Design Patterns

Shin-Jie Lee (李信杰)
Associate Professor
Department of CSIE
National Cheng Kung University

Selected Design Patterns

- ❑ Strategy
- ❑ Composite & Decorator
- ❑ Factory Method & Abstract Factory
- ❑ Template Method
- ❑ Adapter
- ❑ State
- ❑ Visitor

Strategy Pattern

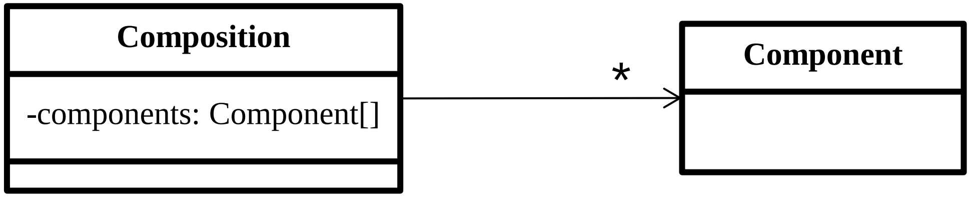
Design Aspect of Strategy

An algorithm

Text Composition Design (Strategy)

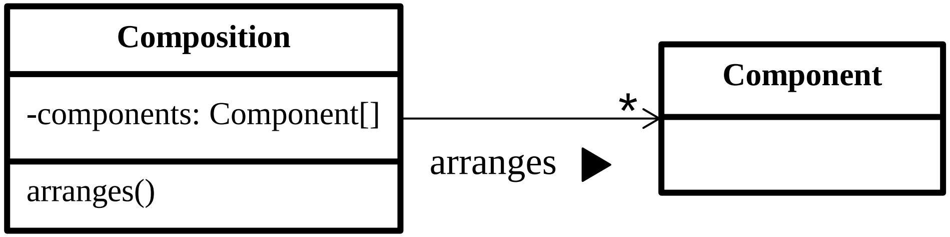
Requirements Statement$_1$

- The Composition class maintains a collection of Component instances, which represent text and graphical elements in a document.



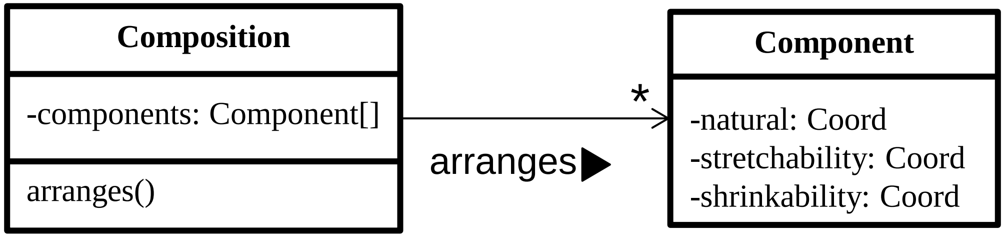
Requirements Statement$_2$

- A composition arranges component objects into lines using a linebreaking strategy.



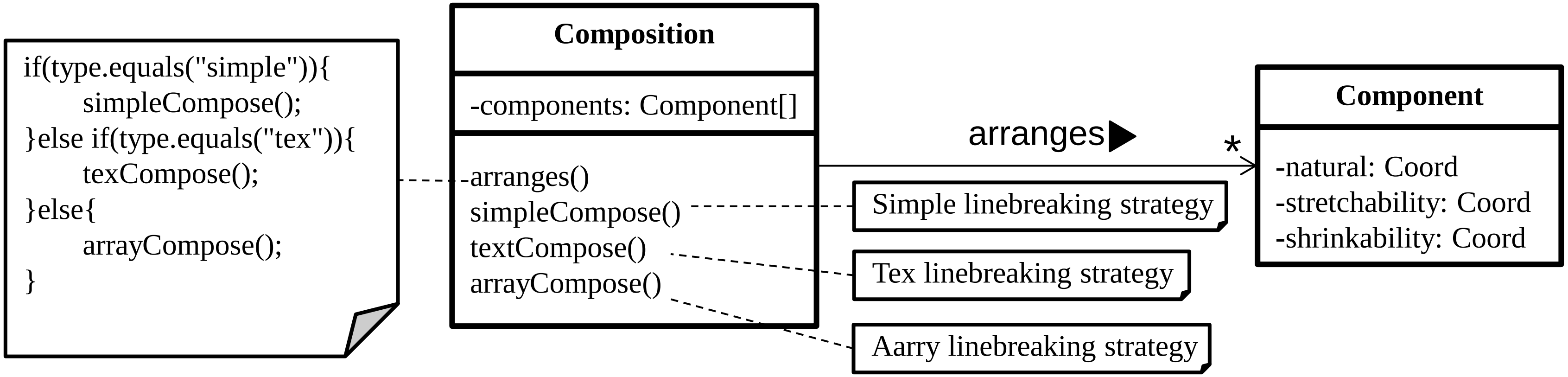
Requirements Statement$_3$

- ❑ Each component has an associated natural size, stretchability, and shrinkability.
- ❑ The stretchability defines how much the component can grow beyond its natural size; shrinkability is how much it can shrink.



Requirements Statement$_4$

- When a new layout is required, the composition calls its compose method to determine where to place linebreaks.
- There are 3 different algorithms for breaking lines:
  - **Simple Composition:** A simple strategy that determines line breaks one at a time.
  - **Tex Composition:** This strategy tries to optimize line breaks globally, that is, one paragraph at a time.
  - **Array Composition:** A strategy that selects breaks so that each row has a fixed number of items. It's useful for breaking a collection of icons into rows, for example.



```
if(type.equals("simple")){
        simpleCompose();
}else if(type.equals("tex")){
        texCompose();
}else{
        arrayCompose();
}
```

Composition

-components: Component[]

-arranges()
simpleCompose()
textCompose()
arrayCompose()

arranges ▶

*

Component

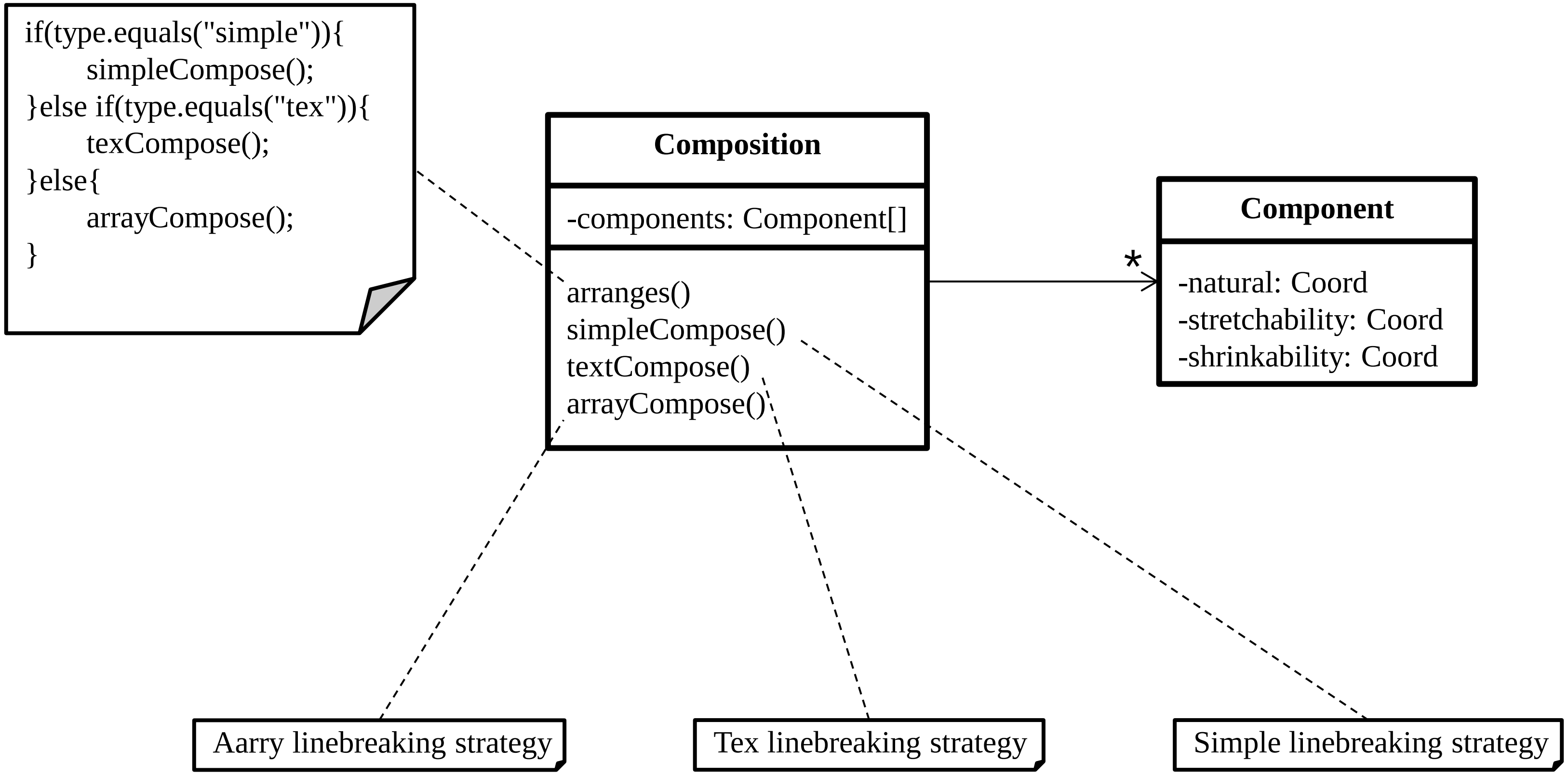
-natural: Coord
-stretchability: Coord
-shrinkability: Coord

Simple linebreaking strategy

Tex linebreaking strategy

Aarry linebreaking strategy

Initial Design



```
if(type.equals("simple")){
        simpleCompose();
}else if(type.equals("tex")){
        texCompose();
}else{
        arrayCompose();
}
```

Composition

-components: Component[]

arranges()
simpleCompose()
textCompose()
arrayCompose()

Component

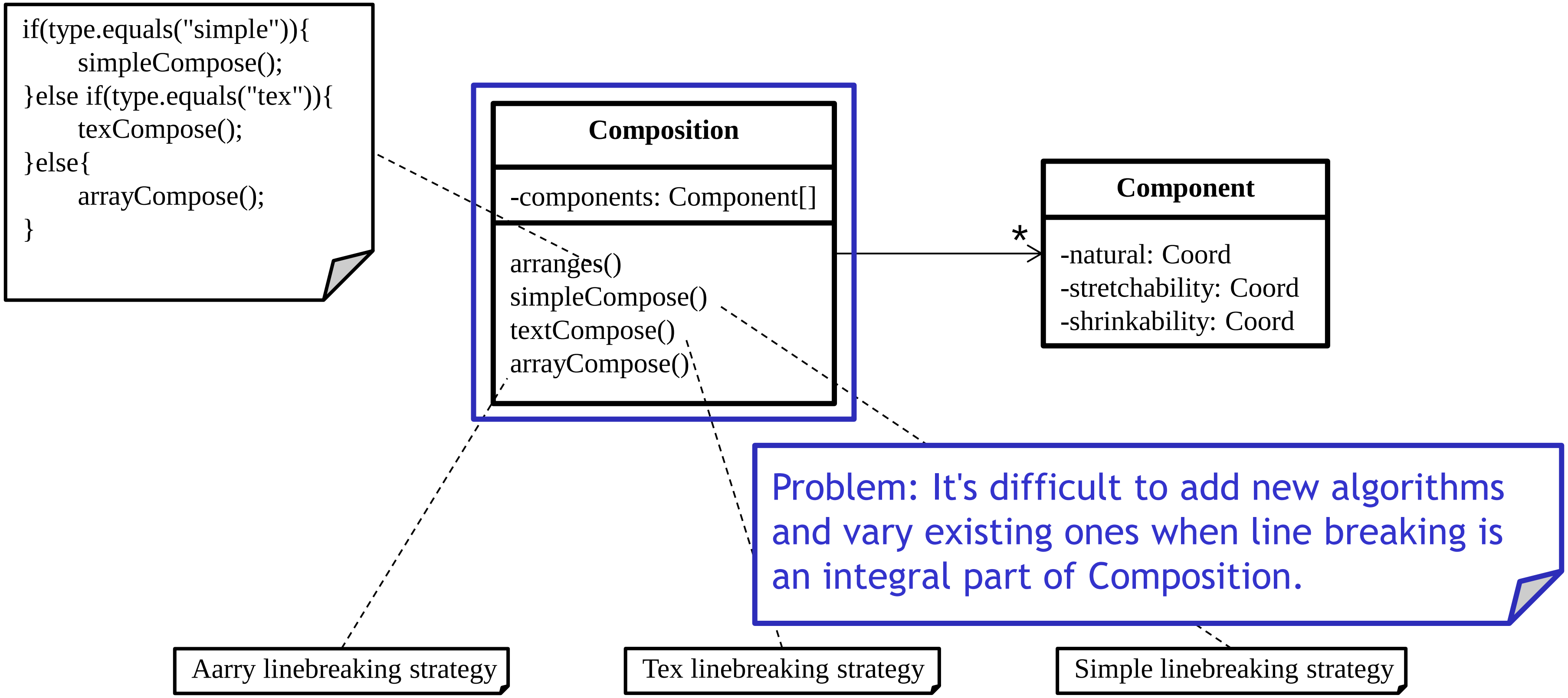
-natural: Coord
-stretchability: Coord
-shrinkability: Coord

Aarry linebreaking strategy

Tex linebreaking strategy

Simple linebreaking strategy

Problems with Initial Design



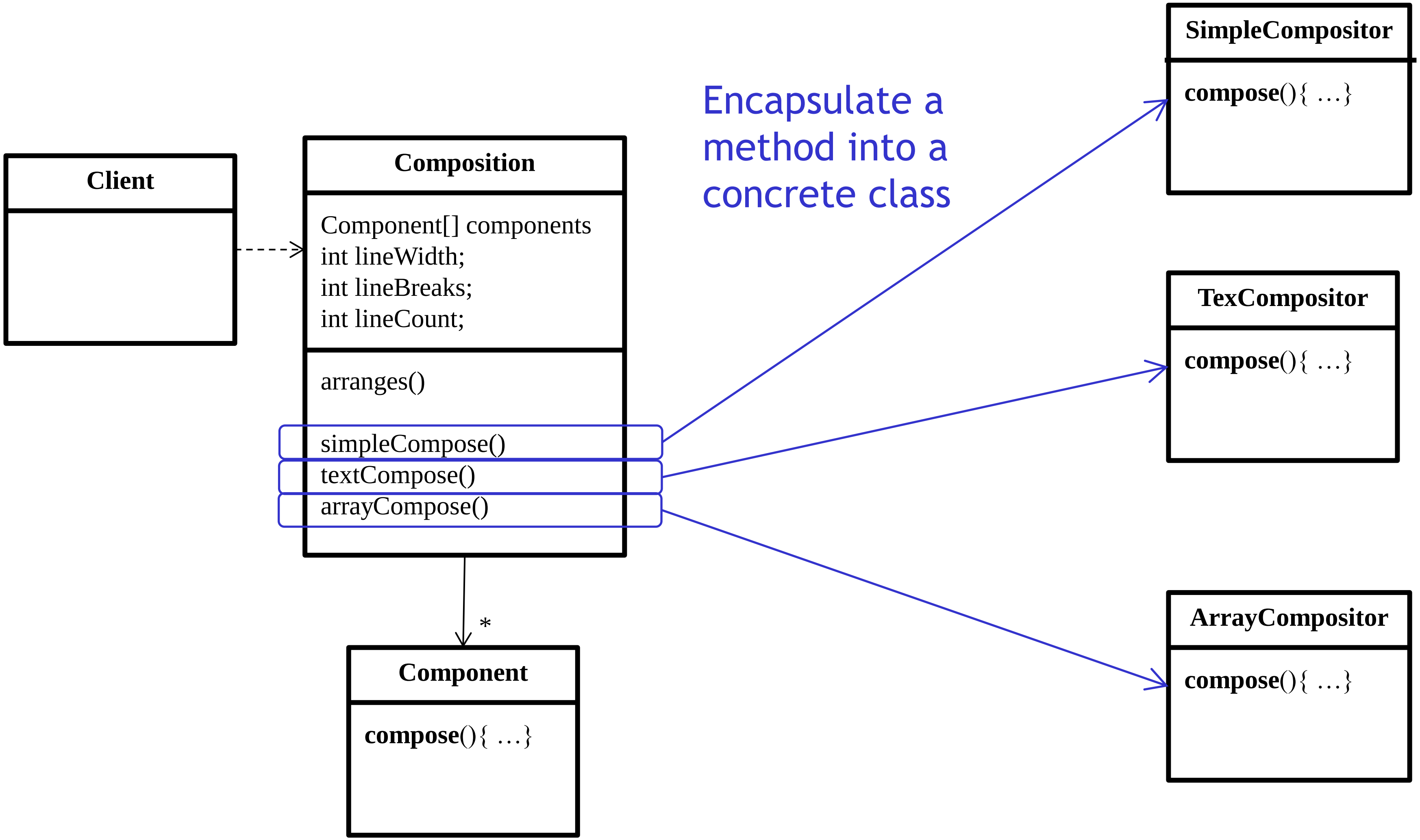
```
if(type.equals("simple")){
        simpleCompose();
}else if(type.equals("tex")){
        texCompose();
}else{
        arrayCompose();
}
```

Problem: It's difficult to add new algorithms and vary existing ones when line breaking is an integral part of Composition.

Refactoring by Design Principles

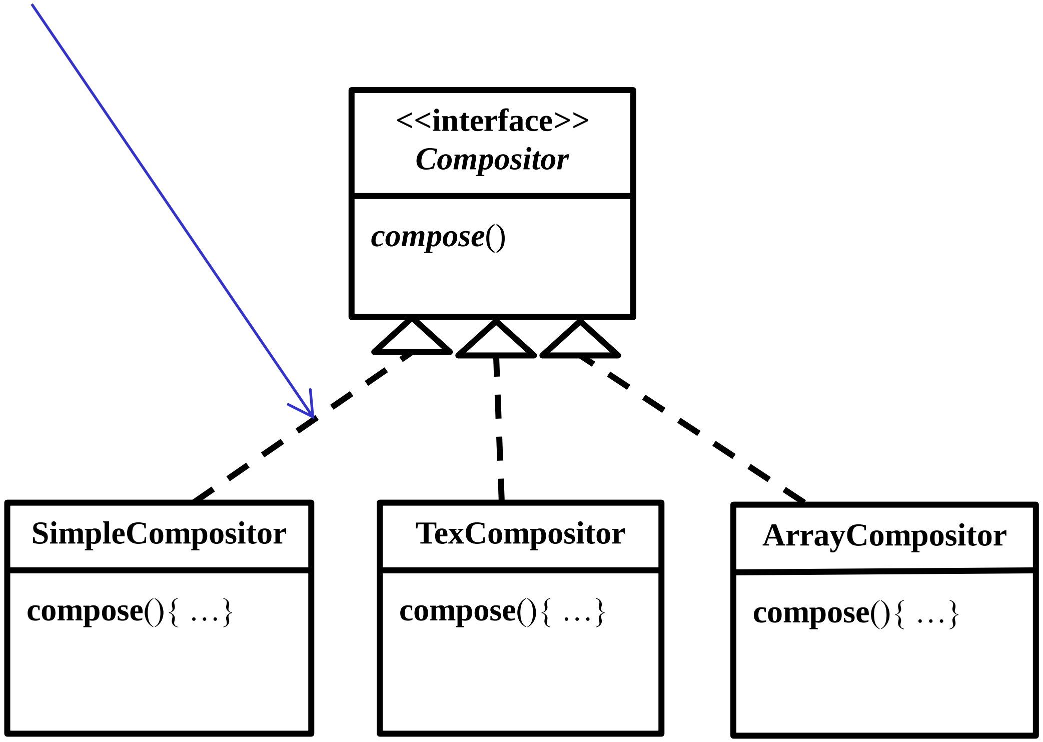
1. Encapsulate what varies
2. Generalize common features
3. Program to an interface, not an implementation

