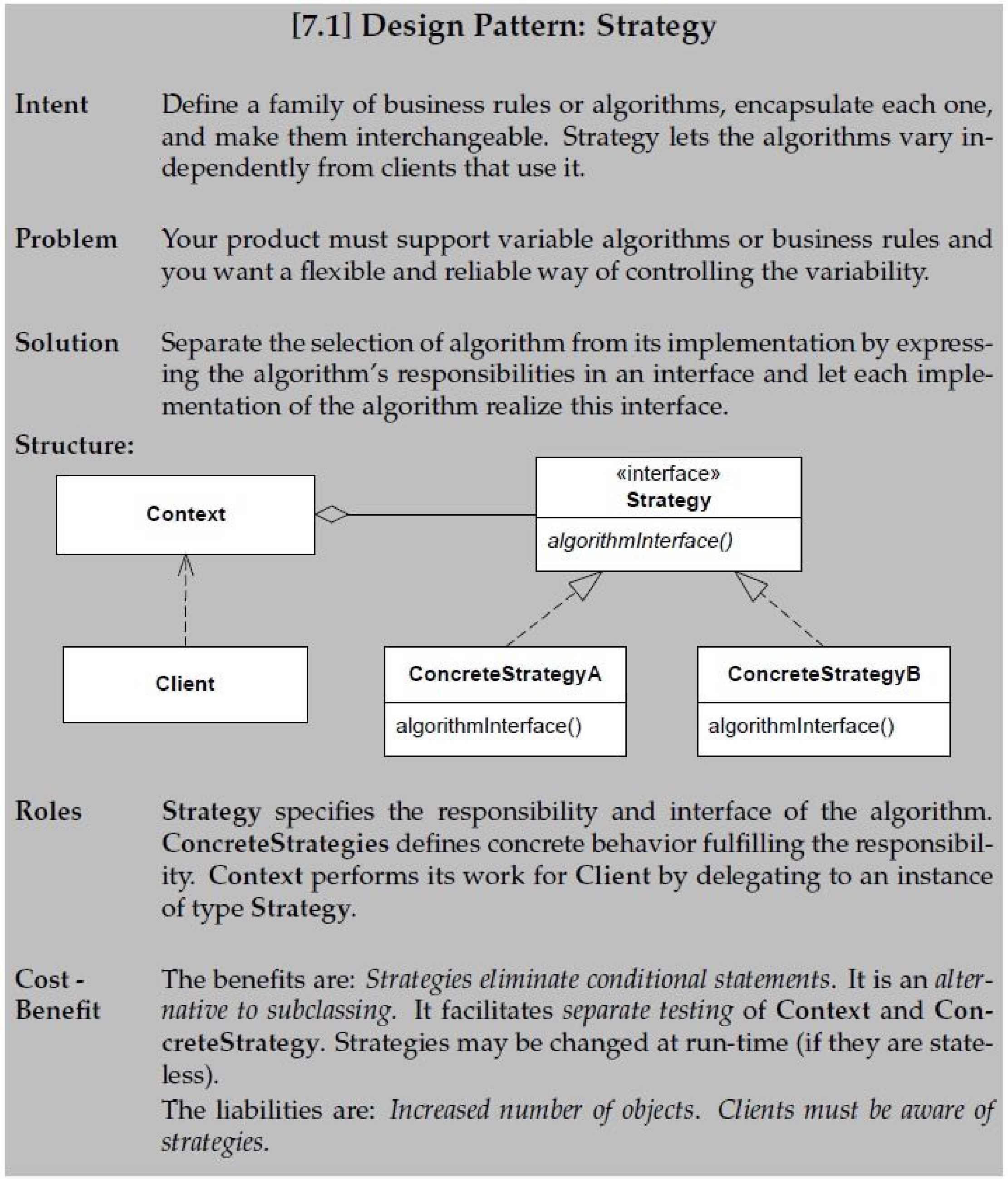
② **Delegate:** The parking machine now performs rate calculations by letting a delegate object do it: the RateStrategy object.

- time = rateStrategy.calculateTime(amount);

Design Patterns

We have derived the **Strategy** design pattern!

- **Strategy** addresses the problem of encapsulating a family of algorithms / business rules and allows implementations to vary independently from the client that uses them



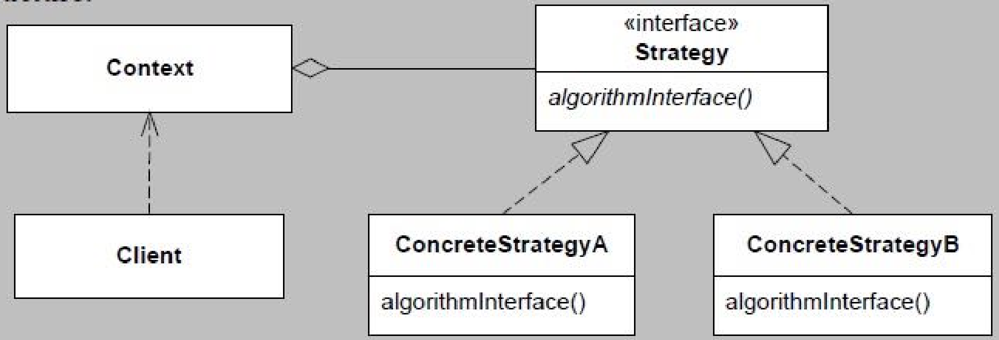
[7.1] Design Pattern: Strategy

Intent Define a family of business rules or algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithms vary independently from clients that use it.

Problem Your product must support variable algorithms or business rules and you want a flexible and reliable way of controlling the variability.

Solution Separate the selection of algorithm from its implementation by expressing the algorithm's responsibilities in an interface and let each implementation of the algorithm realize this interface.