Encapsulate What Varies

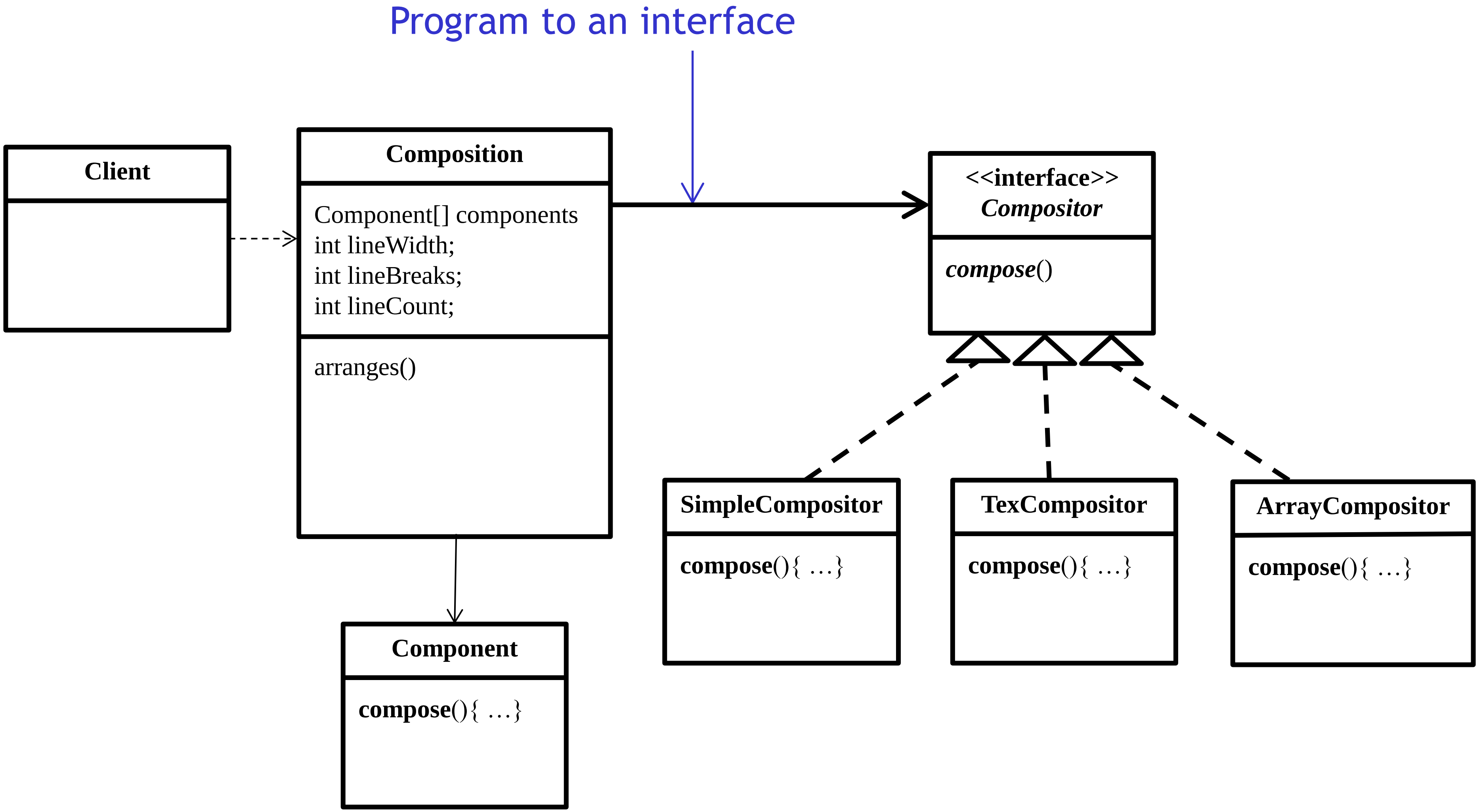


Generalize common features

Generalize common features



Program to an interface



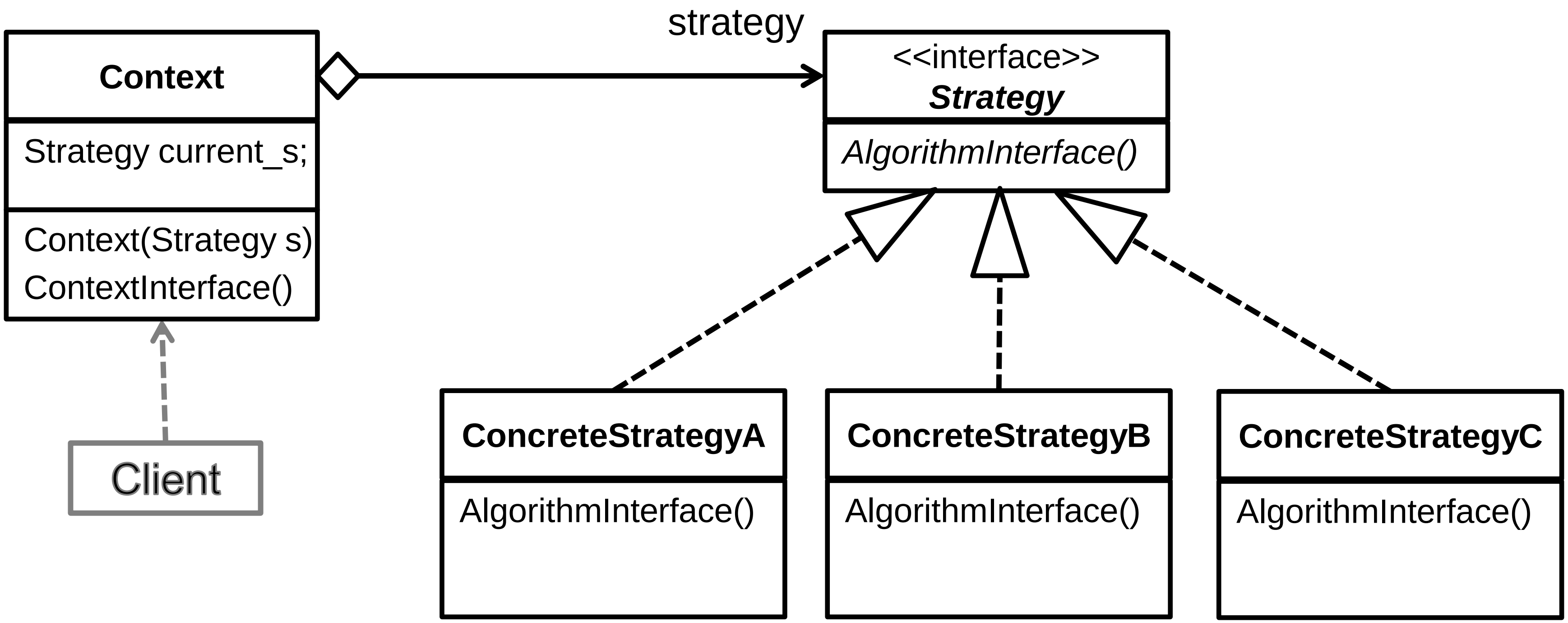
Recurrent Problems

- ❑ Multiple classes will be modified if new behaviors are to be added.
  - ➢ It's difficult to add new algorithms and vary existing ones.
- ❑ All duplicate code will be modified if the behavior is to be changed.
  - ➢ Different algorithms will be appropriate at different times.

Intent

- ❑ Define a family of algorithms, encapsulate each one, and make them interchangeable.
- ❑ Strategy lets the algorithm vary independently from clients that use it.

Strategy Pattern Structure1



Composite Pattern

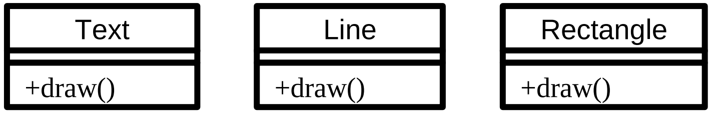
Design Aspect of Composite

Structure and composition of an object

Schematic Capture Systems (Composite)

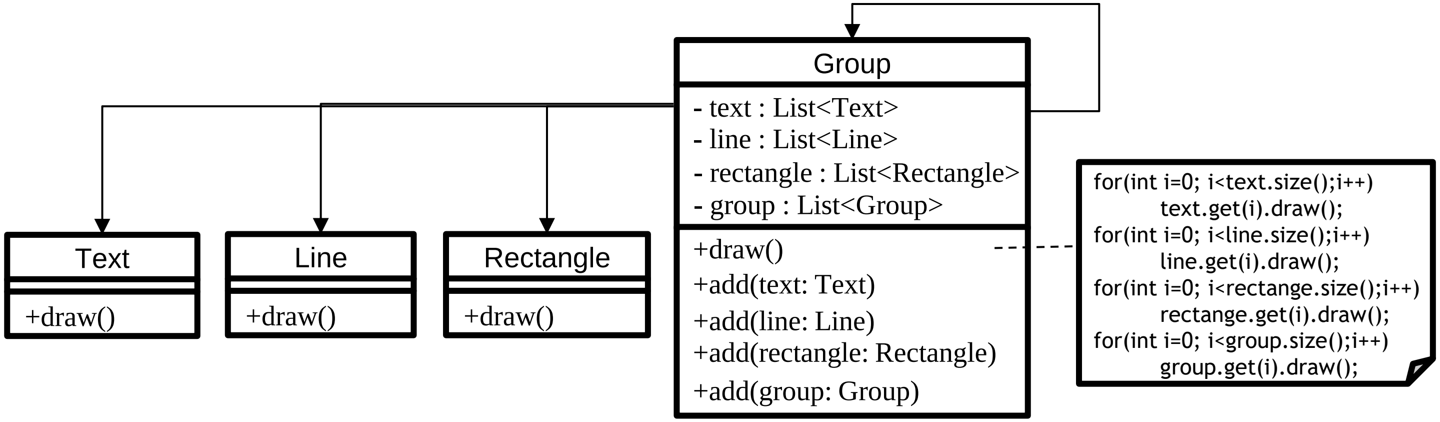
Requirements Statement$_1$

- ❑ In schematic capture application, there are some basic components can be drawn such as Text, Line and Rectangle.



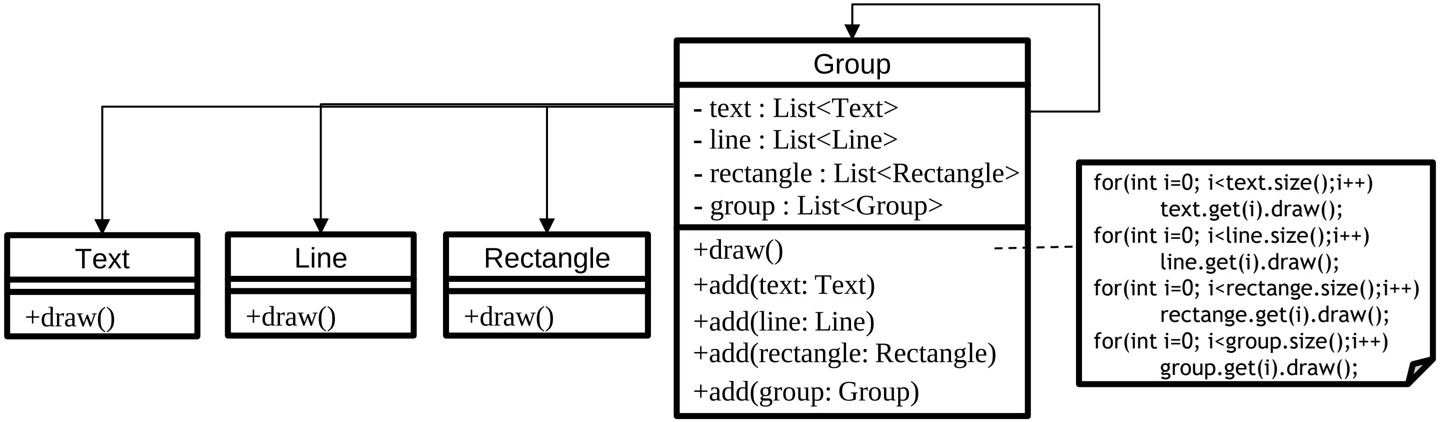
Requirements Statement$_2$

- The user can group basic components to form larger components, which in turn can be grouped to form still larger components.



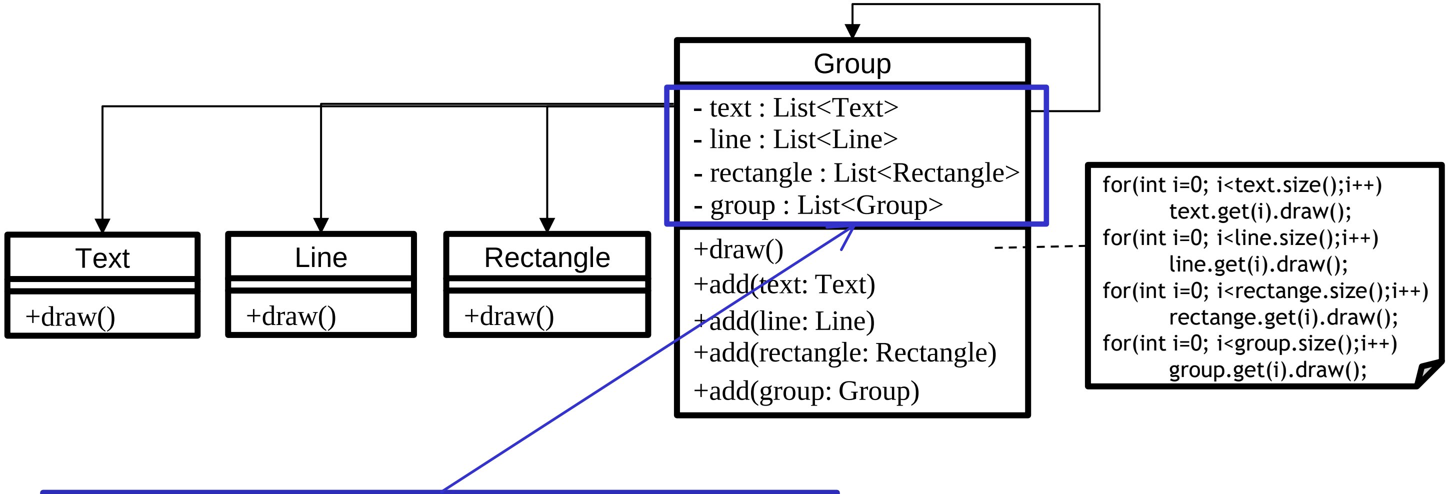
```
for(int i=0; i<text.size();i++)
        text.get(i).draw();
for(int i=0; i<line.size();i++)
        line.get(i).draw();
for(int i=0; i<rectange.size();i++)
        rectange.get(i).draw();
for(int i=0; i<group.size();i++)
        group.get(i).draw();
```

Initial Design



```
for(int i=0; i<text.size();i++)
        text.get(i).draw();
for(int i=0; i<line.size();i++)
        line.get(i).draw();
for(int i=0; i<rectange.size();i++)
        rectange.get(i).draw();
for(int i=0; i<group.size();i++)
        group.get(i).draw();
```

Problems with Initial Design



```
for(int i=0; i<text.size();i++)
        text.get(i).draw();
for(int i=0; i<line.size();i++)
        line.get(i).draw();
for(int i=0; i<rectange.size();i++)
        rectange.get(i).draw();
for(int i=0; i<group.size();i++)
        group.get(i).draw();
```

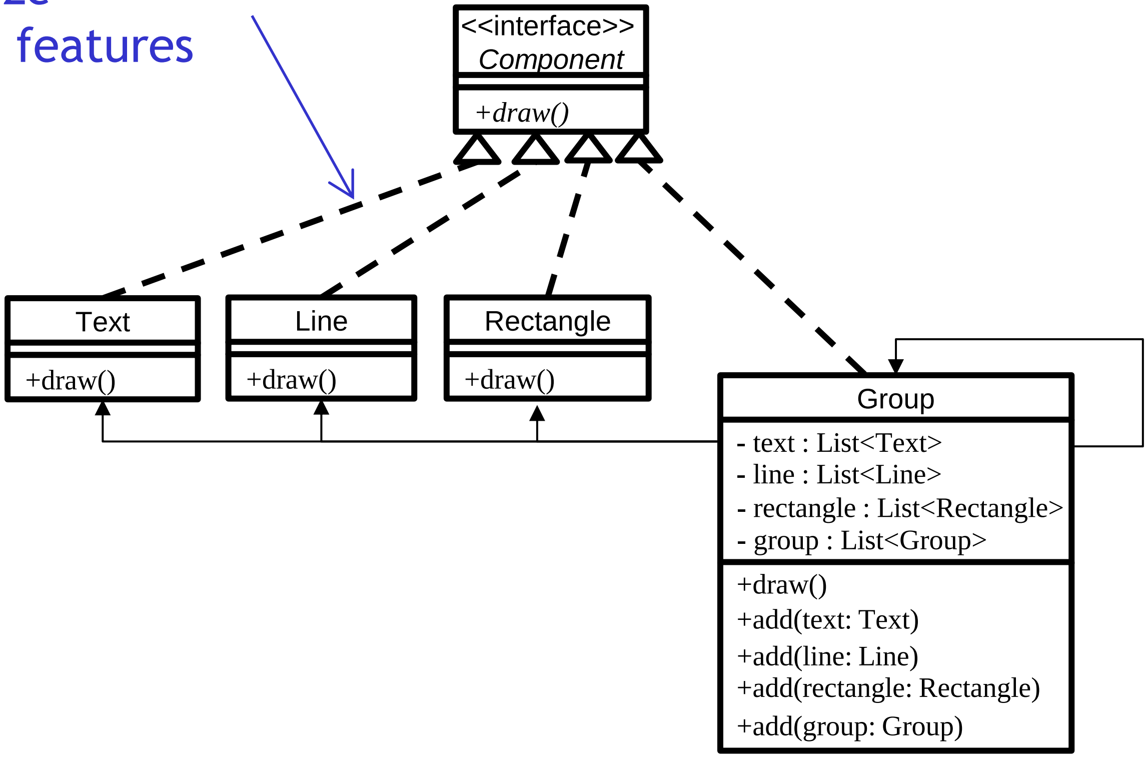
Problem : If a new requirement is to add a new basic component such as Triangle, then we need to modify Group to meet the new requirement.

Refactoring by Design Principles

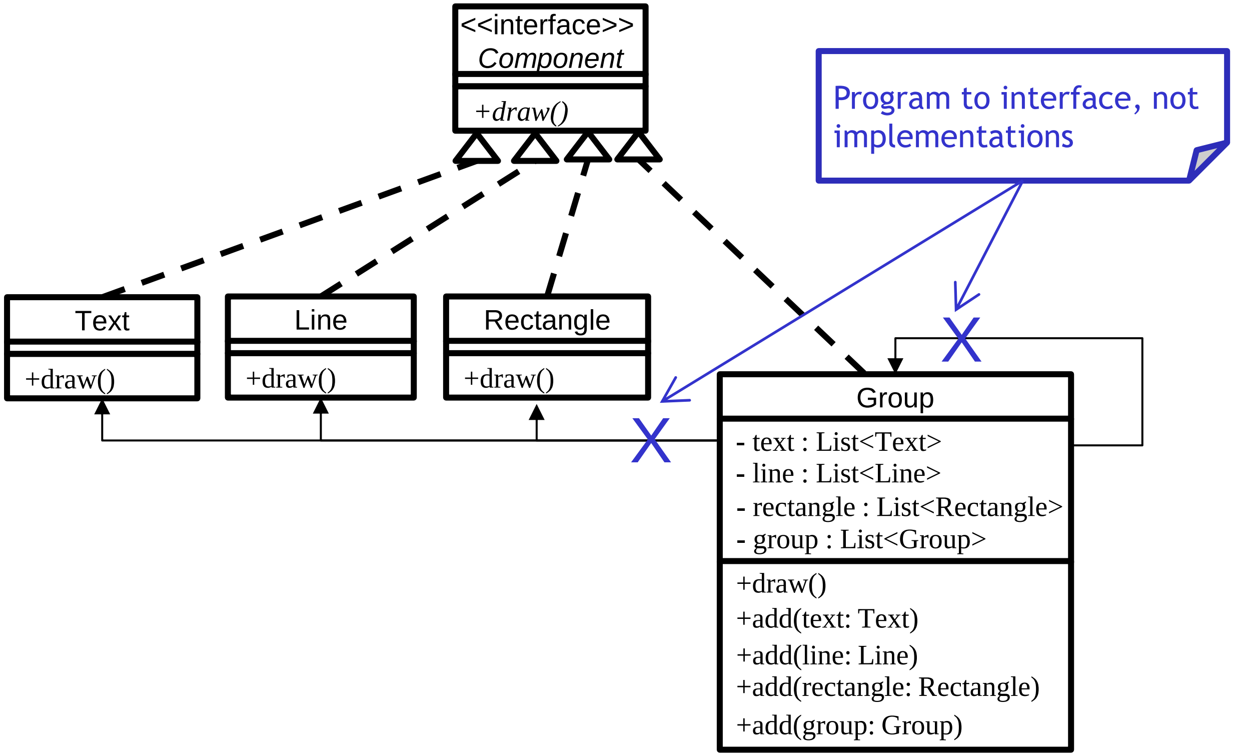
1. Generalize common features
2. Program to an interface, not an implementation.

Generalize common features

Generalize
common features

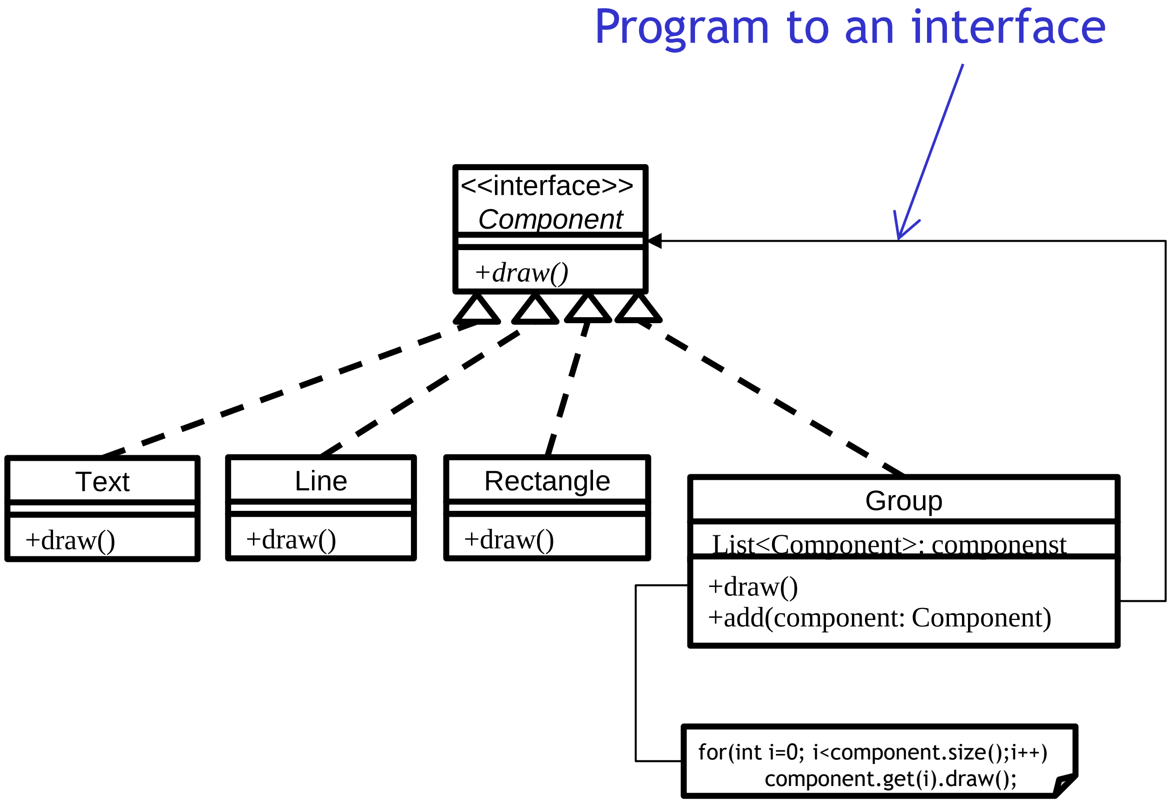


Program to an interface, not an implementation



Program to interface, not implementations

Program to an interface



```
for(int i=0; i<component.size();i++)
        component.get(i).draw();
```

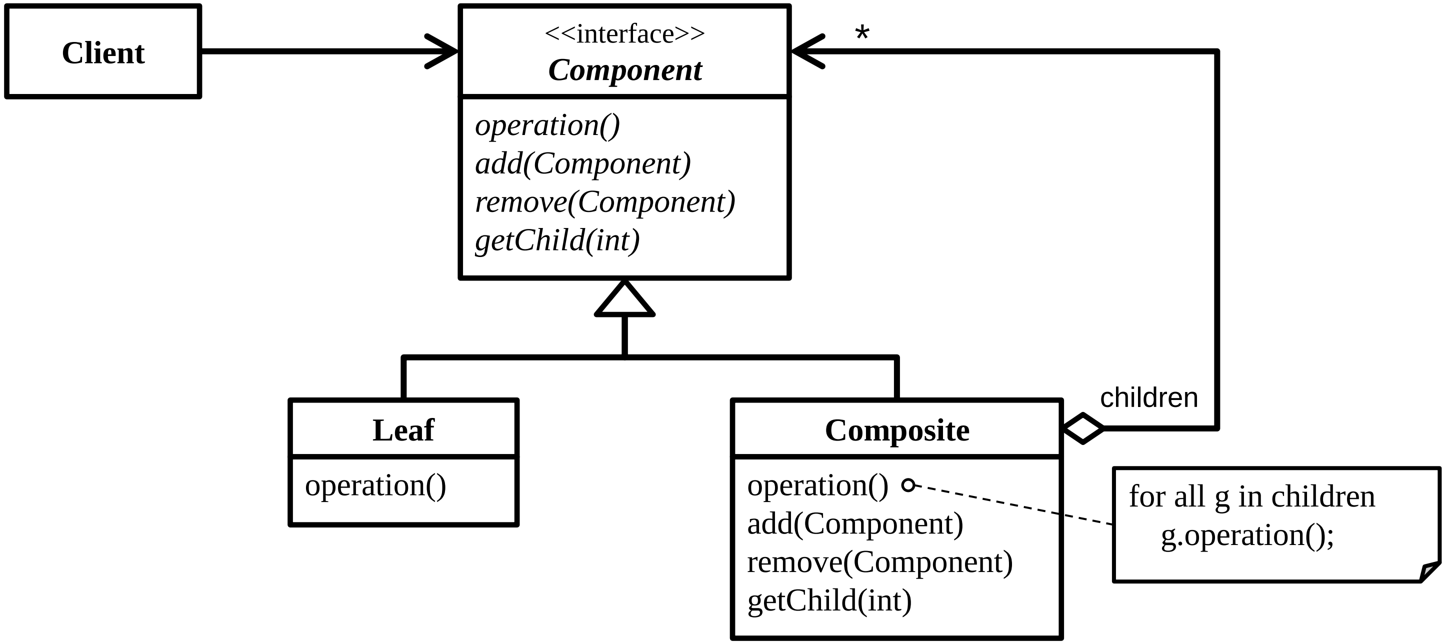
Recurrent Problem

- The user can group components to form larger components, which in turn can be grouped to form still larger components.
  - A simple implementation could define classes for primitives that act as containers for these primitives.
  - But there's a problem with this approach: Code that uses these classes must treat primitive and container objects differently, even if most of the time the user treats them identically.

Intent

- ❑ Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Composite Pattern Structure$_1$



Decorator Pattern

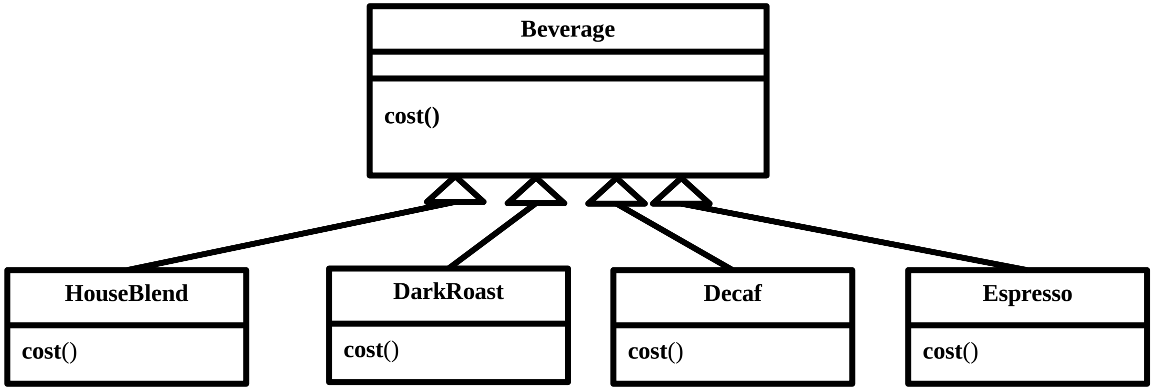
Design Aspect of Decorator

Responsibilities of an object without subclassing

Starbuzz Coffee (Decorator)

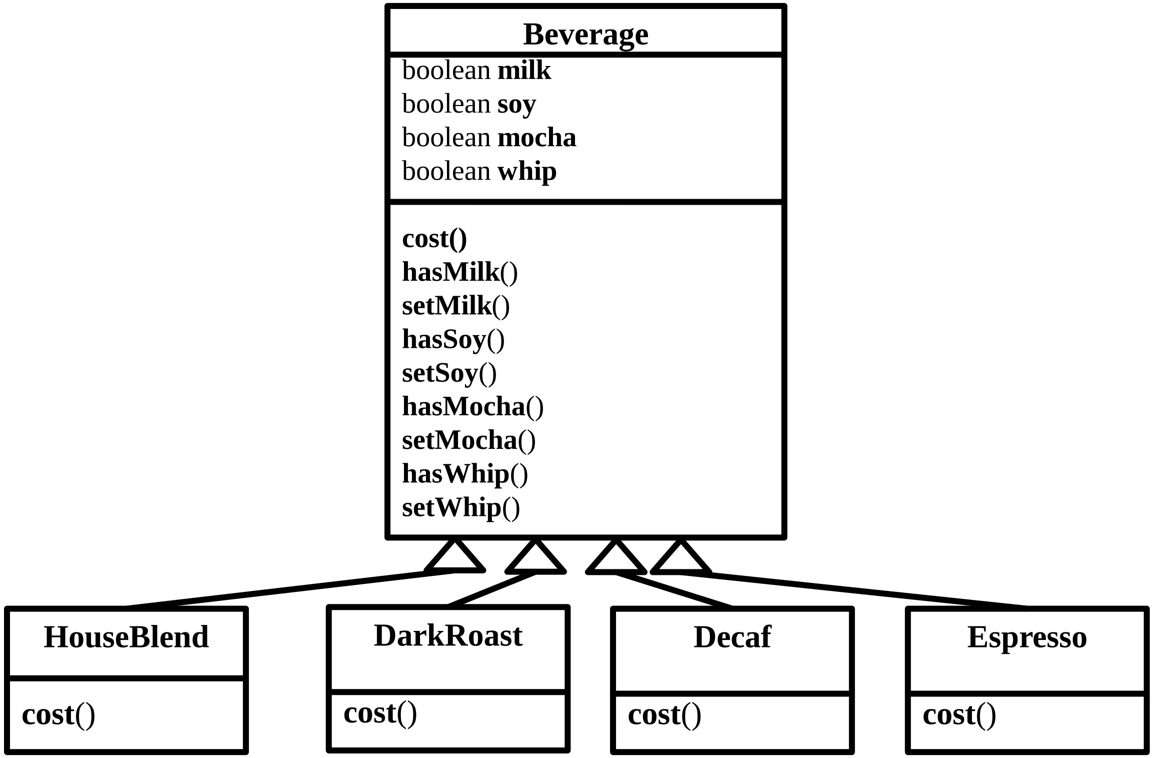
Requirements Statement1

- ❑ Starbuzz Coffee
  - ➢ Starbuzz Coffee shops are scrambling to update their ordering systems to match their beverage offerings (e.g. HouseBlend, DarkRoast, Decaf and Espresso) to summate how they cost.

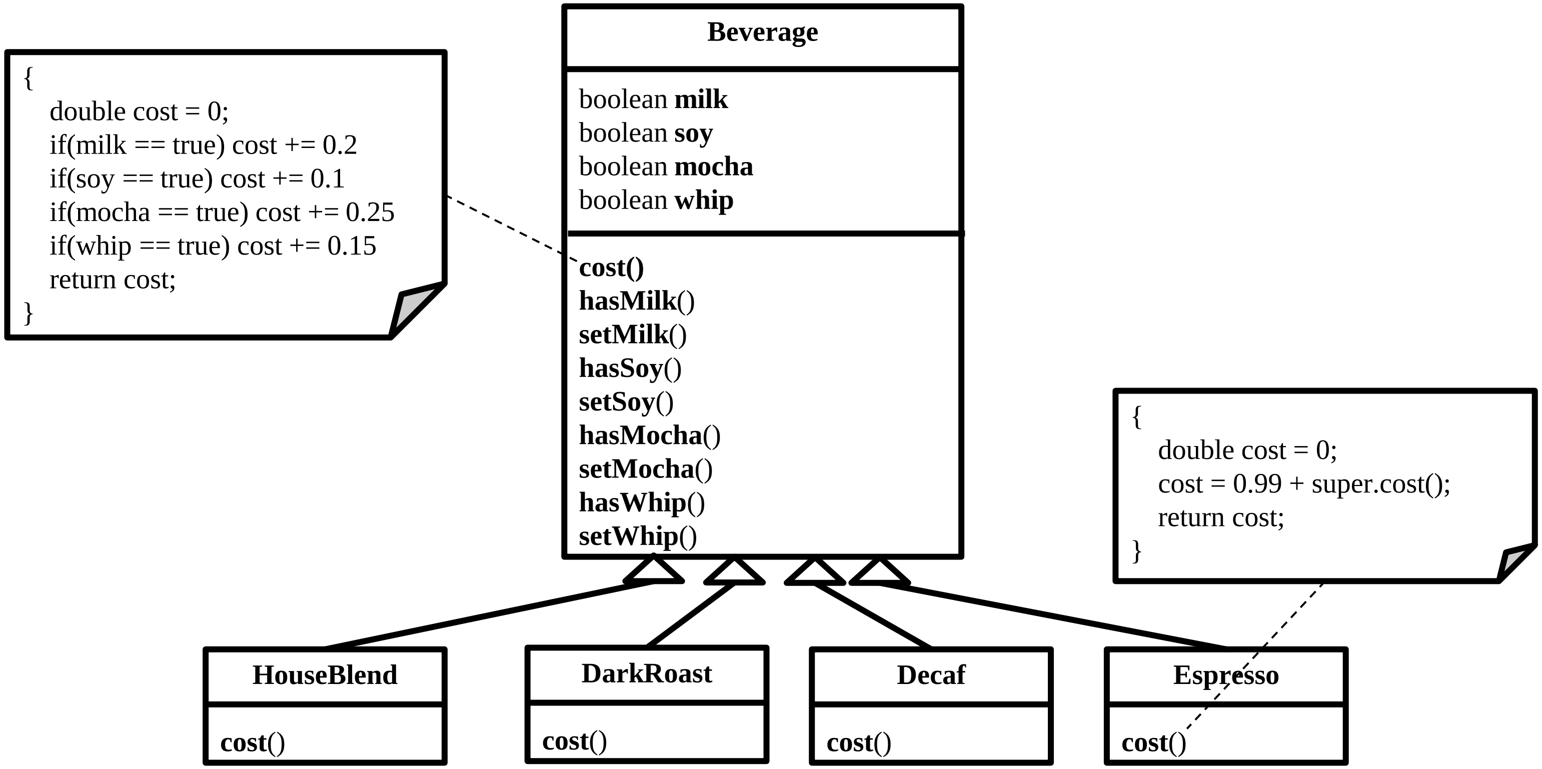


Requirements Statement₂

- ➢ In addition to your coffee, you can also ask for several condiments like steamed milk, soy, and mocha, and have these, so they really need to get them built into their order system



Initial Design - Class Diagram



```
{
   double cost = 0;
   if(milk == true) cost += 0.2
   if(soy == true) cost += 0.1
   if(mocha == true) cost += 0.25
   if(whip == true) cost += 0.15
   return cost;
}
```

Beverage

boolean **milk**
boolean **soy**
boolean **mocha**
boolean **whip**

cost()
hasMilk()
setMilk()
hasSoy()
setSoy()
hasMocha()
setMocha()
hasWhip()
setWhip()

```
{
   double cost = 0;
   cost = 0.99 + super.cost();
   return cost;
}
```

HouseBlend

cost()

DarkRoast

cost()

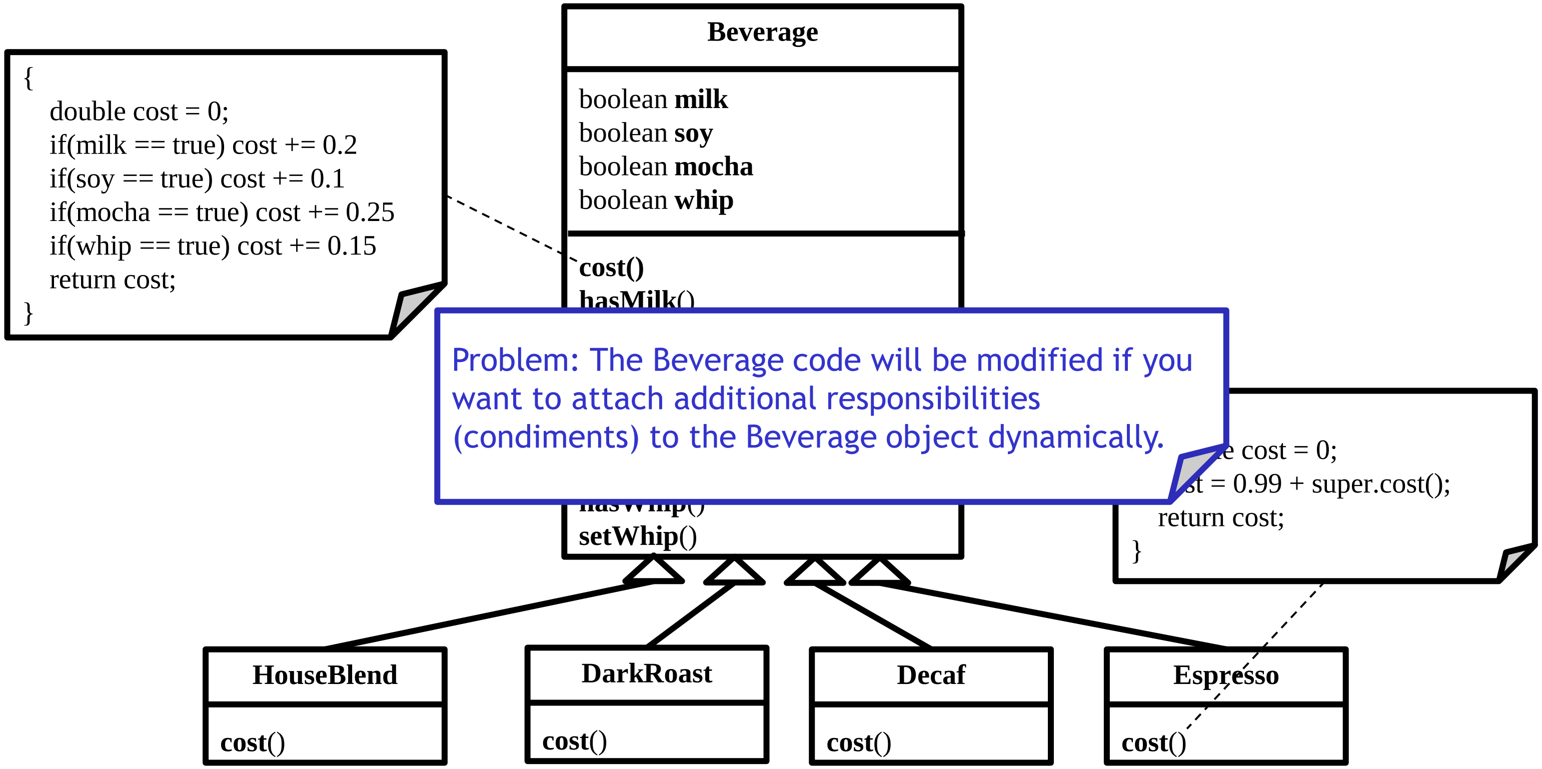
Decaf

cost()

Espresso

cost()

Problems with Initial Design



```
{
   double cost = 0;
   if(milk == true) cost += 0.2
   if(soy == true) cost += 0.1
   if(mocha == true) cost += 0.25
   if(whip == true) cost += 0.15
   return cost;
}
```

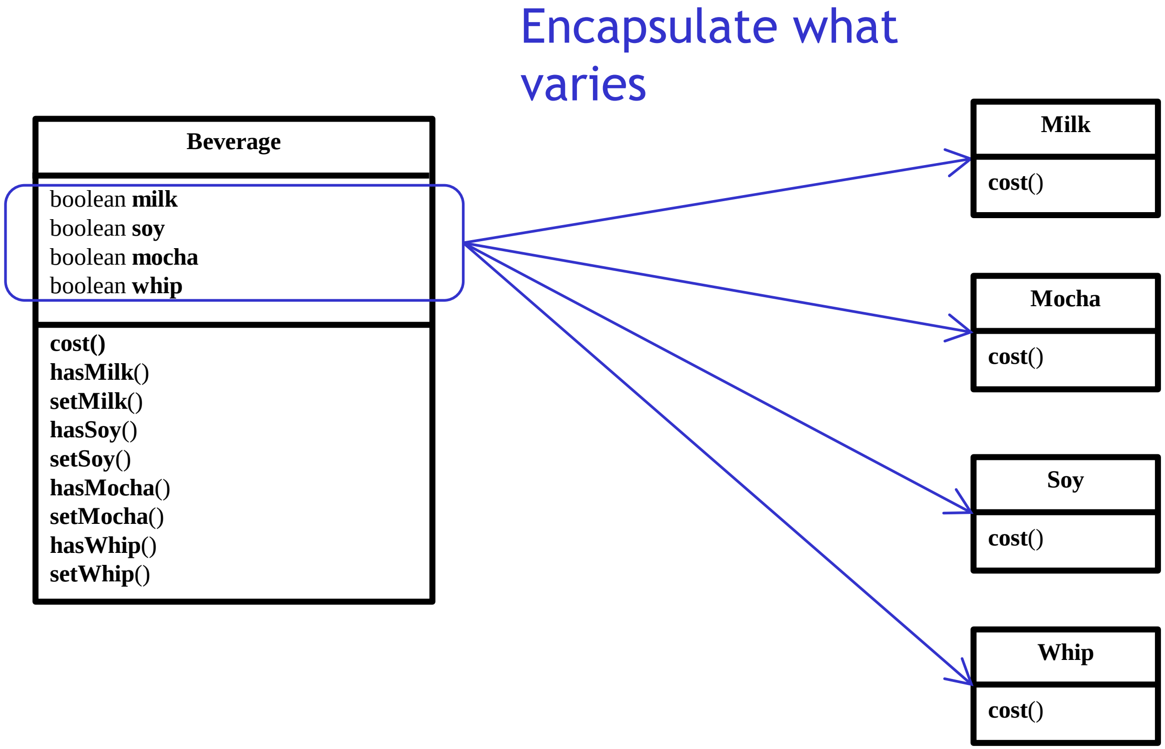
Problem: The Beverage code will be modified if you want to attach additional responsibilities (condiments) to the Beverage object dynamically.

```
   e cost = 0;
   st = 0.99 + super.cost();
   return cost;
}
```

Refactoring by Design Principles

1. Encapsulate what varies
2. Generalize common features
3. Program to an interface, not an implementation.

Encapsulate what varies



Encapsulate what varies

Beverage

boolean **milk**
boolean **soy**
boolean **mocha**
boolean **whip**

cost()
hasMilk()
setMilk()
hasSoy()
setSoy()
hasMocha()
setMocha()
hasWhip()
setWhip()

Milk

cost()

Mocha

cost()