Structure:



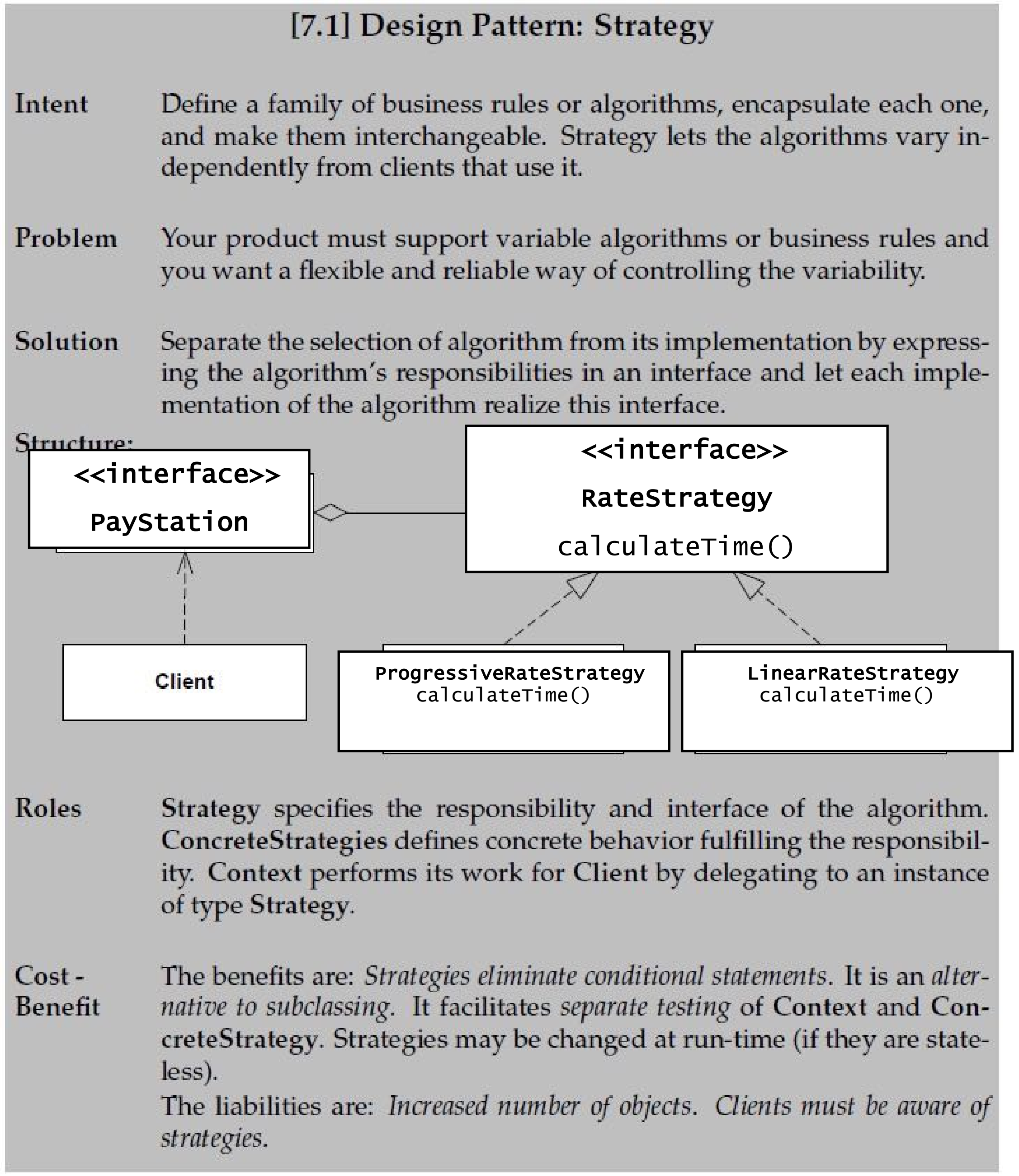
Roles **Strategy** specifies the responsibility and interface of the algorithm. **ConcreteStrategies** defines concrete behavior fulfilling the responsibility. **Context** performs its work for **Client** by delegating to an instance of type **Strategy**.

Cost - Benefit The benefits are: *Strategies eliminate conditional statements*. It is an *alternative to subclassing*. It facilitates *separate testing* of **Context** and **ConcreteStrategy**. Strategies may be changed at run-time (if they are stateless).
The liabilities are: *Increased number of objects*. *Clients must be aware of strategies*.

Design Patterns

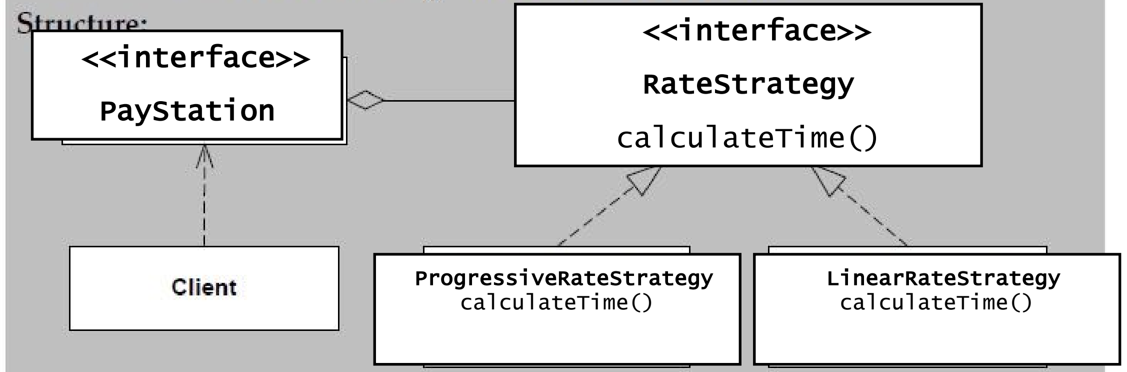
We have derived the **Strategy** design pattern!