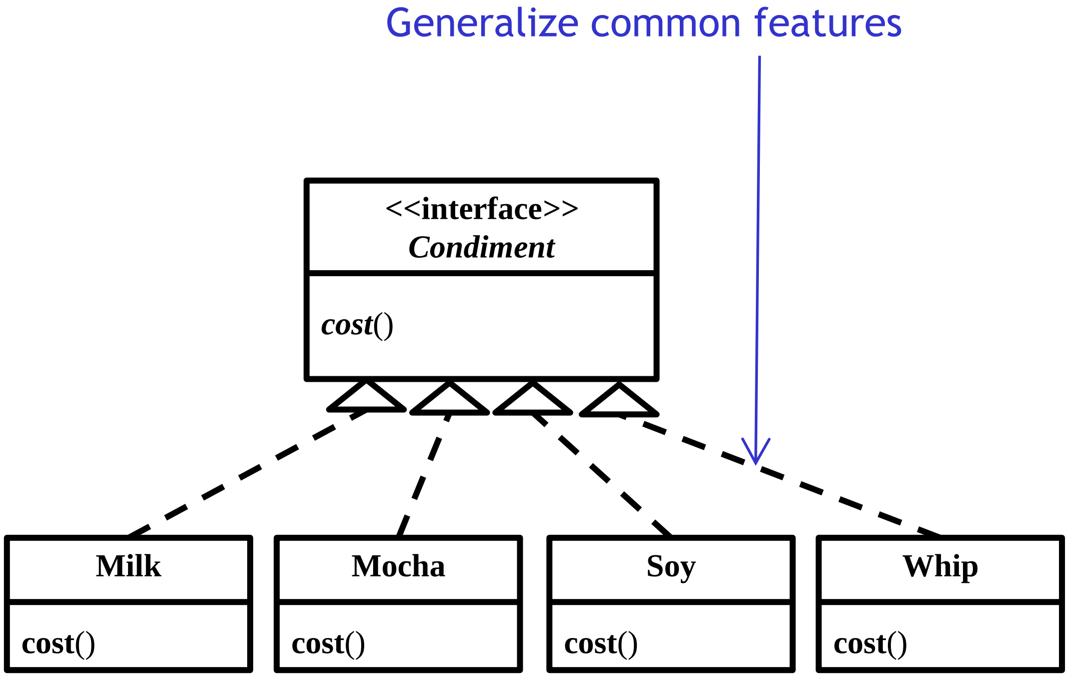
Soy

cost()

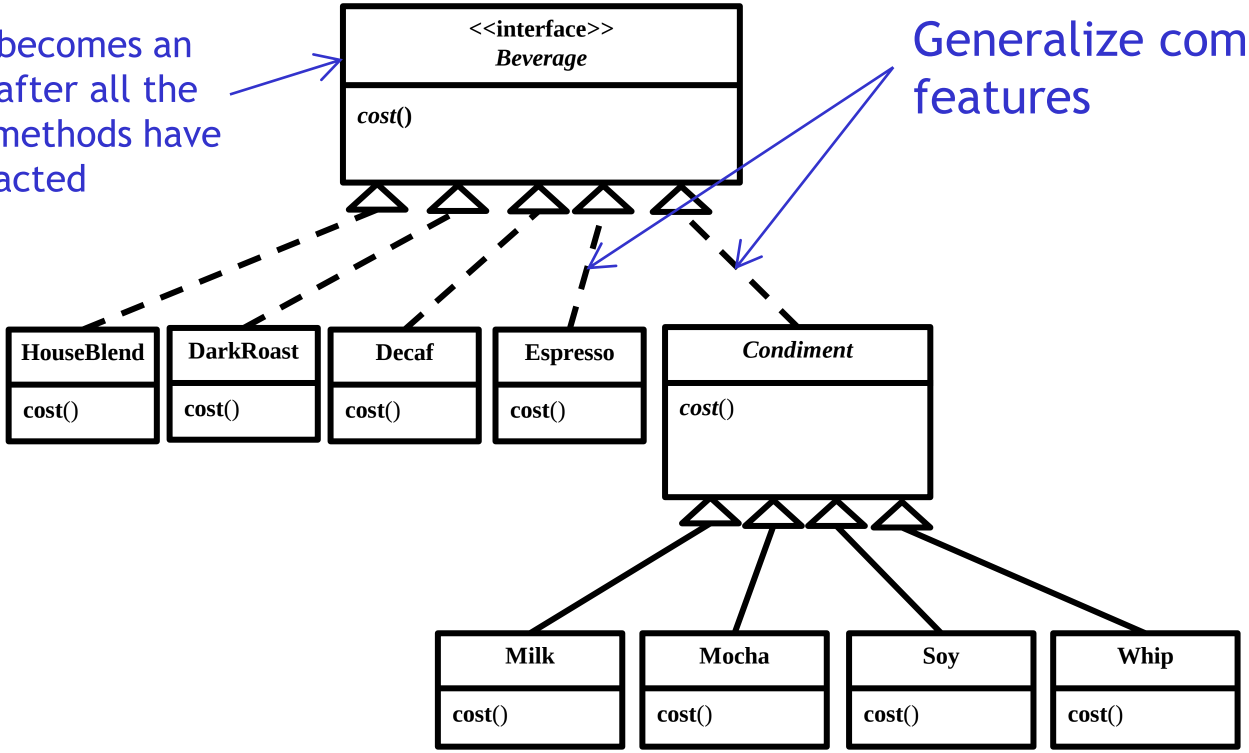
Whip

cost()

Generalize common features



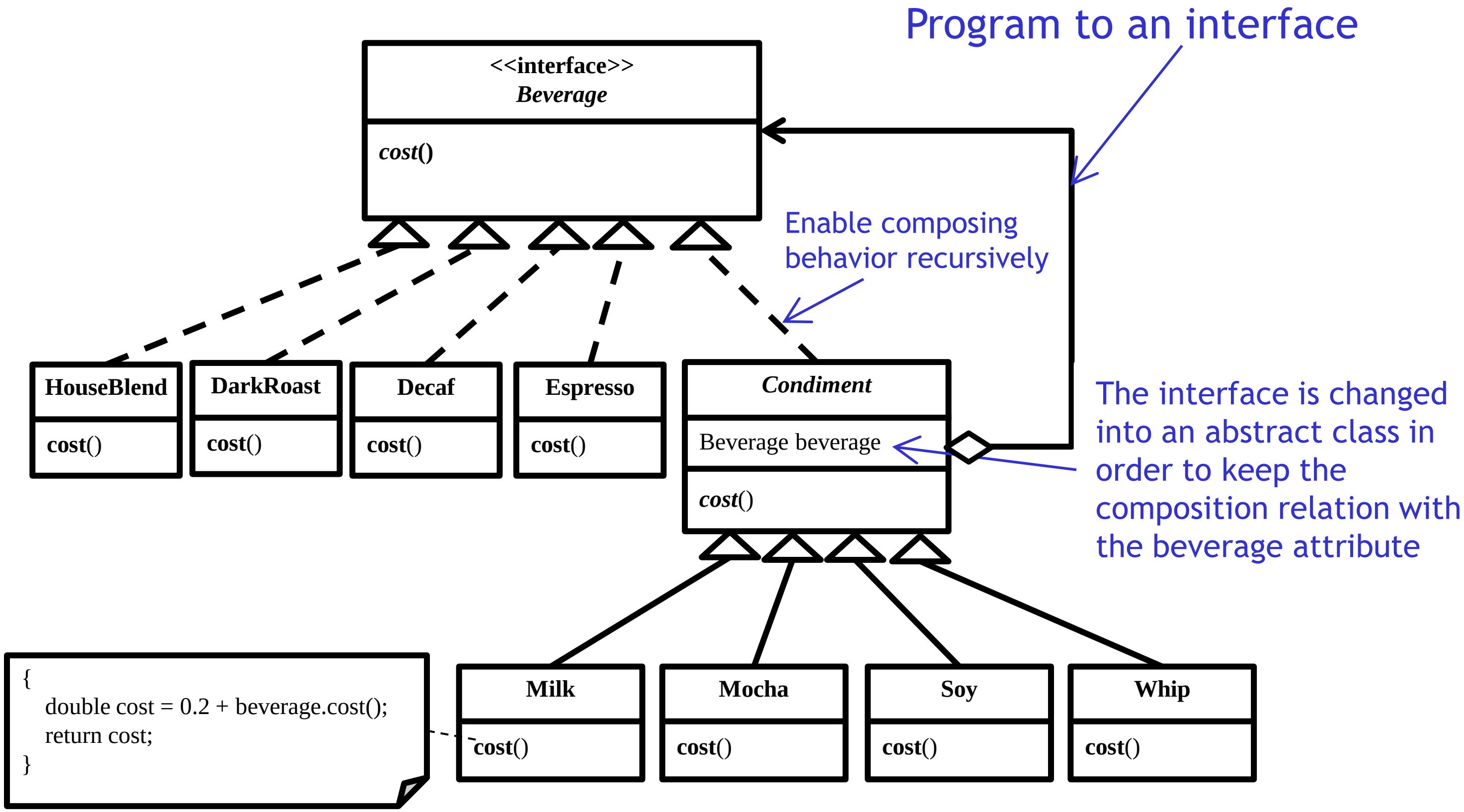
Generalize common features



Beverage becomes an interface after all the concrete methods have been extracted

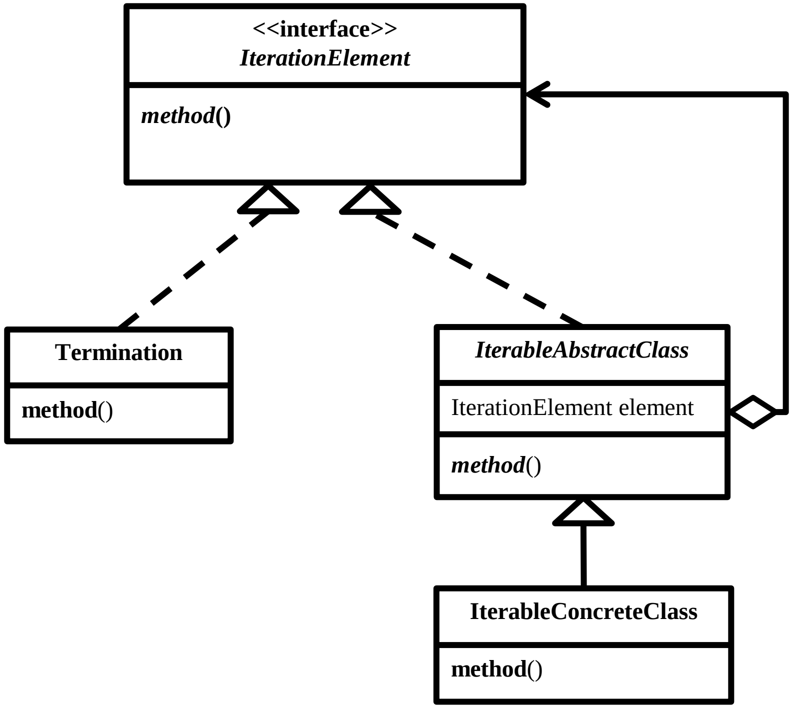
Generalize common features

Program to an interface

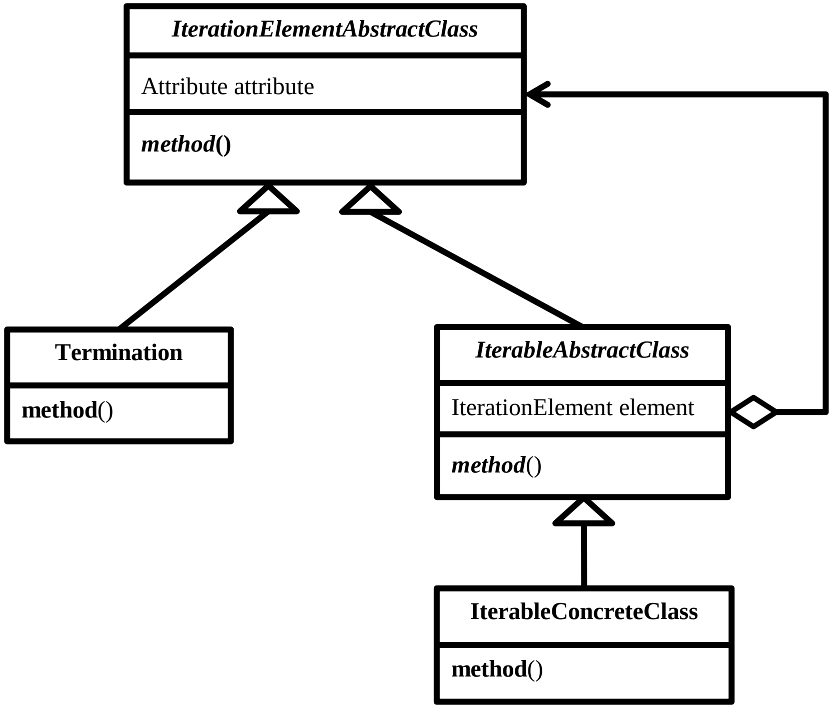


```
{
   double cost = 0.2 + beverage.cost();
   return cost;
}
```

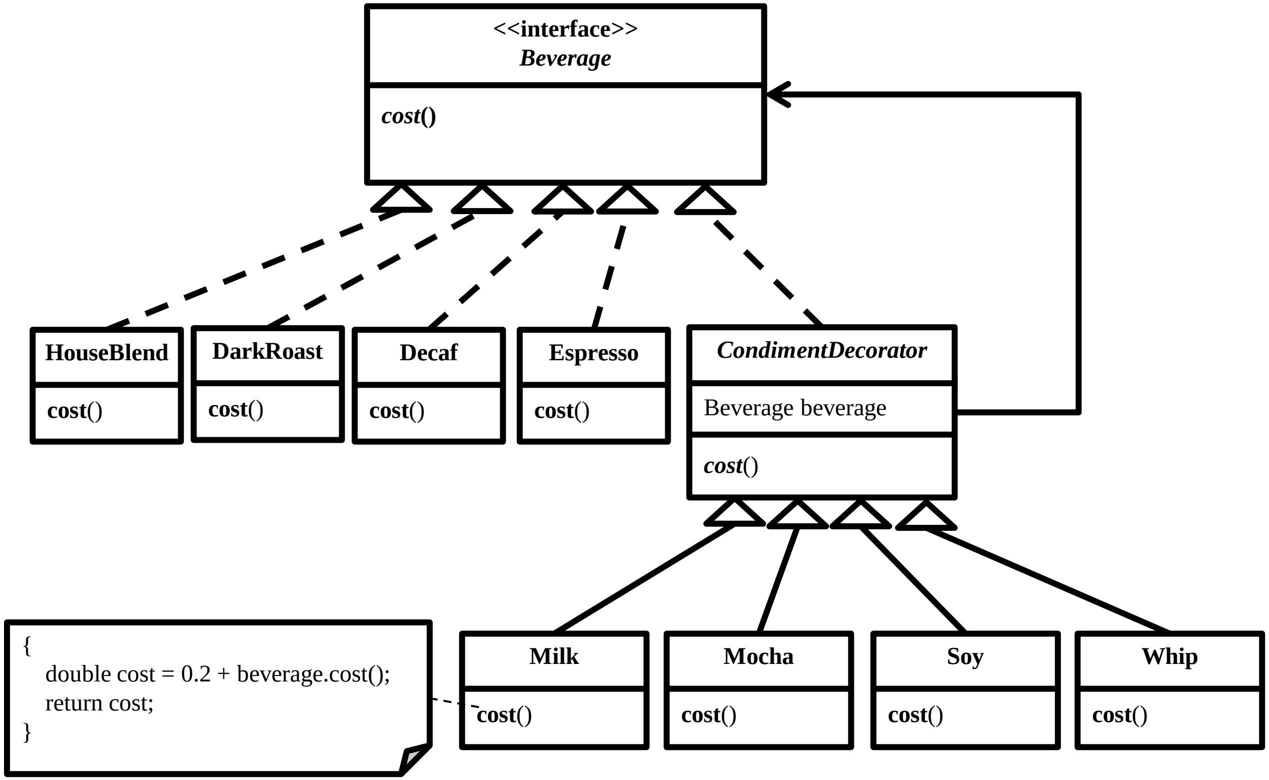
Recursive Design$_1$



Recursive Design$_2$



Refactored Design



```
{
    double cost = 0.2 + beverage.cost();
    return cost;
}
```

Recurrent Problem$_1$

- A class will be modified if you want to attach additional responsibilities (decorators) to an object dynamically.
  - Sometimes we want to add responsibilities to individual objects, not to an entire class. A graphical user interface toolkit.
  - For example, should let you add properties like borders or behaviors like scrolling to any user interface component.

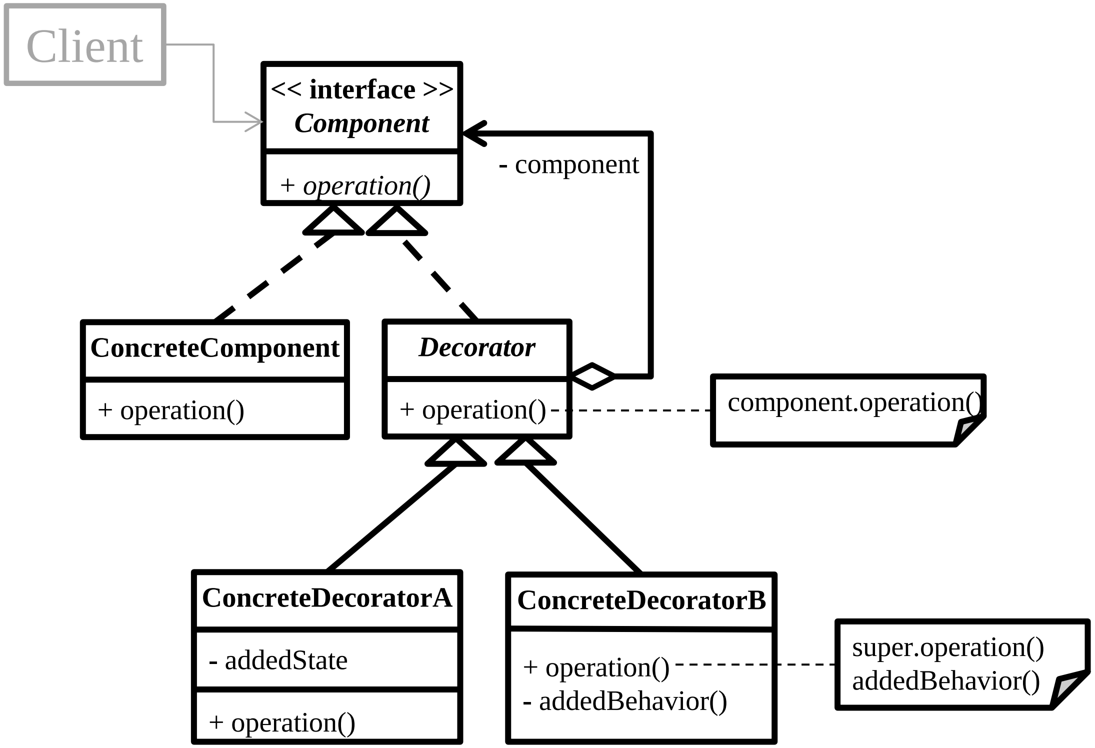
Recurrent Problem$_2$

- ❑ One way to add responsibilities is with inheritance. Inheriting a border from another class puts a border around every subclass instance.
- ❑ This is inflexible, however, because the choice of border is made statically.
- ❑ A client can't control how and when to decorate the component with a border.