- **Strategy** addresses the problem of encapsulating a family of algorithms / business rules and allows implementations to vary independently from the client that uses them



[7.1] Design Pattern: Strategy

Intent	Define a family of business rules or algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithms vary independently from clients that use it.
Problem	Your product must support variable algorithms or business rules and you want a flexible and reliable way of controlling the variability.
Solution	Separate the selection of algorithm from its implementation by expressing the algorithm's responsibilities in an interface and let each implementation of the algorithm realize this interface.
Structure:	
Roles	**Strategy** specifies the responsibility and interface of the algorithm. **ConcreteStrategies** defines concrete behavior fulfilling the responsibility. **Context** performs its work for **Client** by delegating to an instance of type **Strategy**.
Cost - Benefit	The benefits are: *Strategies eliminate conditional statements*. It is an *alternative to subclassing*. It facilitates *separate testing* of **Context** and **ConcreteStrategy**. Strategies may be changed at run-time (if they are stateless). The liabilities are: *Increased number of objects*. *Clients must be aware of strategies*.



Design Patterns

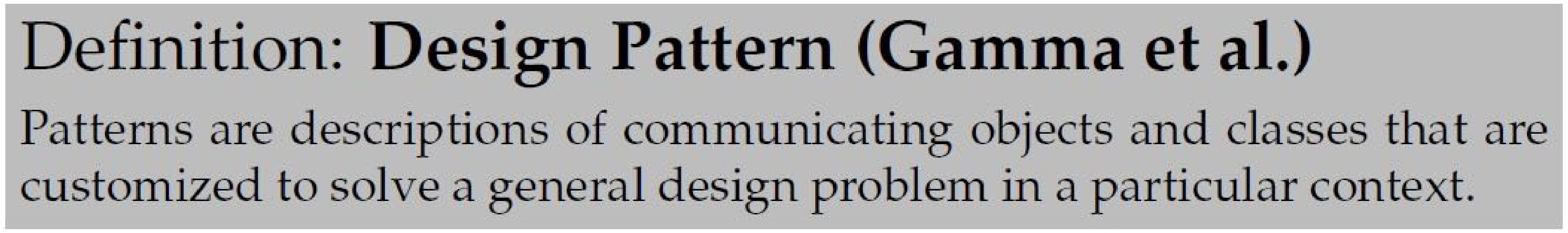
A **design pattern** is a solution to a problem in a context.

“Pattern” concept originally applied to houses, urban planning
→ Patterns at different scales, can be used recursively

The collection of patterns (**pattern catalog**) is a design tool.
Also a **communication** tool – “pattern language”.
→ Facilitate design discussions
→ Represent knowledge gained through design experience

Design Patterns

Design patterns didn't catch on in architecture… but they did in software engineering! (~1980s)


Definition: **Design Pattern (Gamma et al.)**

Patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.

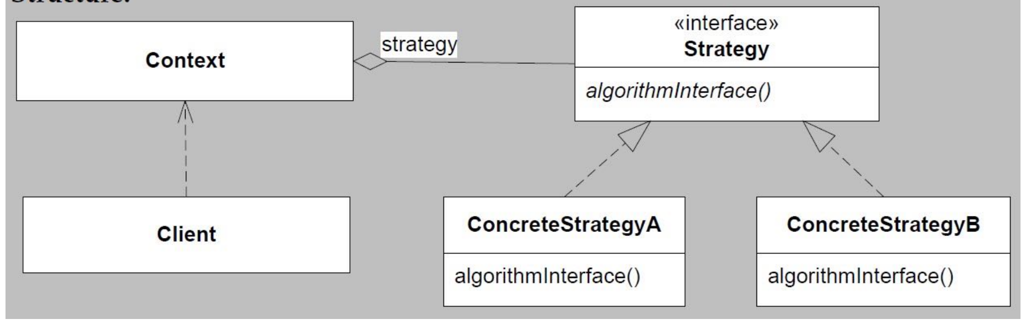
Generally a design must have been used several times successfully (>= 3) before it is considered a "pattern".

Design Patterns

Definition: Design Pattern (Gamma et al.)

Patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.

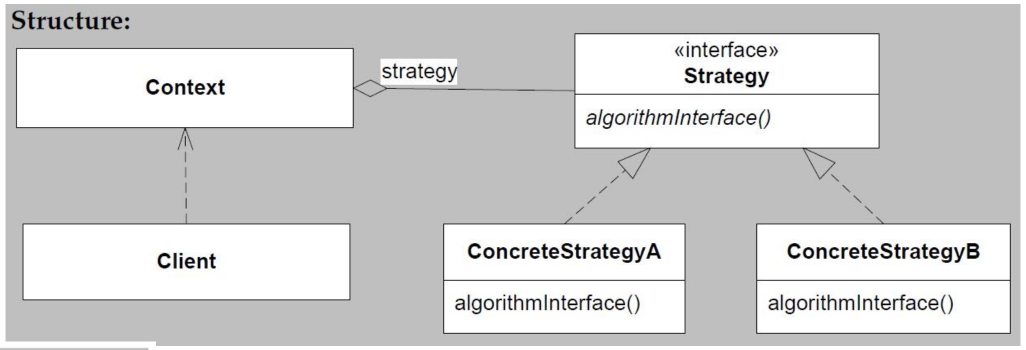
Structure:



Design Patterns

Definition: Design Pattern (Gamma et al.)

Patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.

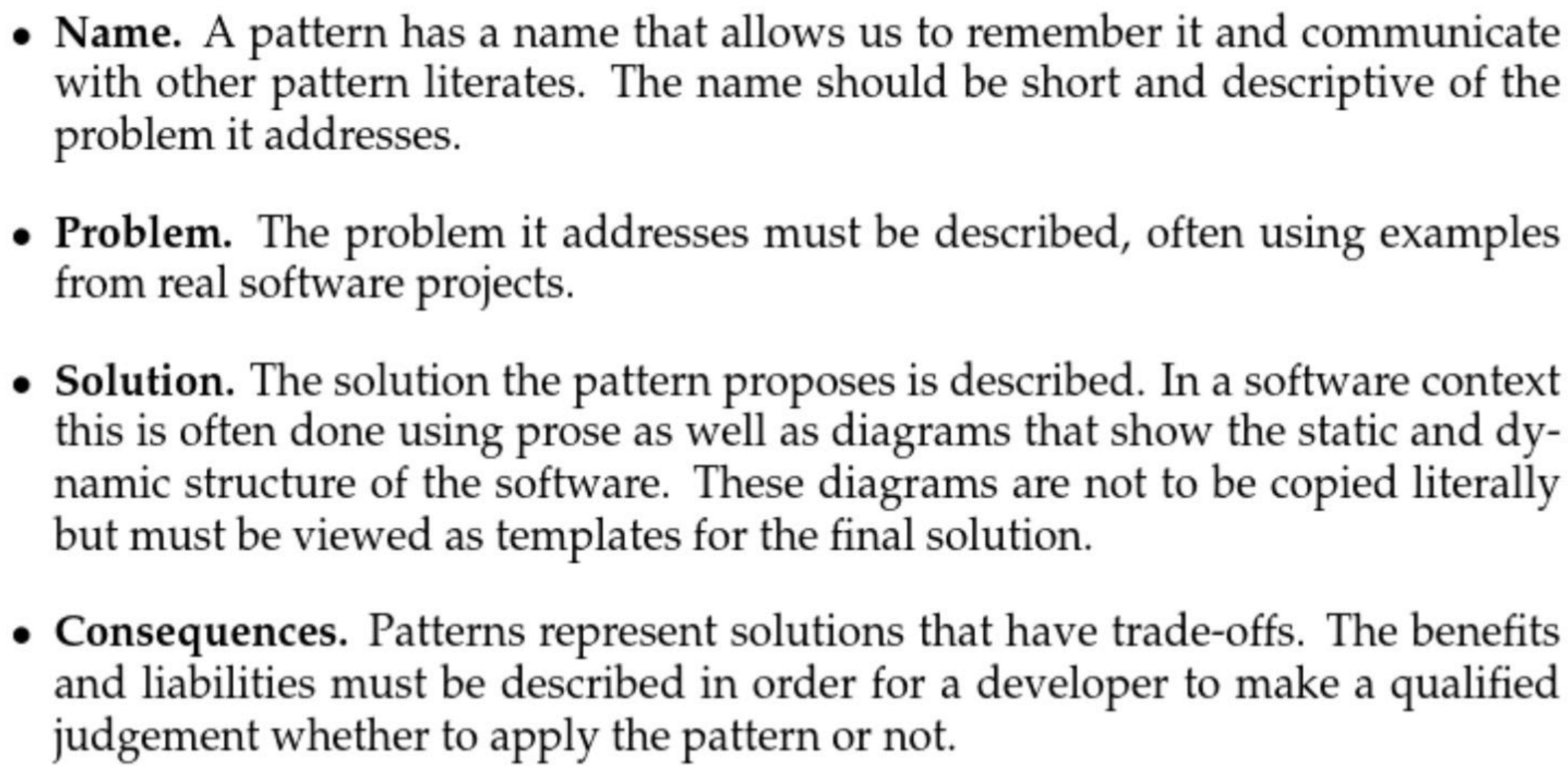


Definition: Design Pattern (Beck et al.)

A design pattern is a particular prose form of recording design information such that designs which have worked well in the past can be applied again in similar situations in the future.

Prose form = “template”, e.g.,
- Name
- Problem
- Solution
- Consequences

Design Patterns



- **Name.** A pattern has a name that allows us to remember it and communicate with other pattern literates. The name should be short and descriptive of the problem it addresses.

- **Problem.** The problem it addresses must be described, often using examples from real software projects.

- **Solution.** The solution the pattern proposes is described. In a software context this is often done using prose as well as diagrams that show the static and dynamic structure of the software. These diagrams are not to be copied literally but must be viewed as templates for the final solution.

- **Consequences.** Patterns represent solutions that have trade-offs. The benefits and liabilities must be described in order for a developer to make a qualified judgement whether to apply the pattern or not.