Intent

- ❑ Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Decorator Pattern Structure$_1$



Composite vs. Decorator

Feature	**Composite Pattern**	**Decorator Pattern**
Purpose	Organize objects into a tree structure and treat them uniformly	Dynamically add functionality to objects
Structure	Focuses on organizing objects into a hierarchy	Focuses on wrapping objects to add behavior
Component Types	Supports composite (group) and leaf (individual) objects	Works on a single object at a time
Common Operations	Recursively traverses child components	Wraps and delegates calls to the base object
Typical Use Cases	File systems, organizational charts	GUI decorations, feature enhancements

Factory Method Pattern

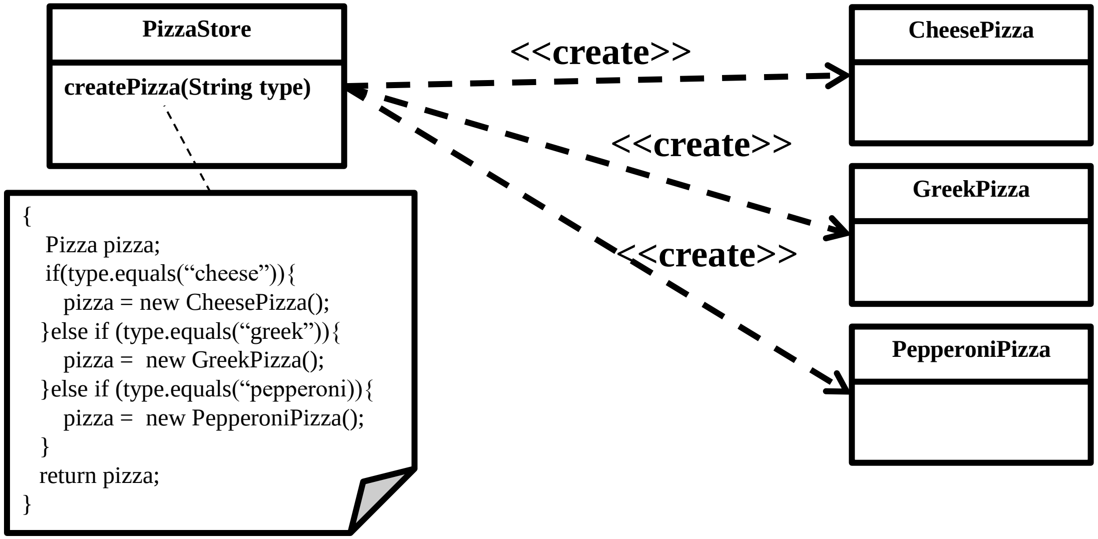
Design Aspect of Factory Method

Subclass of object that is instantiated

Pizza Store

Requirements Statement$_1$

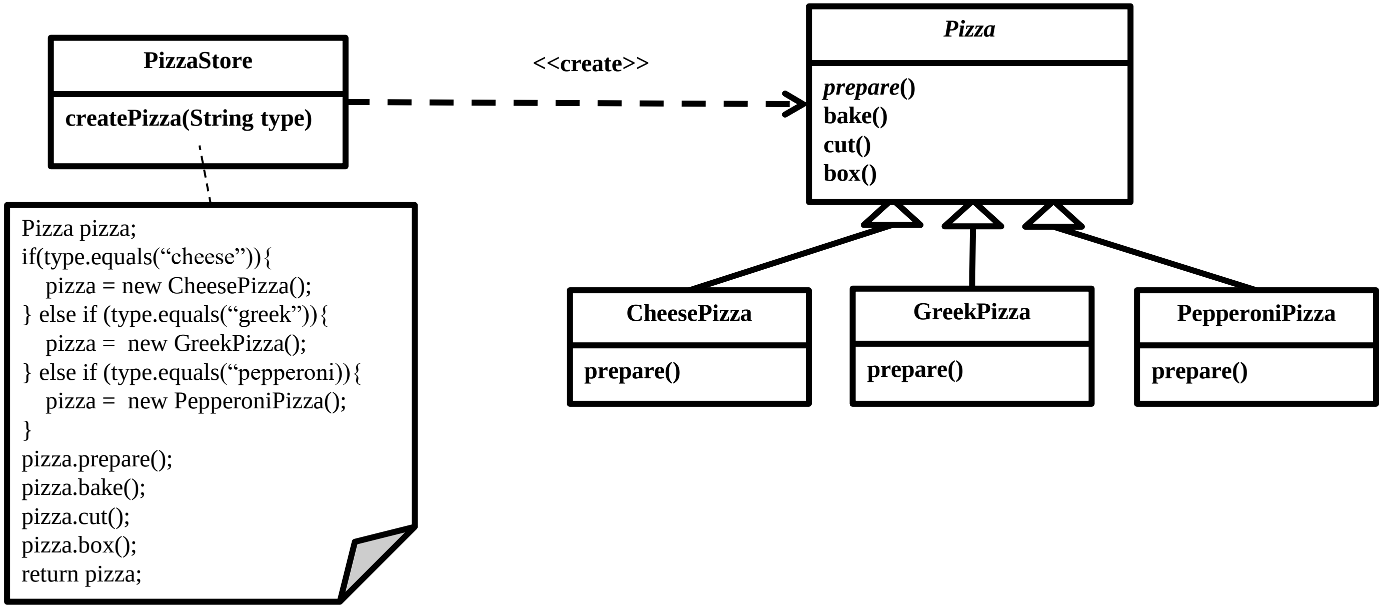
- ❑ Pizza Store
  - ➢ The store makes more than one type of pizza: Cheese Pizza, Greek Pizza, and Pepperoni Pizza



```
{
   Pizza pizza;
   if(type.equals(“cheese”)){
      pizza = new CheesePizza();
  }else if (type.equals(“greek”)){
      pizza =  new GreekPizza();
  }else if (type.equals(“pepperoni)){
      pizza =  new PepperoniPizza();
  }
  return pizza;
}
```

Requirements Statement$_2$

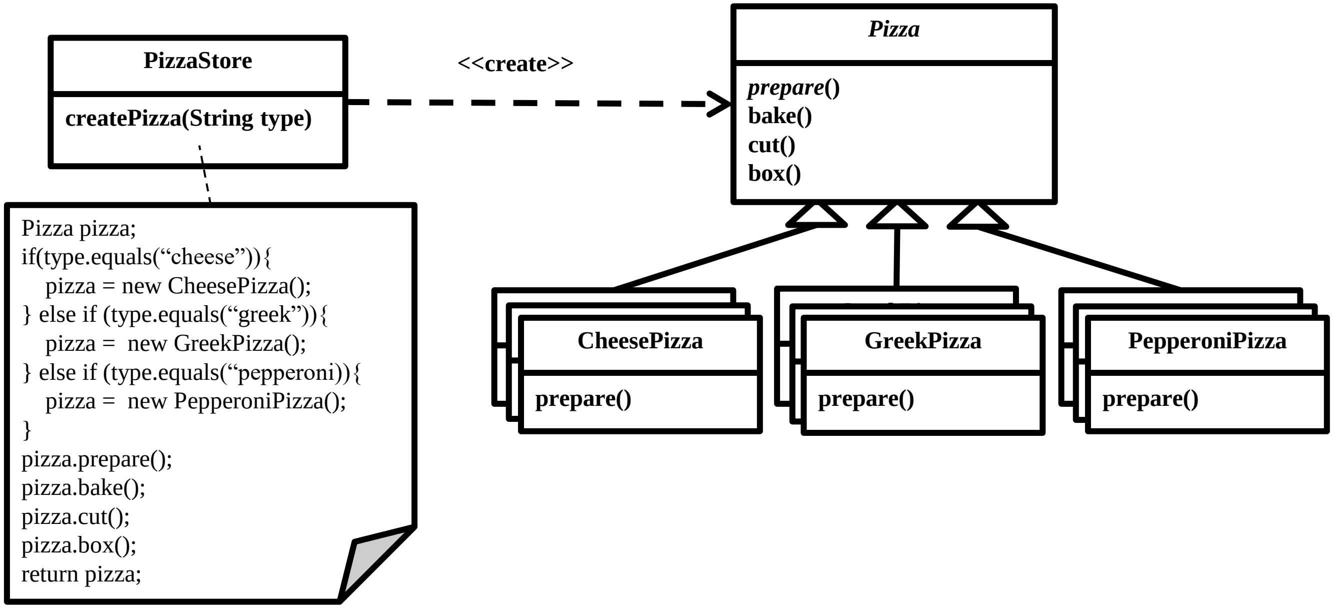
- Pizza Store
  - Each pizza has different way to prepare, and has the same way to bake, to cut, and to box.



```
Pizza pizza;
if(type.equals(“cheese”)){
    pizza = new CheesePizza();
} else if (type.equals(“greek”)){
    pizza =  new GreekPizza();
} else if (type.equals(“pepperoni)){
    pizza =  new PepperoniPizza();
}
pizza.prepare();
pizza.bake();
pizza.cut();
pizza.box();
return pizza;
```

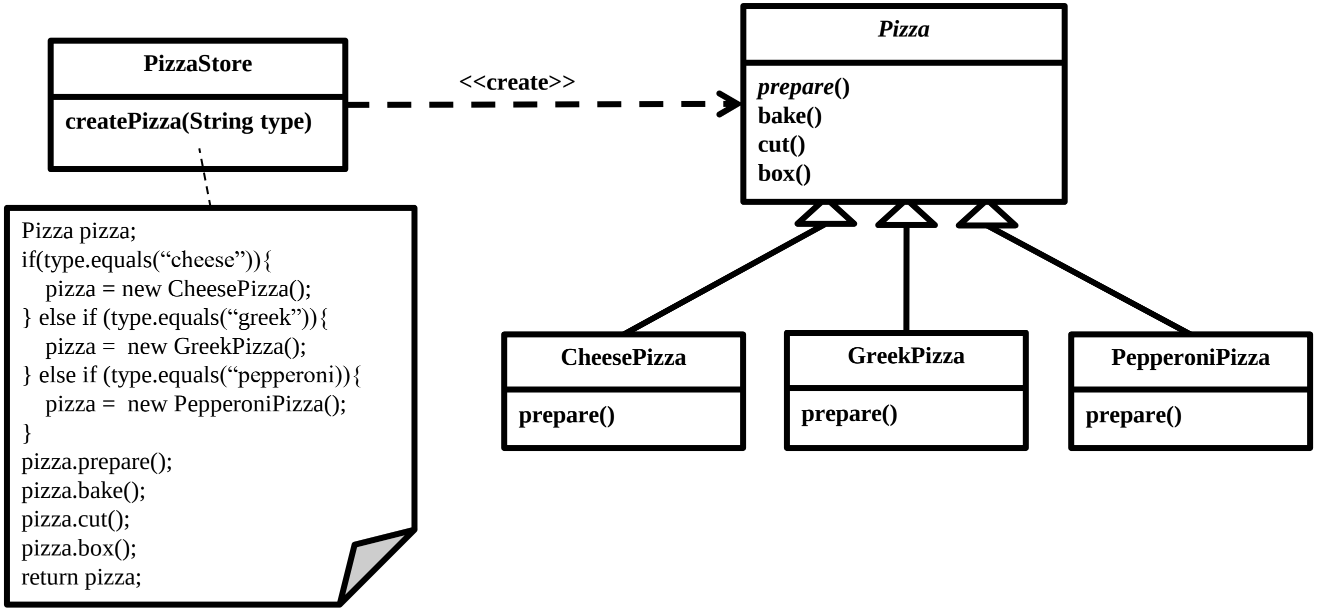
Requirements Statement$_3$

- ❑ Pizza Store
  - ➢ To make this store more competitive, you may add a new flavor of pizza or remove unpopular ones.



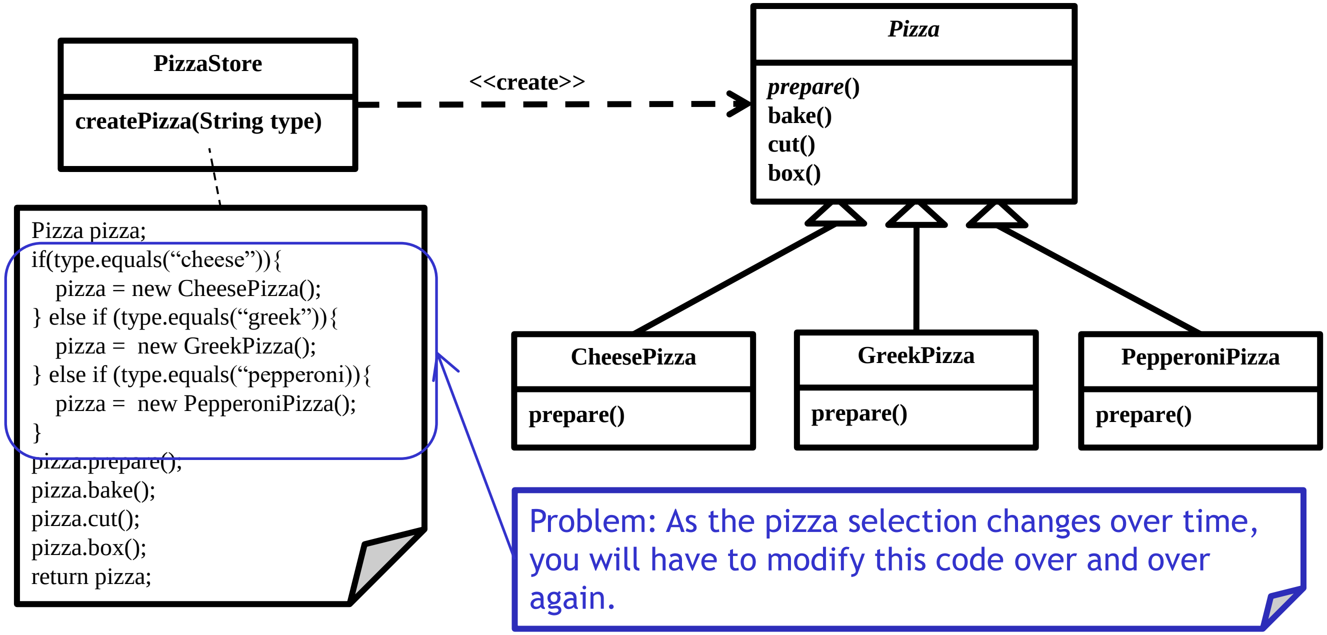
```
Pizza pizza;
if(type.equals(“cheese”)){
    pizza = new CheesePizza();
} else if (type.equals(“greek”)){
    pizza =  new GreekPizza();
} else if (type.equals(“pepperoni)){
    pizza =  new PepperoniPizza();
}
pizza.prepare();
pizza.bake();
pizza.cut();
pizza.box();
return pizza;
```

Initial Design - Class Diagram



```
Pizza pizza;
if(type.equals(“cheese”)){
    pizza = new CheesePizza();
} else if (type.equals(“greek”)){
    pizza =  new GreekPizza();
} else if (type.equals(“pepperoni)){
    pizza =  new PepperoniPizza();
}
pizza.prepare();
pizza.bake();
pizza.cut();
pizza.box();
return pizza;
```

Problems with Initial Design



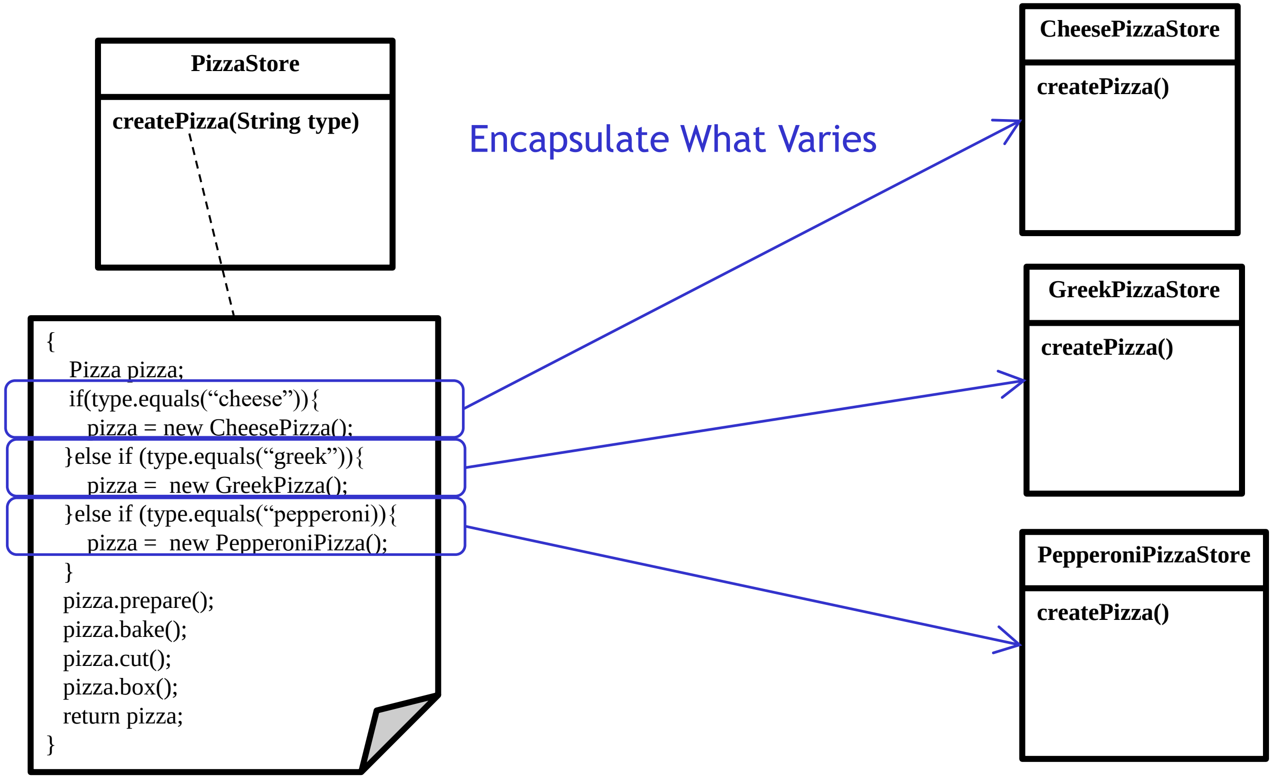
```
Pizza pizza;
if(type.equals(“cheese”)){
   pizza = new CheesePizza();
} else if (type.equals(“greek”)){
   pizza =  new GreekPizza();
} else if (type.equals(“pepperoni)){
   pizza =  new PepperoniPizza();
}
pizza.prepare();
pizza.bake();
pizza.cut();
pizza.box();
return pizza;
```

Problem: As the pizza selection changes over time, you will have to modify this code over and over again.

Refactoring by Design Principles

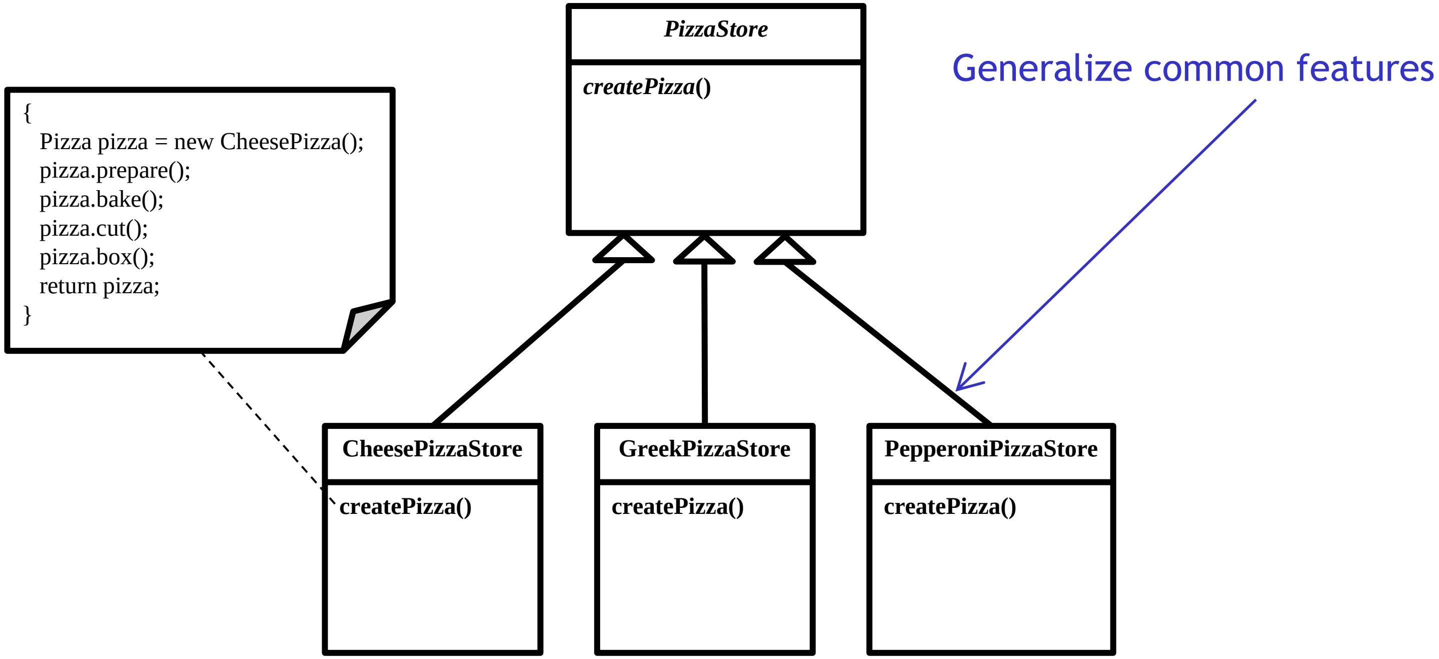
1. Encapsulate what varies
2. Generalize common features

Encapsulate What Varies



```
{
    Pizza pizza;
    if(type.equals(“cheese”)){
      pizza = new CheesePizza();
   }else if (type.equals(“greek”)){
      pizza =  new GreekPizza();
   }else if (type.equals(“pepperoni)){
      pizza =  new PepperoniPizza();
   }
   pizza.prepare();
   pizza.bake();
   pizza.cut();
   pizza.box();
   return pizza;
}
```

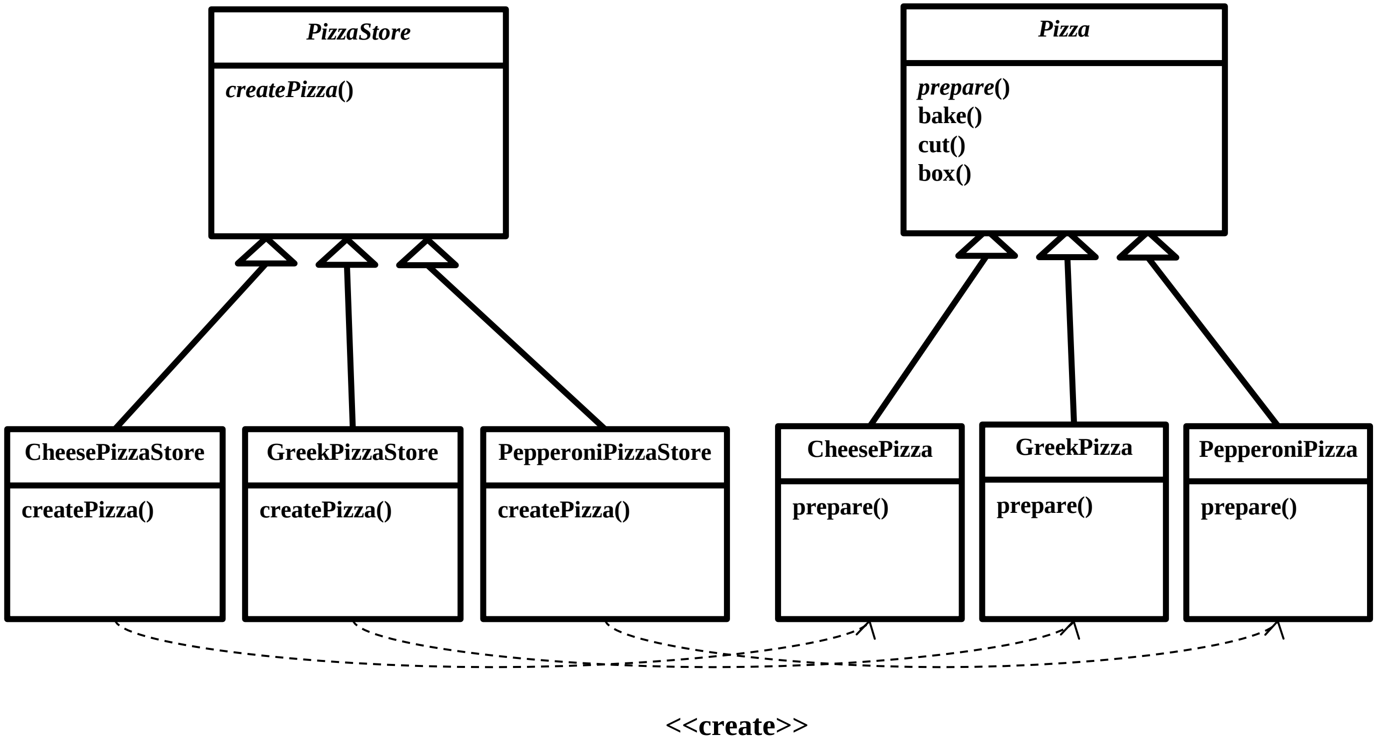
Generalize common features



```
{
   Pizza pizza = new CheesePizza();
   pizza.prepare();
   pizza.bake();
   pizza.cut();
   pizza.box();
   return pizza;
}
```

Generalize common features

Refactored Design



Recurrent Problem

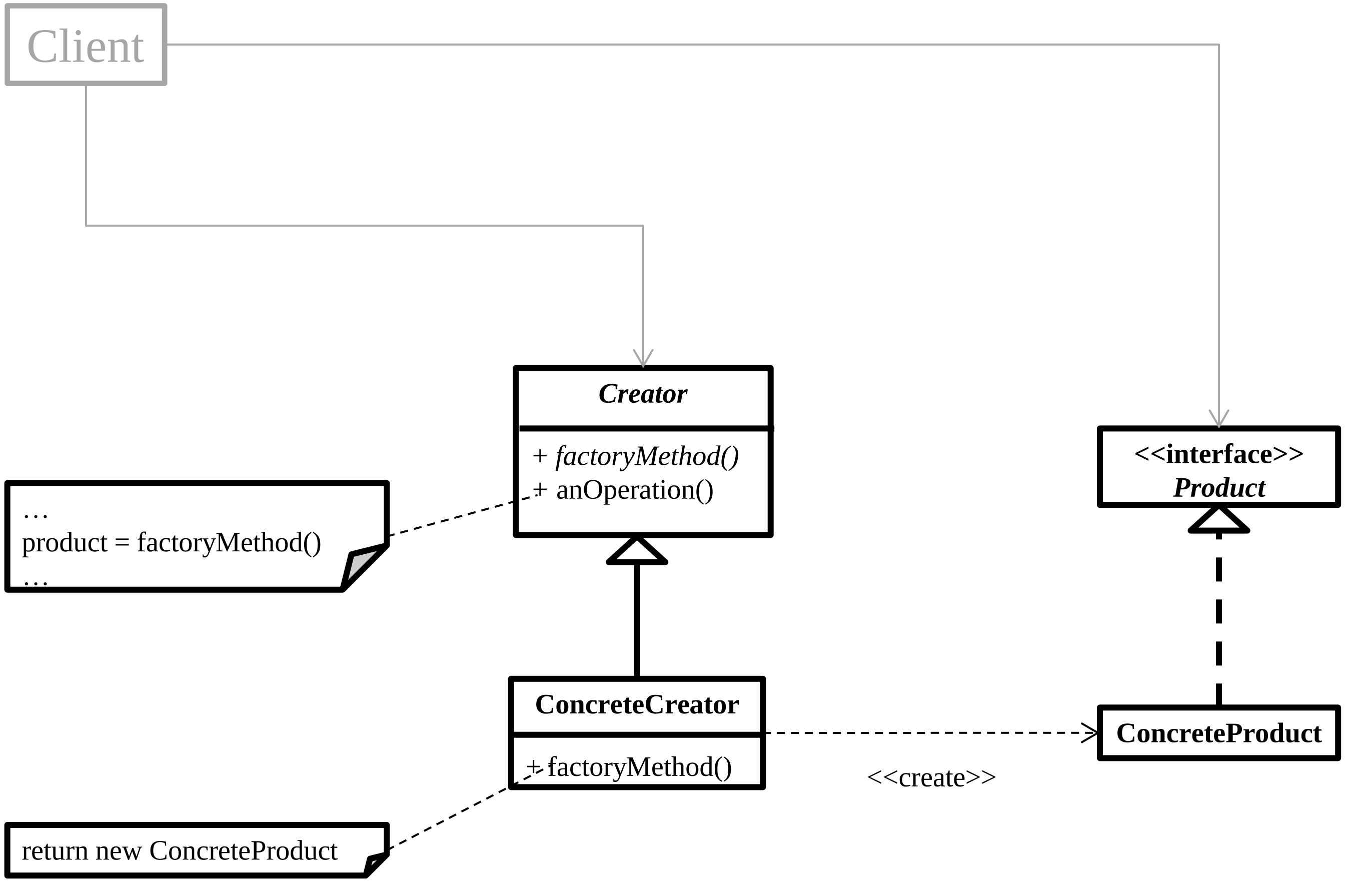
- As the objects being created changes over time, we need to modify the code of the creator object for the creations over and over again.
  - We need to encapsulate the knowledge of which objects to create and moves this knowledge out of the creator object.

Factory Method Pattern

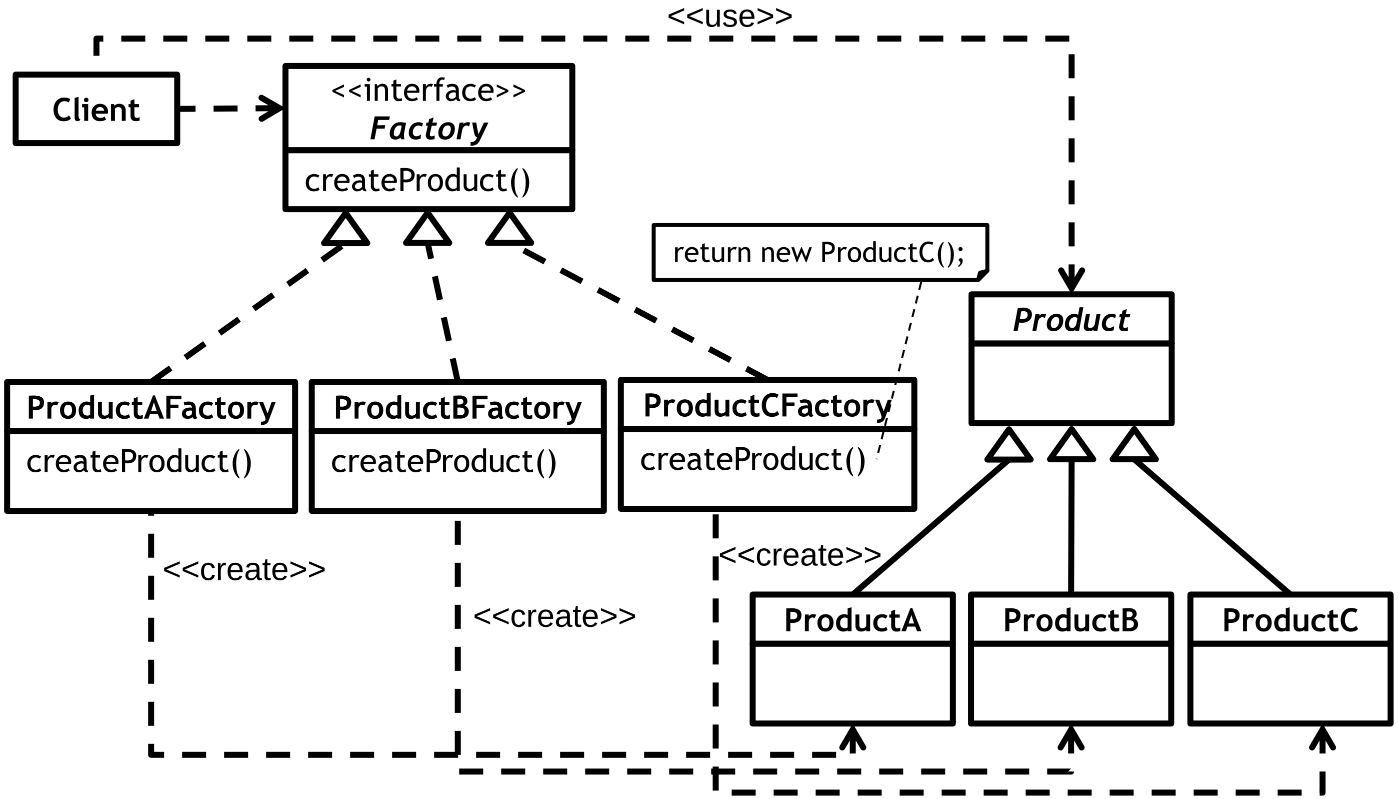
❑ Intent

- Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Factory Method Pattern Structure$_1$

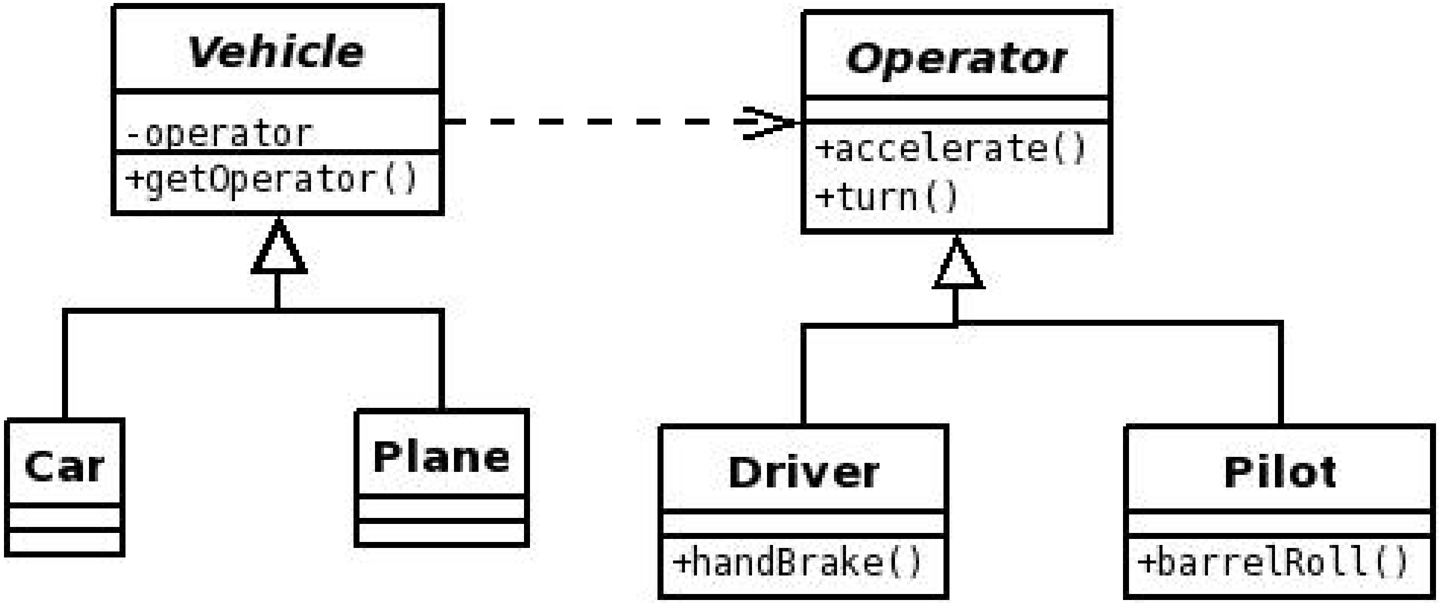


Factory Method Pattern Structure$_2$



補充：Parallel Inheritances Hierarchies問題

- Whenever you create a subclass for a class, you find yourself needing to create a subclass for another class.
- 問題：無法滿足兩個樹底下的物件互相有特定配對依賴關係的要求

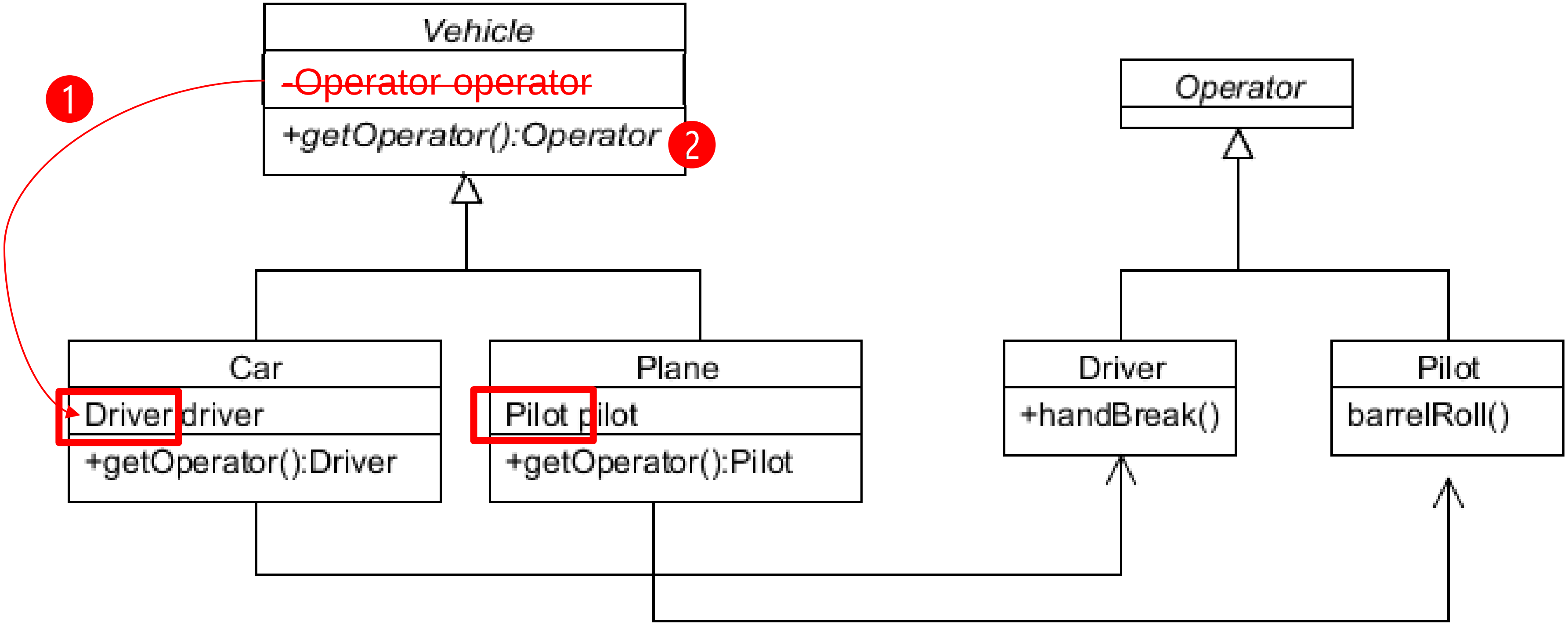


一個Car物件的operator屬性狀態可能會被設定為一個Pilot物件

From: https://oowisdom.csse.canterbury.ac.nz/index.php/Parallel_hierarchies_problem

Refactoring by *Defer Identification of State Variables Pattern*

- 第一步(屬性降階層)
  - 將Vehicle的operator屬性移除，並在Car與Plane中各別加入欲配對的屬性型態
- 第二步(加Abstract Accessor)
  - 在Vehicle中加入getOperator (稱之為Abstract Accessor)讓Car與Plane實作，以達成維持原本Vehicle與Operator的關係



Abstract Factory Pattern

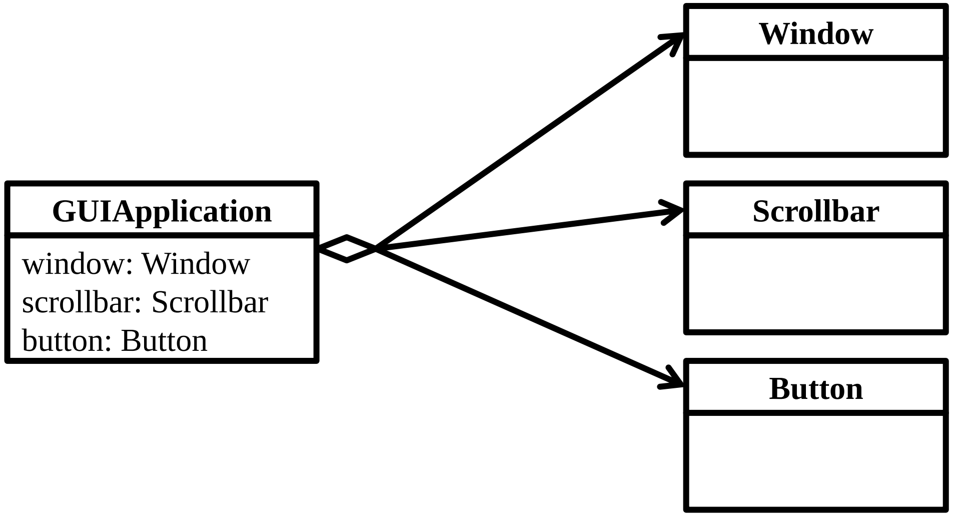
Design Aspect of Abstract Factory

Families of product objects

A GUI Application with Multiple Styles (Abstract Factory)

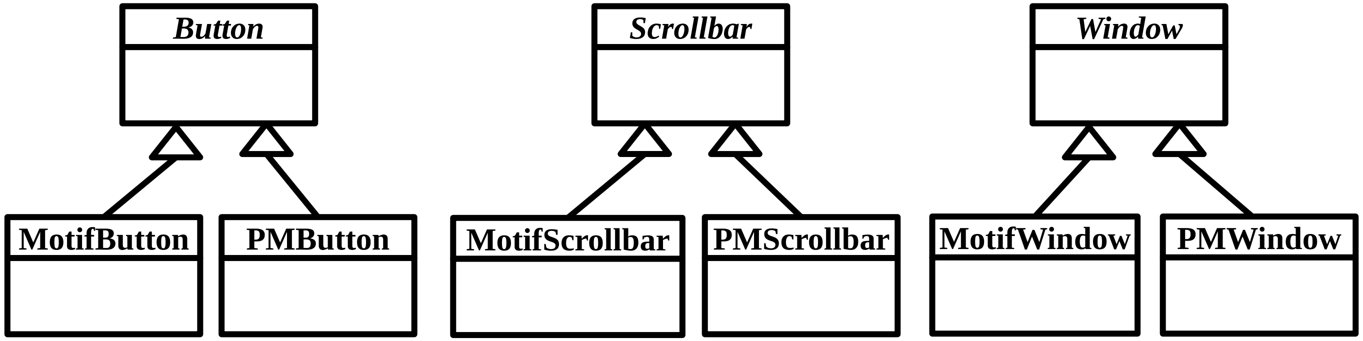
Requirements Statement$_1$

- A GUI Application consists of some kinds of widgets like window, scroll bar, and button.



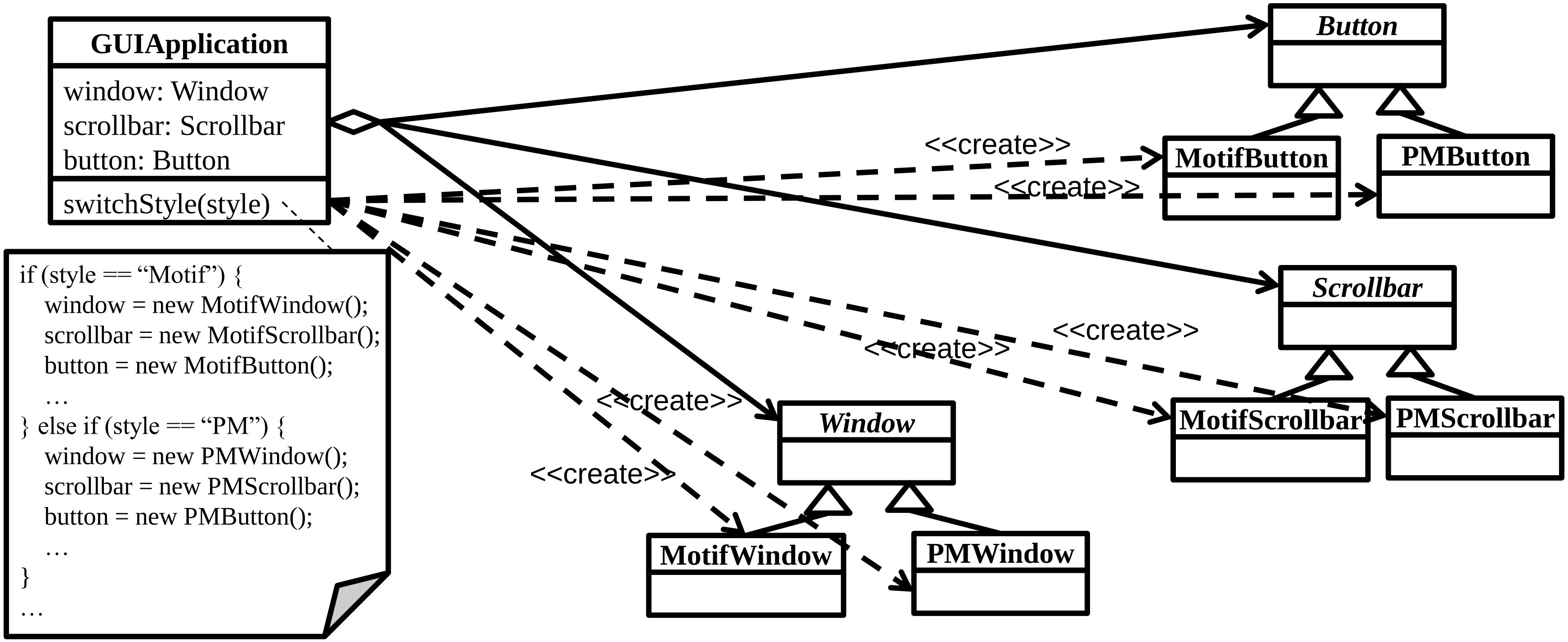
Requirements Statement$_2$

- Each widget in the GUI application has two or more implementations according to different look-and-feel standards, such as Motif and Presentation Manager.