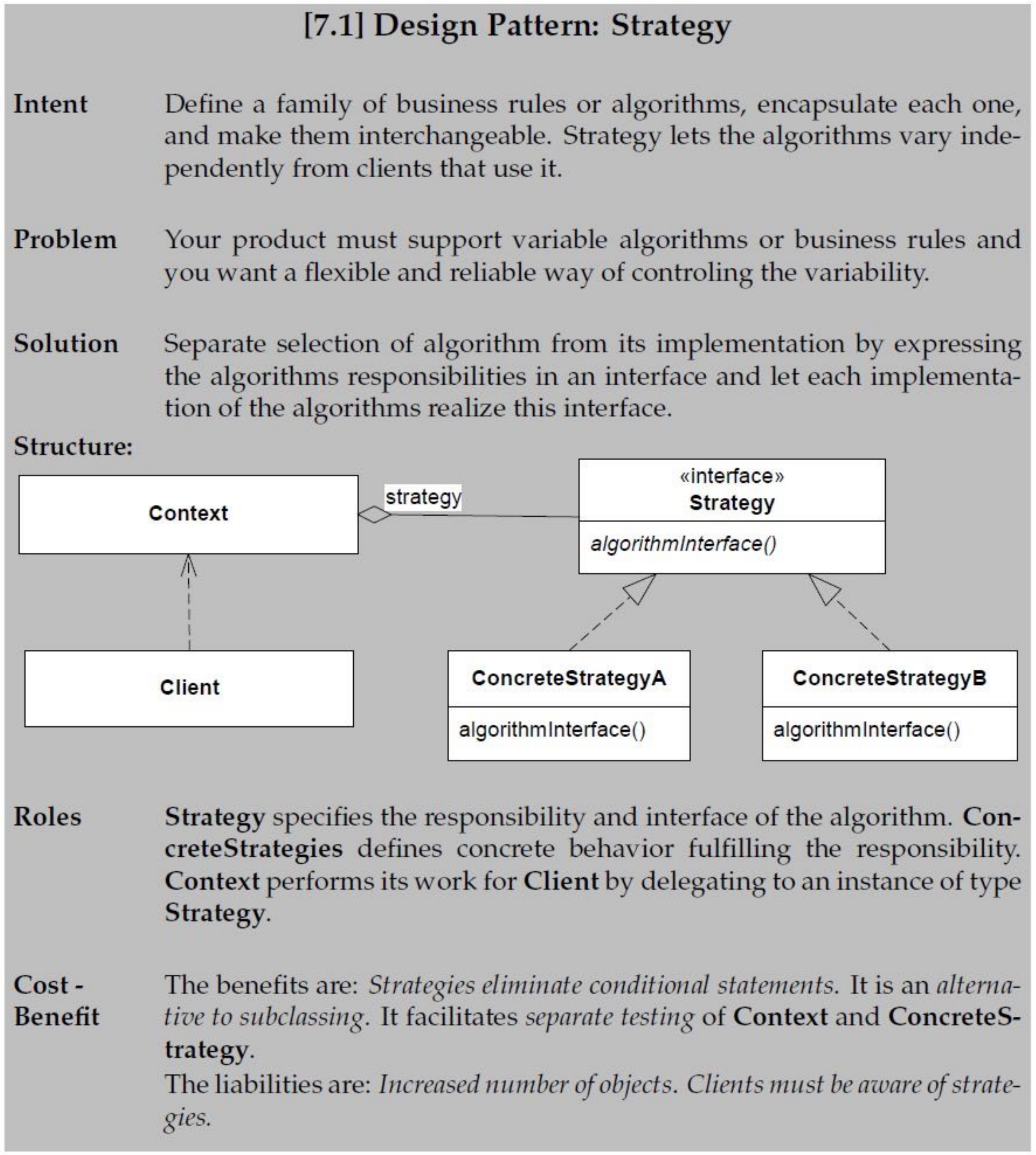
Design Patterns

Intent

- Short description

Roles

- Responsibilities of each participating object/abstraction in the pattern



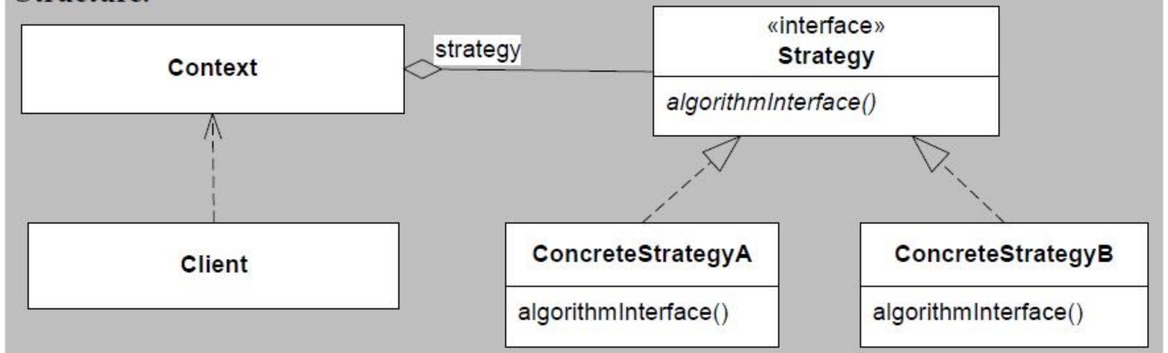
[7.1] Design Pattern: Strategy

Intent Define a family of business rules or algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithms vary independently from clients that use it.

Problem Your product must support variable algorithms or business rules and you want a flexible and reliable way of controling the variability.

Solution Separate selection of algorithm from its implementation by expressing the algorithms responsibilities in an interface and let each implementation of the algorithms realize this interface.

Structure:



Roles **Strategy** specifies the responsibility and interface of the algorithm. **ConcreteStrategies** defines concrete behavior fulfilling the responsibility. **Context** performs its work for **Client** by delegating to an instance of type **Strategy**.