Requirements Statement$_3$

- ❑ The GUI application can switch its look-and-feel style from one to another.



GUIApplication

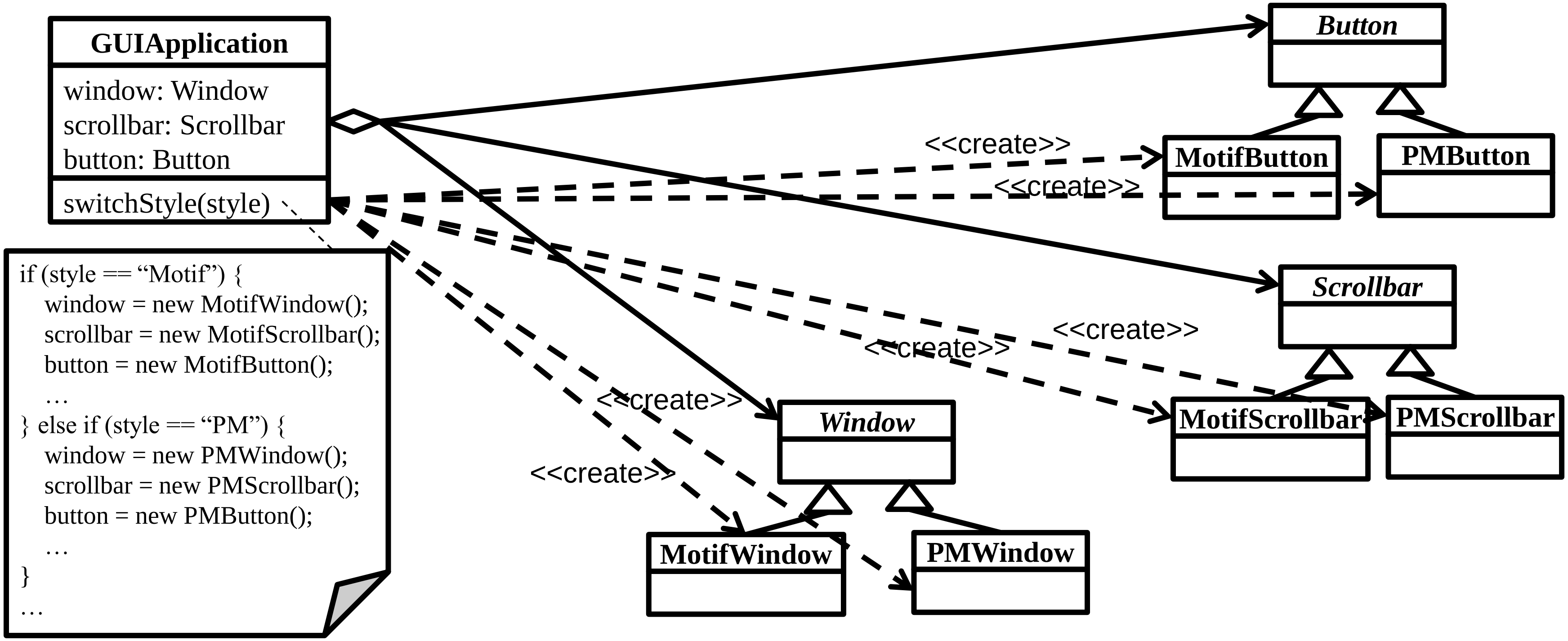
window: Window
scrollbar: Scrollbar
button: Button

switchStyle(style)

```
if (style == "Motif") {
   window = new MotifWindow();
   scrollbar = new MotifScrollbar();
   button = new MotifButton();
   …
} else if (style == "PM") {
   window = new PMWindow();
   scrollbar = new PMScrollbar();
   button = new PMButton();
   …
}
…
```

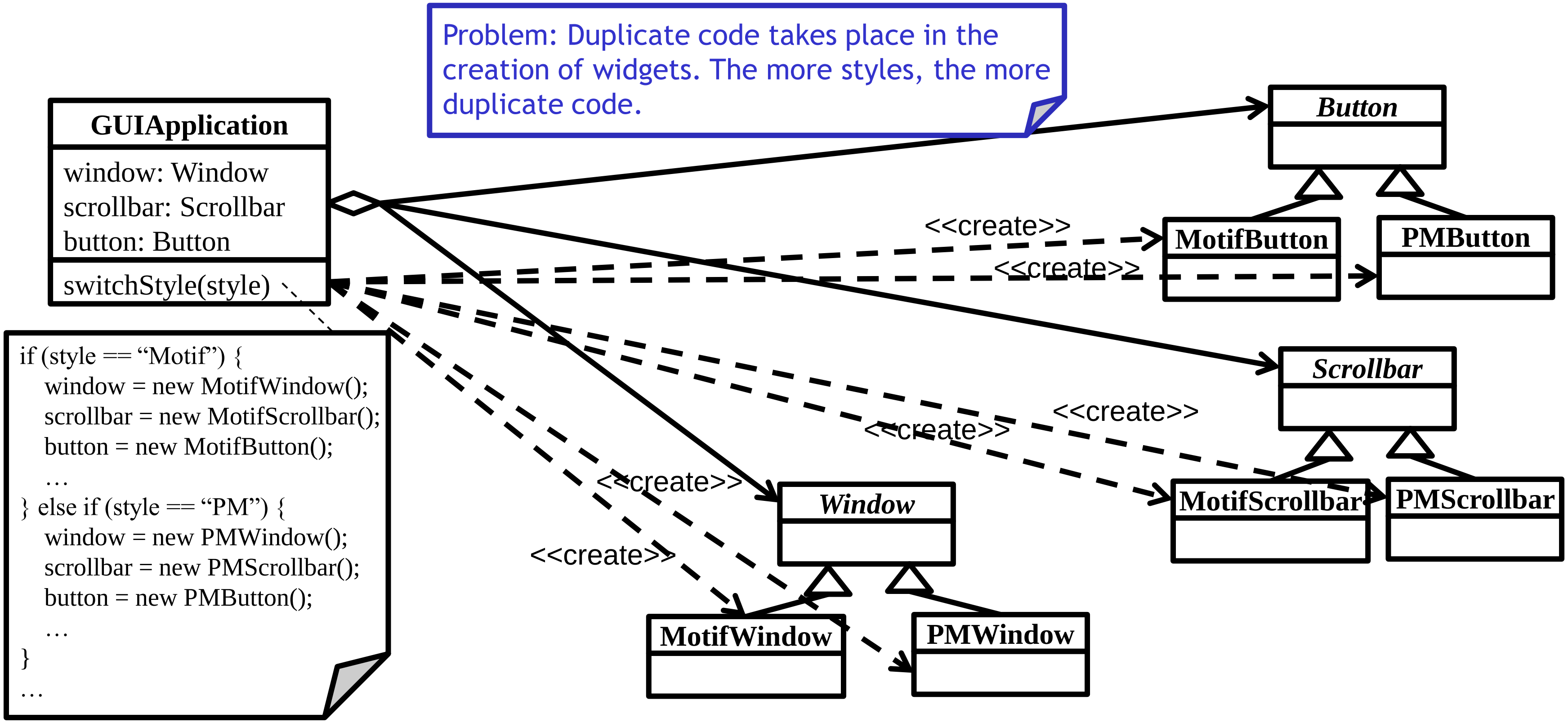
Initial Design

- ❑ The GUI application can switch its look-and-feel style from one to another.



```
if (style == "Motif") {
    window = new MotifWindow();
    scrollbar = new MotifScrollbar();
    button = new MotifButton();
    …
} else if (style == "PM") {
    window = new PMWindow();
    scrollbar = new PMScrollbar();
    button = new PMButton();
    …
}
…
```

Problems with Initial Design



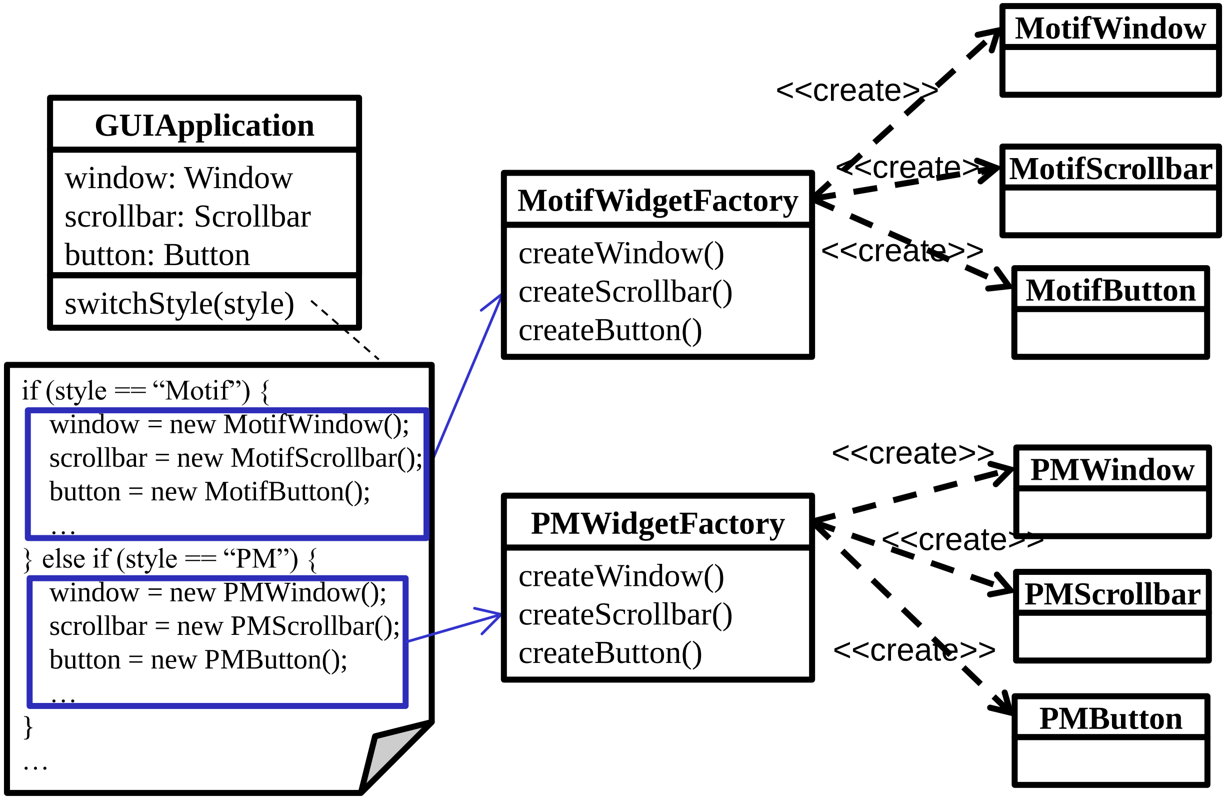
Problem: Duplicate code takes place in the creation of widgets. The more styles, the more duplicate code.

```
if (style == “Motif”) {
    window = new MotifWindow();
    scrollbar = new MotifScrollbar();
    button = new MotifButton();
    …
} else if (style == “PM”) {
    window = new PMWindow();
    scrollbar = new PMScrollbar();
    button = new PMButton();
    …
}
…
```

Refactoring by Design Principles

1. Encapsulate what varies
2. Generalize common features
3. Program to an interface, not an implementation.

Encapsulate What Varies



GUIApplication

window: Window
scrollbar: Scrollbar
button: Button

switchStyle(style)

```
if (style == “Motif”) {
    window = new MotifWindow();
    scrollbar = new MotifScrollbar();
    button = new MotifButton();
    …
} else if (style == “PM”) {
    window = new PMWindow();
    scrollbar = new PMScrollbar();
    button = new PMButton();
    …
}
…
```

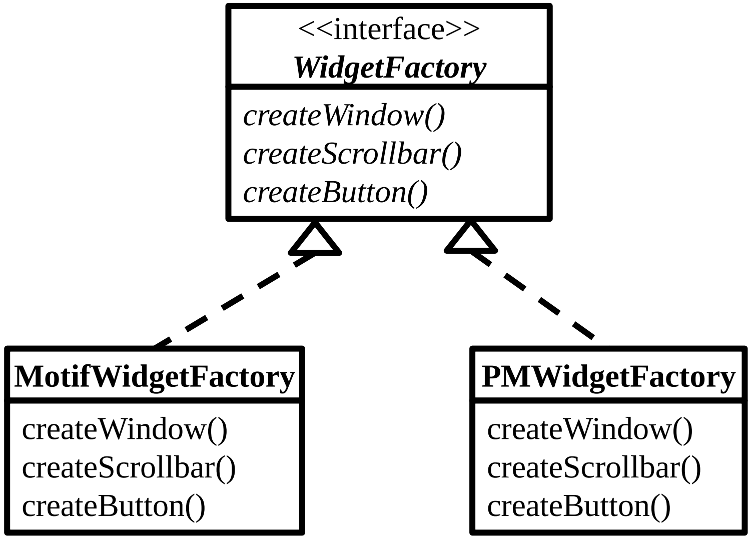
MotifWidgetFactory

createWindow()
createScrollbar()
createButton()

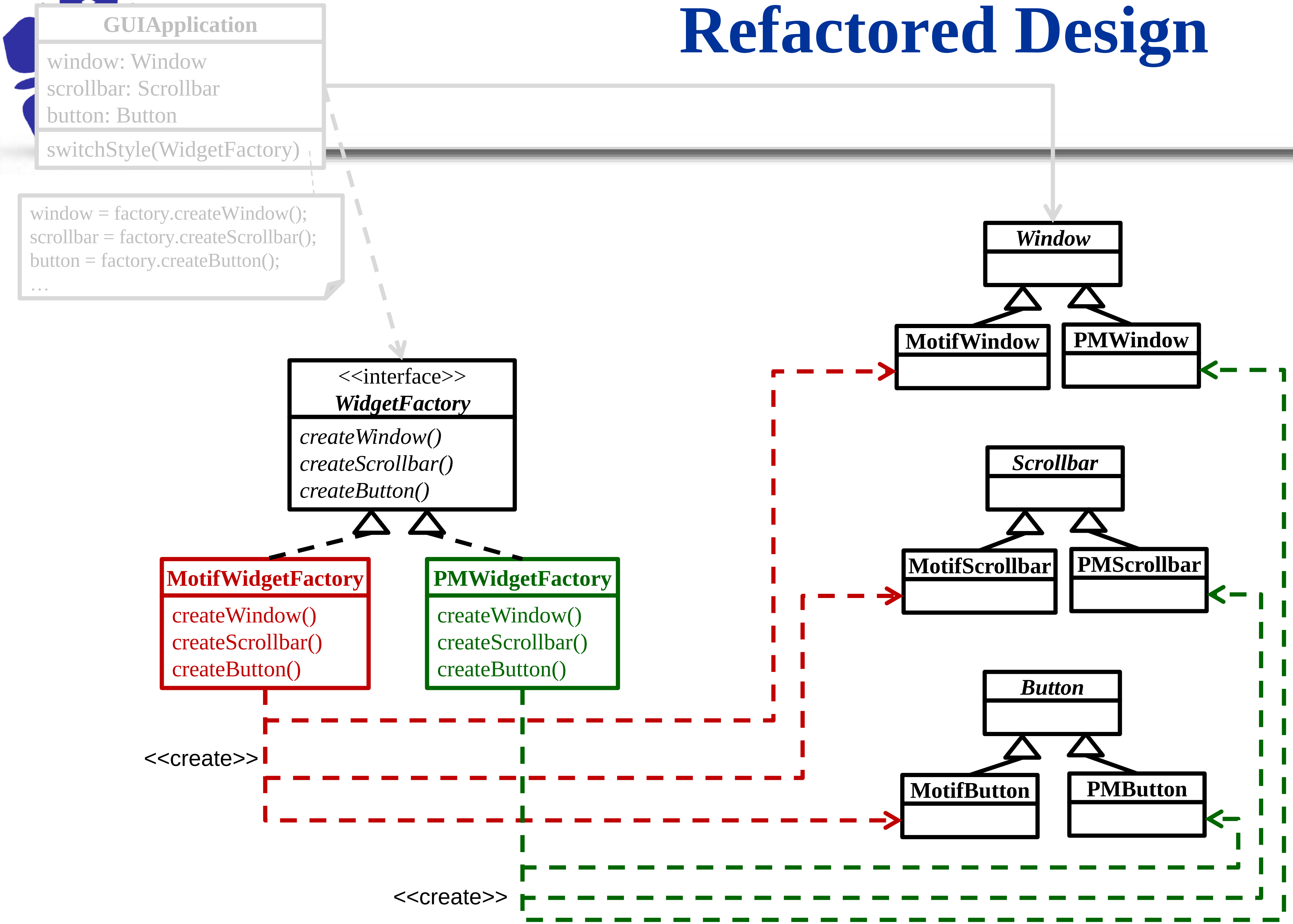
PMWidgetFactory

createWindow()
createScrollbar()
createButton()

Generalize common features



Refactored Design



GUIApplication

window: Window
scrollbar: Scrollbar
button: Button

switchStyle(WidgetFactory)

```
window = factory.createWindow();
scrollbar = factory.createScrollbar();
button = factory.createButton();
…
```

<<interface>>
WidgetFactory

createWindow()
createScrollbar()
createButton()

MotifWidgetFactory

createWindow()
createScrollbar()
createButton()

PMWidgetFactory

createWindow()
createScrollbar()
createButton()

<<create>>

<<create>>

Window

MotifWindow

PMWindow

Scrollbar

MotifScrollbar

PMScrollbar

Button

MotifButton

PMButton

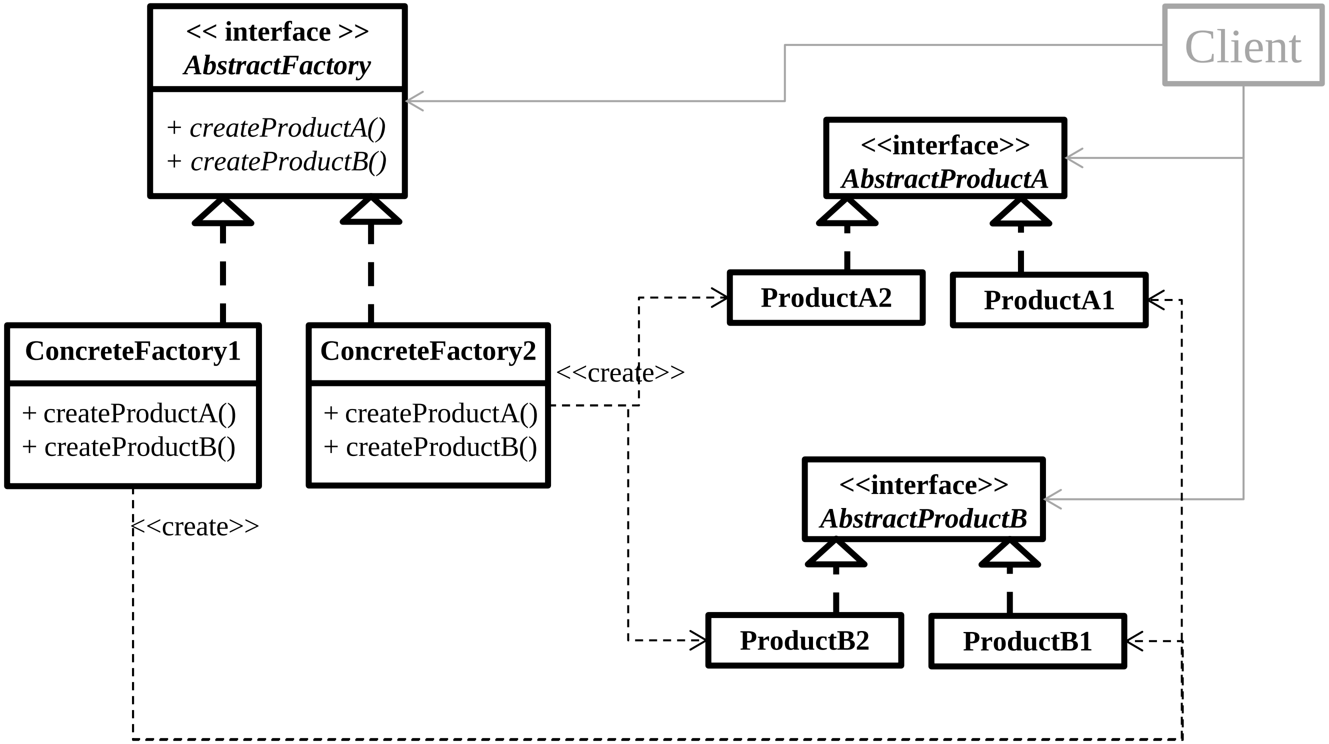
Recurrent Problem

- As the families of related or dependent objects are added, we need to write new object classes for the new families
  - For example, different look-and-feels define different appearances and behaviors for user interface "widgets" like scroll bars, windows, and buttons.