Cost - Benefit The benefits are: *Strategies eliminate conditional statements*. It is an *alternative to subclassing*. It facilitates *separate testing* of **Context** and **ConcreteStrategy**.
The liabilities are: *Increased number of objects. Clients must be aware of strategies.*

Design Patterns

UML: Unified Modeling Language

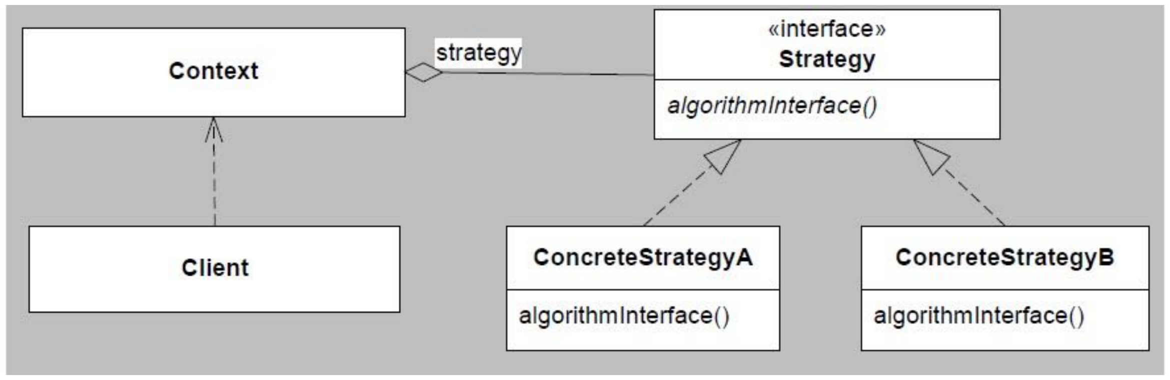
- A graphical language for describing architecture/design

UML Class Diagrams

- Structure

UML Sequence Diagrams

- Processes



Design Patterns

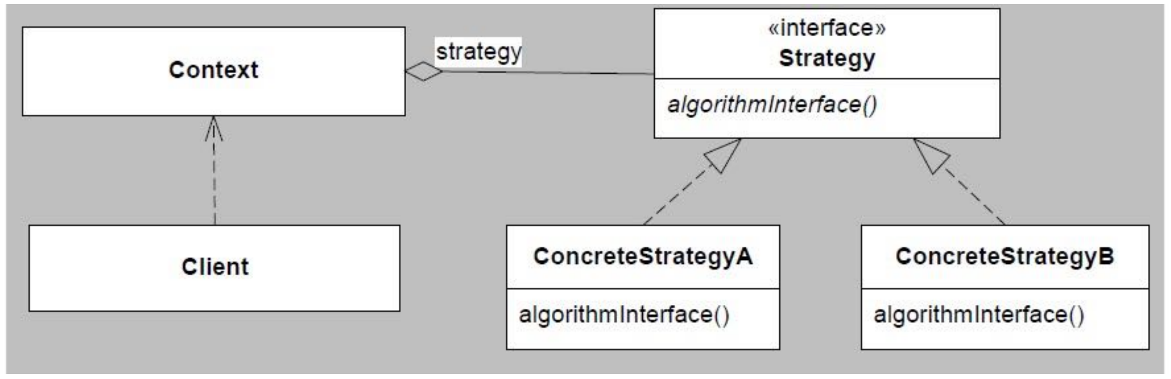
UML: Unified Modeling Language

- A graphical language for describing architecture/design

UML Class Diagram:

Class

Interface



Implementations

Design Patterns

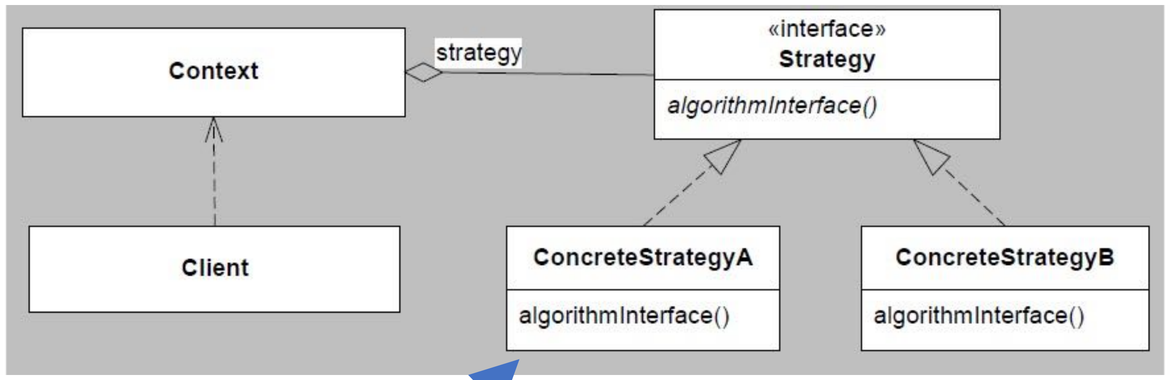
UML: Unified Modeling Language

- A graphical language for describing architecture/design

UML Class Diagram:

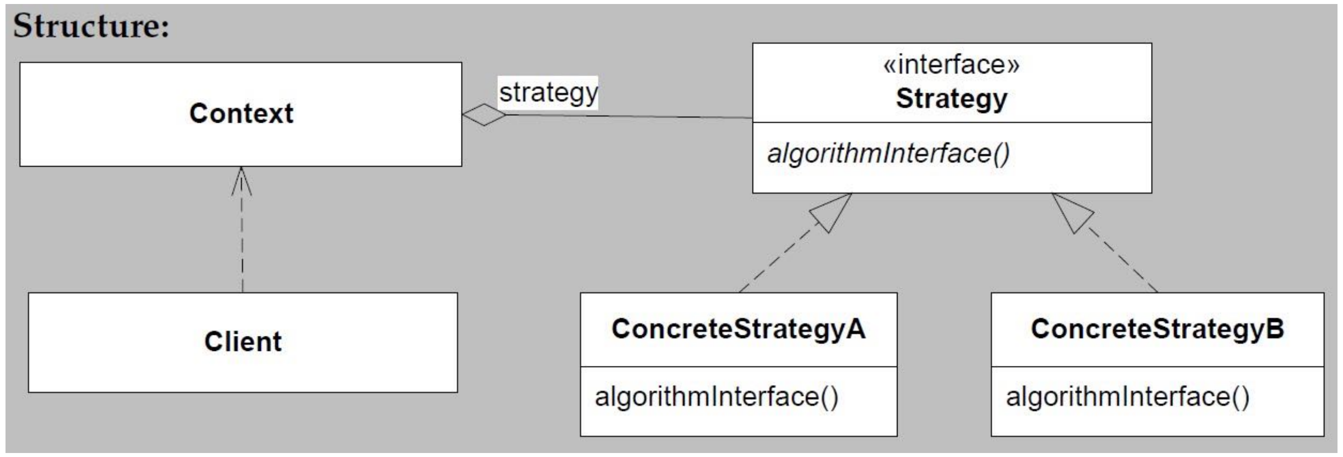
Aggregation

Association



Attributes/methods

Design Patterns



Many patterns are **structurally equal**
→ Their UML class diagrams are more or less identical!

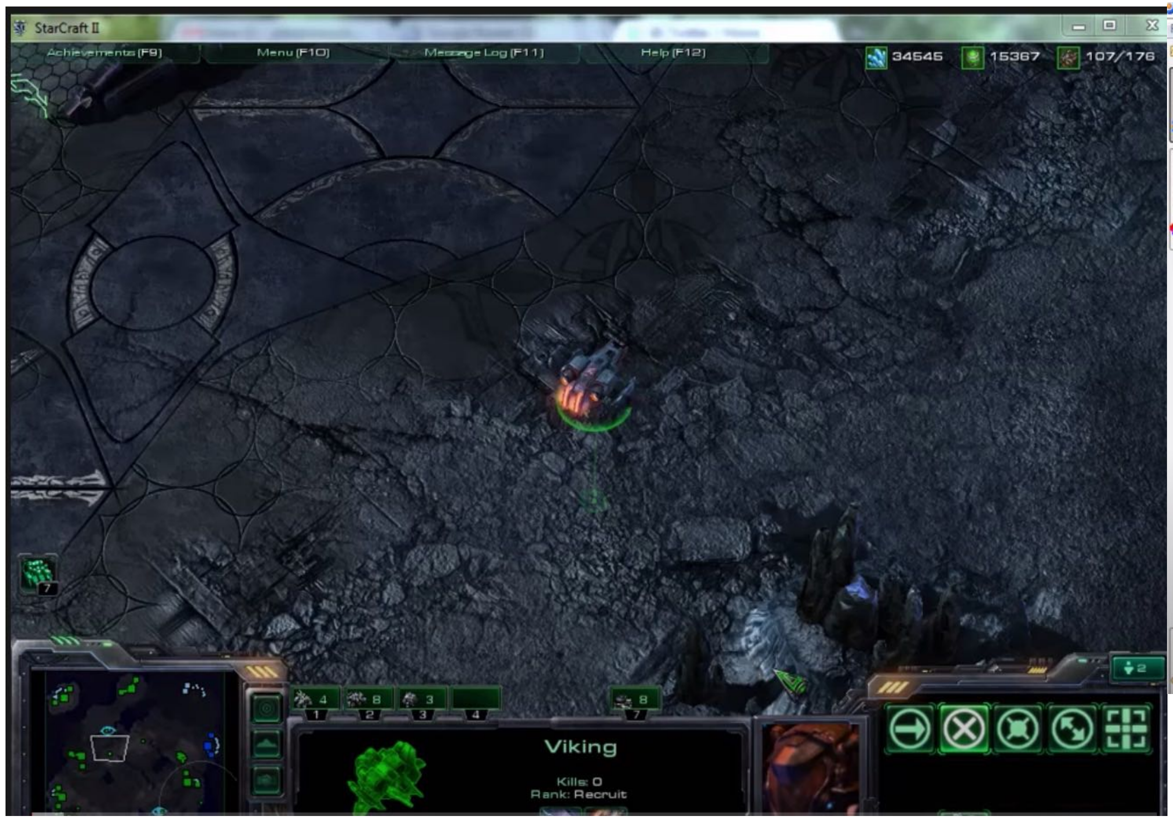
Patterns are defined by the **problem they solve!**

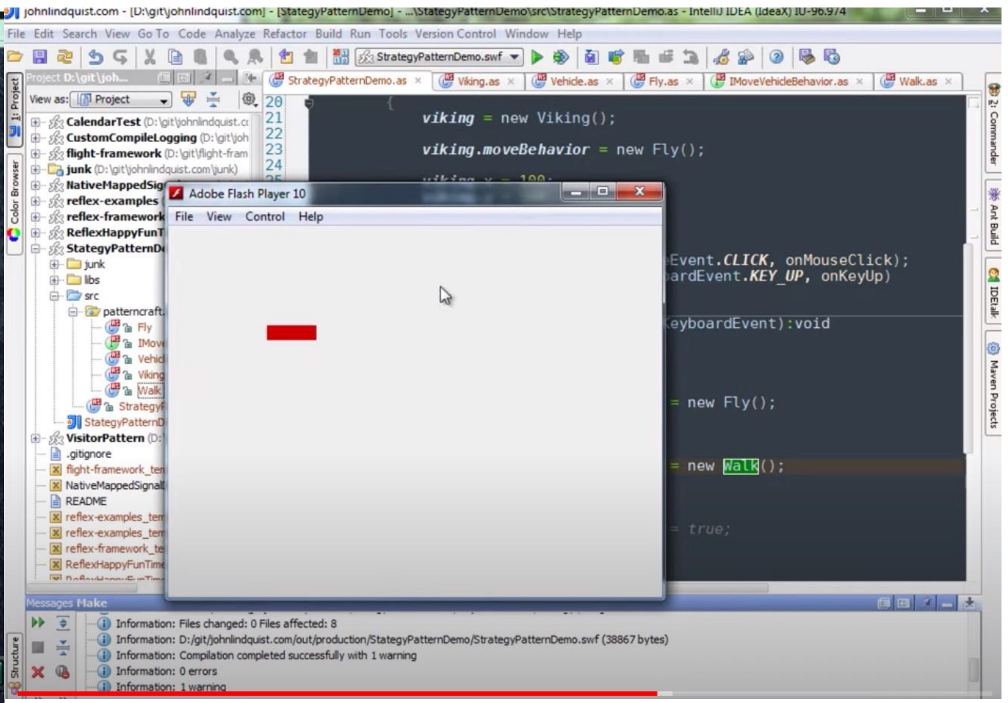
Recall: Strategy addresses the problem of **handling variability of algorithms / business rules, making them interchangeable**

Design Patterns

PatternCraft: https://www.youtube.com/playlist?list=PL6A29DA14366FF6C3

Strategy pattern: https://www.youtube.com/watch?v=MOEsKHqLiBM





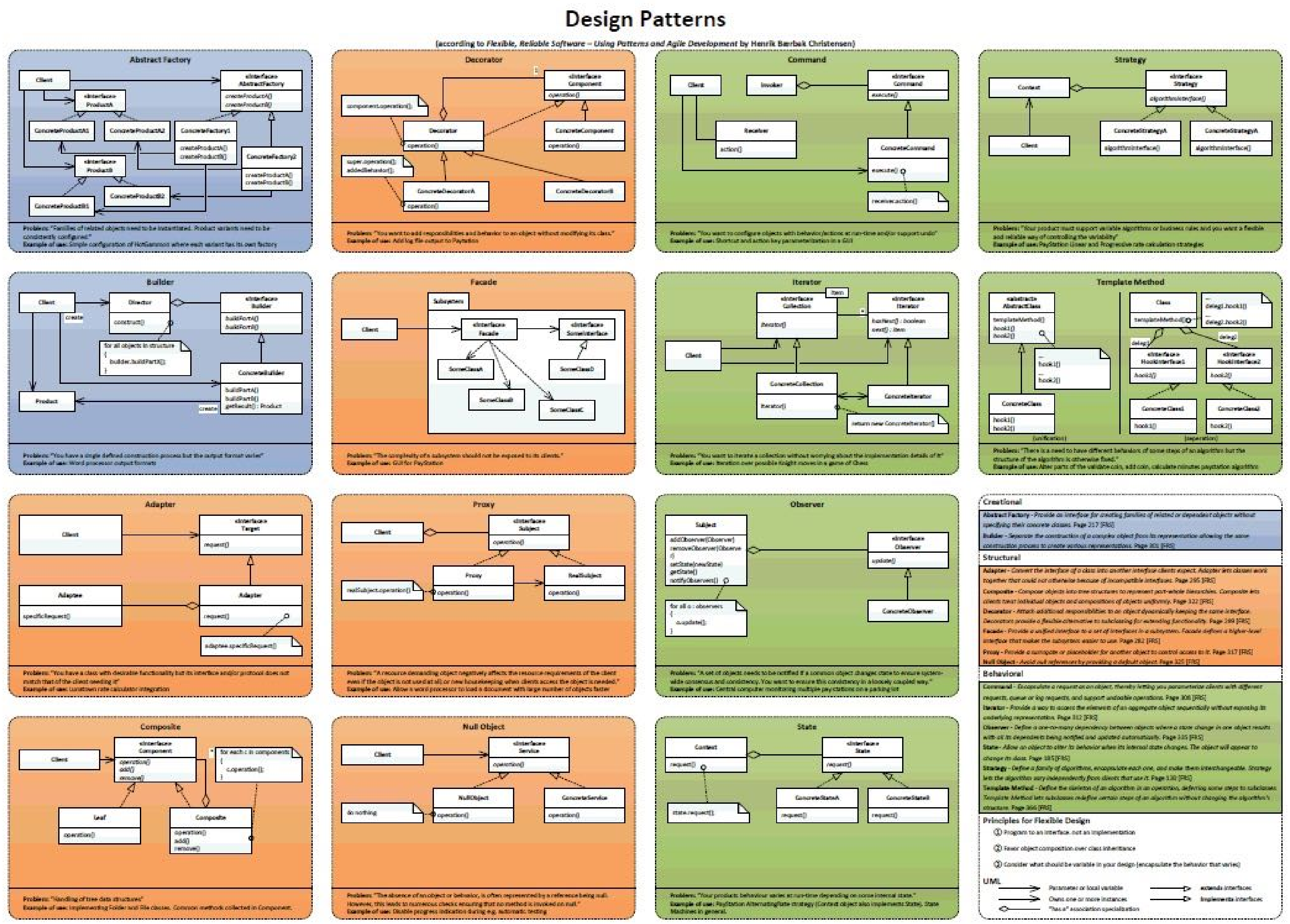
```
viking = new Viking();
viking.moveBehavior = new Fly();
```

```
Event.CLICK, onMouseClick);
ardEvent.KEY_UP, onKeyUp)
eyboardEvent):void
= new Fly();
= new Walk();
= true;
```

```
Information: Files changed: 0 Files affected: 8
Information: D:/git/johnlindquist.com/out/production/StategyPatternDemo/StrategyPatternDemo.swf (38867 bytes)
Information: Compilation completed successfully with 1 warning
Information: 0 errors
Information: 1 warning
```

Design Patterns

More patterns later…



Next time:

Refactoring

Integration Testing