Intent

- ❑ Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Abstract Factory Structure$_1$



Abstract Factory vs. Factory Method

- ❑ Factory Method
  - ➢ creates single products

- ❑ Abstract Factory
  - ➢ consists of multiple factory methods
  - ➢ each factory method creates a related or dependent product

Template Method Pattern

Design Aspect of Template Method

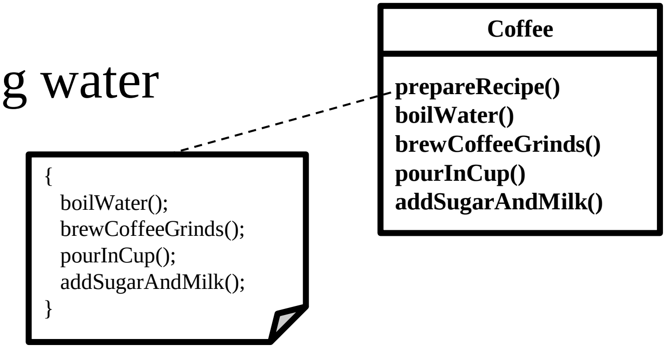
Steps of an algorithm

Prepare Caffeine Beverages

Requirements Statement$_1$

❑ Please follow these recipes precisely when preparing Starbuzz beverages

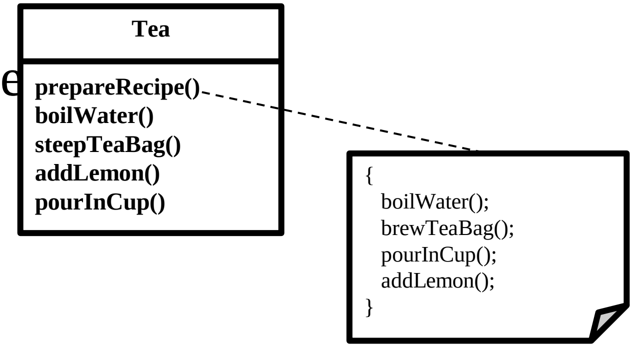
- ➢ Starbuzz Coffee Recipe
  - Boil some water
  - Brew coffee in boiling water
  - Pour Coffee in cup
  - Add sugar and milk



```
{
   boilWater();
   brewCoffeeGrinds();
   pourInCup();
   addSugarAndMilk();
}
```

Requirements Statement$_2$

- ❑ Please follow these recipes precisely when preparing Starbuzz beverages
  - ➢ Starbuzz Tea Recipe
    - Boil some water
    - Steep tea in boiling wate
    - Pour tea in cup
    - Add lemon

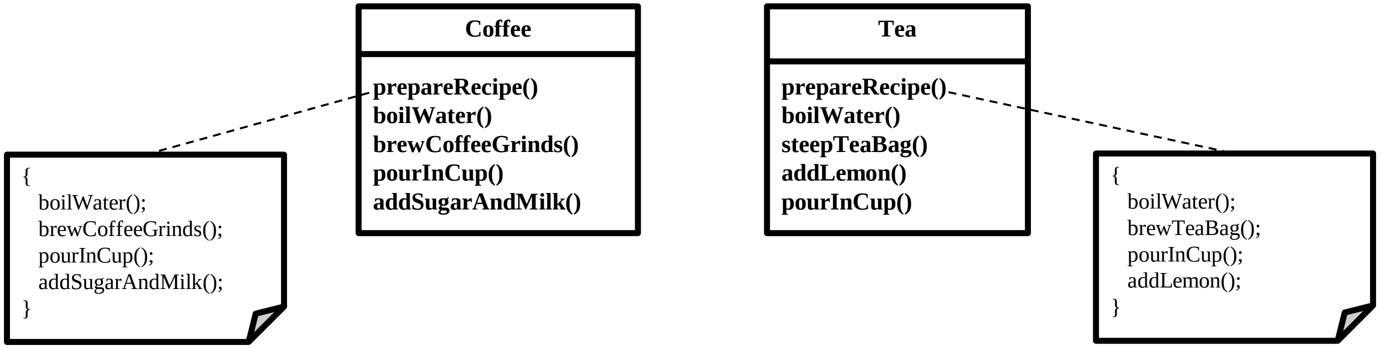


Tea

prepareRecipe()
boilWater()
steepTeaBag()
addLemon()
pourInCup()

```
{
   boilWater();
   brewTeaBag();
   pourInCup();
   addLemon();
}
```

Initial Design - Class Diagram



```
{
  boilWater();
  brewCoffeeGrinds();
  pourInCup();
  addSugarAndMilk();
}
```

Coffee

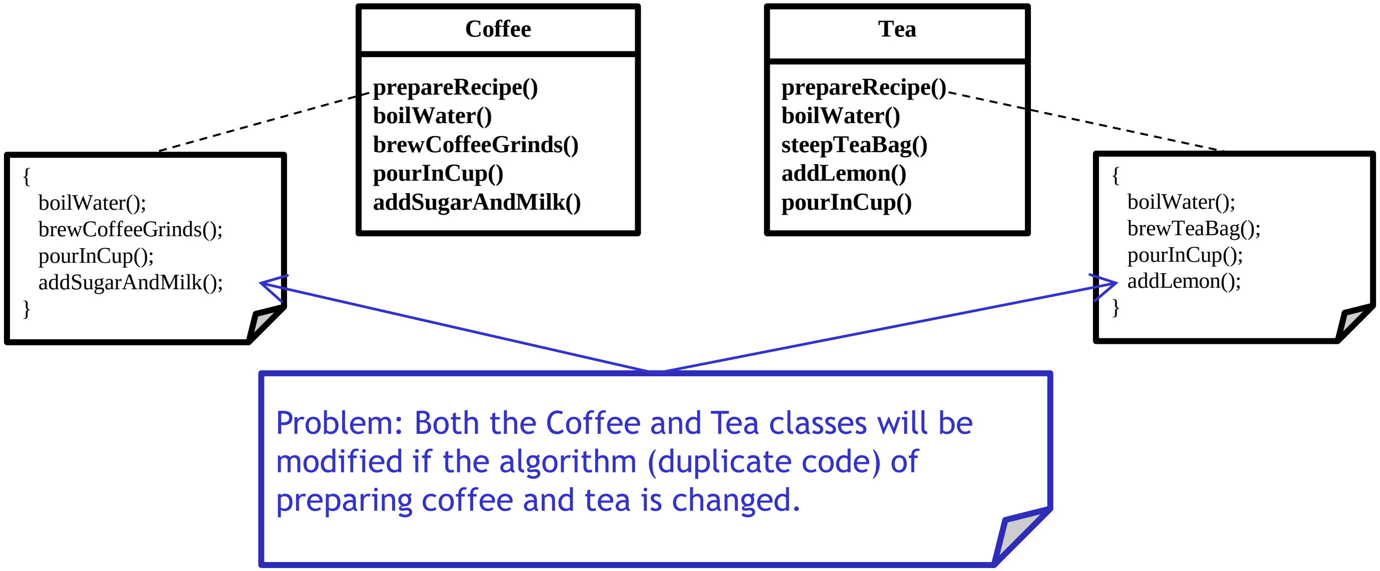
prepareRecipe()
boilWater()
brewCoffeeGrinds()
pourInCup()
addSugarAndMilk()

Tea

prepareRecipe()
boilWater()
steepTeaBag()
addLemon()
pourInCup()

```
{
  boilWater();
  brewTeaBag();
  pourInCup();
  addLemon();
}
```

Problems with Initial Design



Coffee

- **prepareRecipe()**
- **boilWater()**
- **brewCoffeeGrinds()**
- **pourInCup()**
- **addSugarAndMilk()**

```
{
  boilWater();
  brewCoffeeGrinds();
  pourInCup();
  addSugarAndMilk();
}
```

Tea

- **prepareRecipe()**
- **boilWater()**
- **steepTeaBag()**
- **addLemon()**
- **pourInCup()**

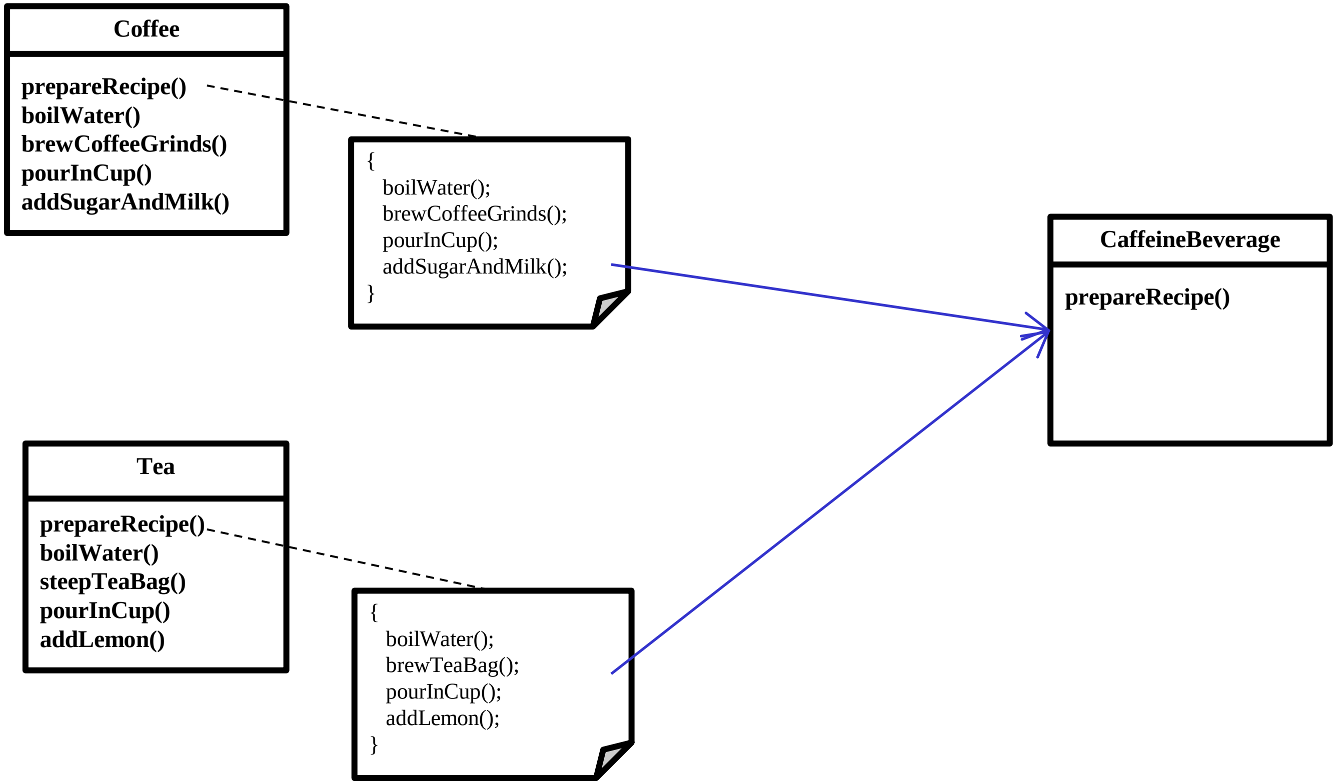
```
{
  boilWater();
  brewTeaBag();
  pourInCup();
  addLemon();
}
```

Problem: Both the Coffee and Tea classes will be modified if the algorithm (duplicate code) of preparing coffee and tea is changed.

Refactoring by Design Principles

1. Encapsulate what varies
2. Generalize common features

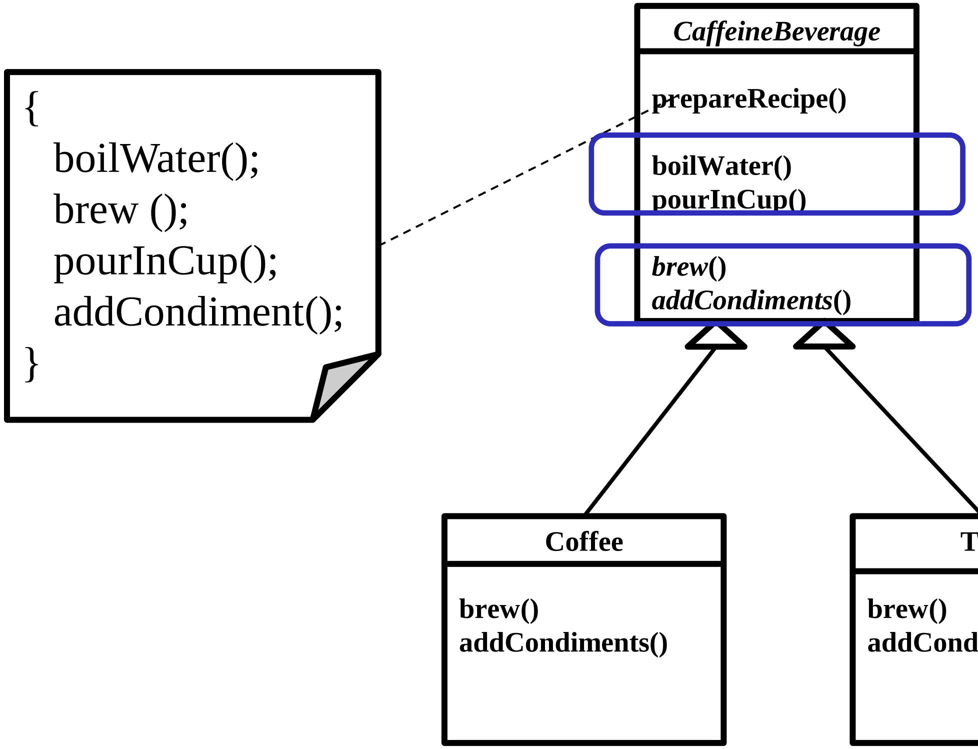
Encapsulate what varies



```
{
  boilWater();
  brewCoffeeGrinds();
  pourInCup();
  addSugarAndMilk();
}
```

```
{
  boilWater();
  brewTeaBag();
  pourInCup();
  addLemon();
}
```

Generalize common features



```
{
  boilWater();
  brew ();
  pourInCup();
  addCondiment();
}
```

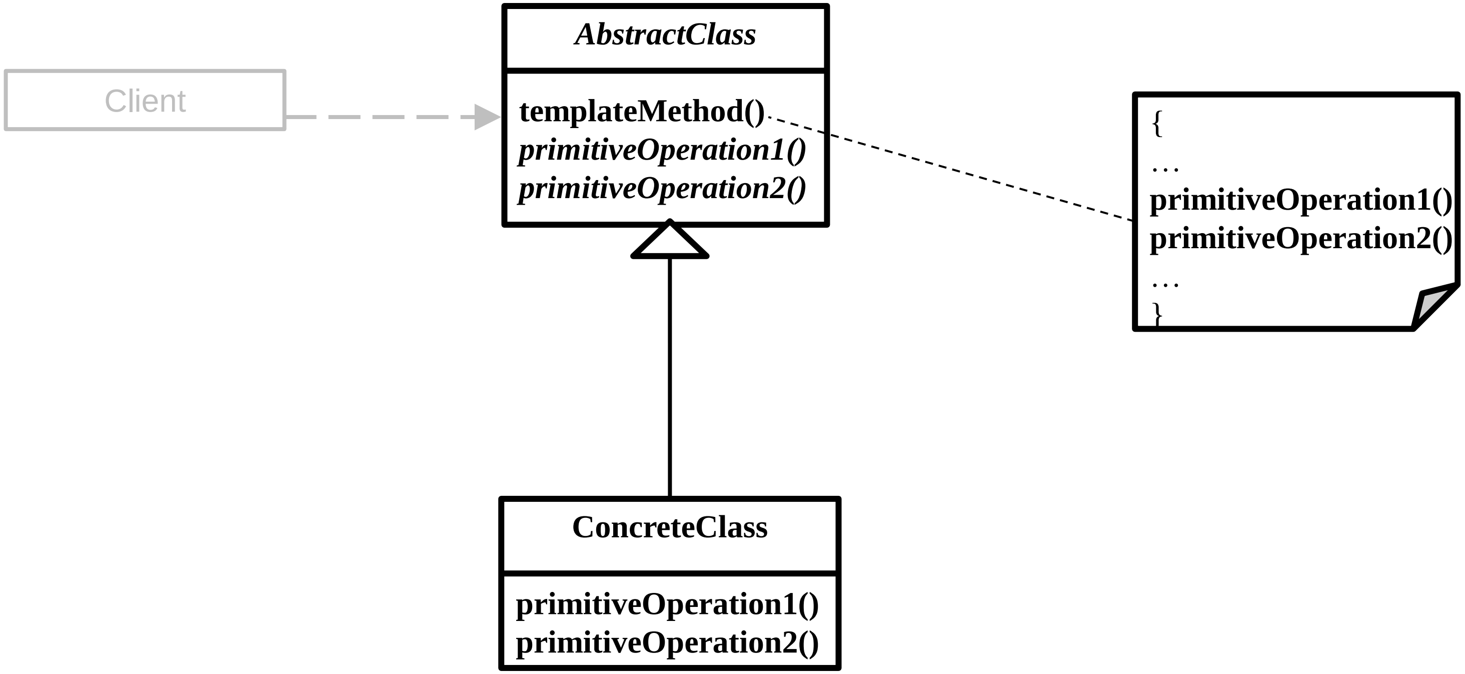
Recurrent Problem

- ❑ Two classes with code duplications would be modified at the same time if the duplicate code is being changed.

Intent

- ❑ Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.
- ❑ Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Template Pattern Structure$_1$



Adapter Pattern

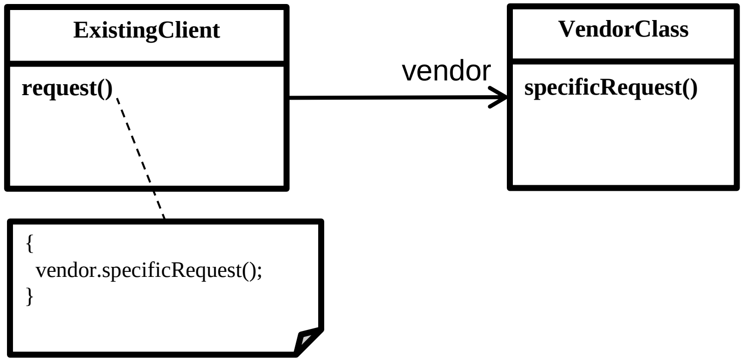
Design Aspect of Adapter

Interface to an object

New Vendor in Existing Software

Requirements Statement$_1$

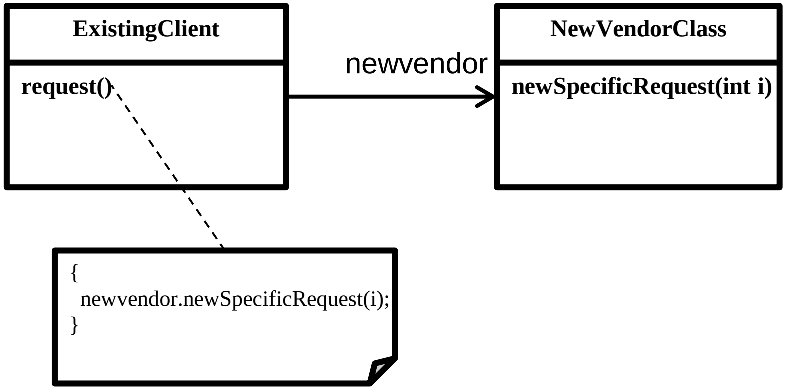
- ❑ You’ve got an existing client class that use a vendor class library.



```
{
  vendor.specificRequest();
}
```

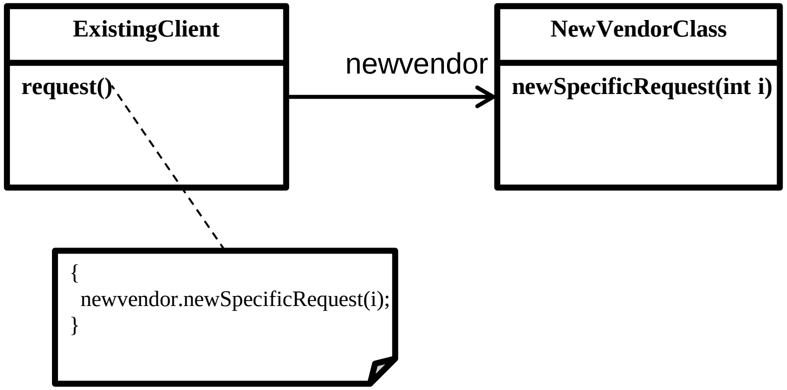
Requirements Statement$_2$

- After a while you found another vendor class library is better, but the new vendor designed their interfaces differently.



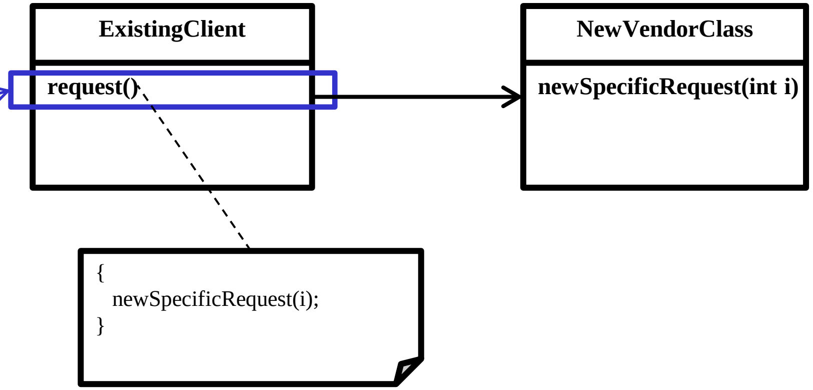
```
{
  newvendor.newSpecificRequest(i);
}
```

Initial Design



```
{
  newvendor.newSpecificRequest(i);
}
```

Problems with Initial Design



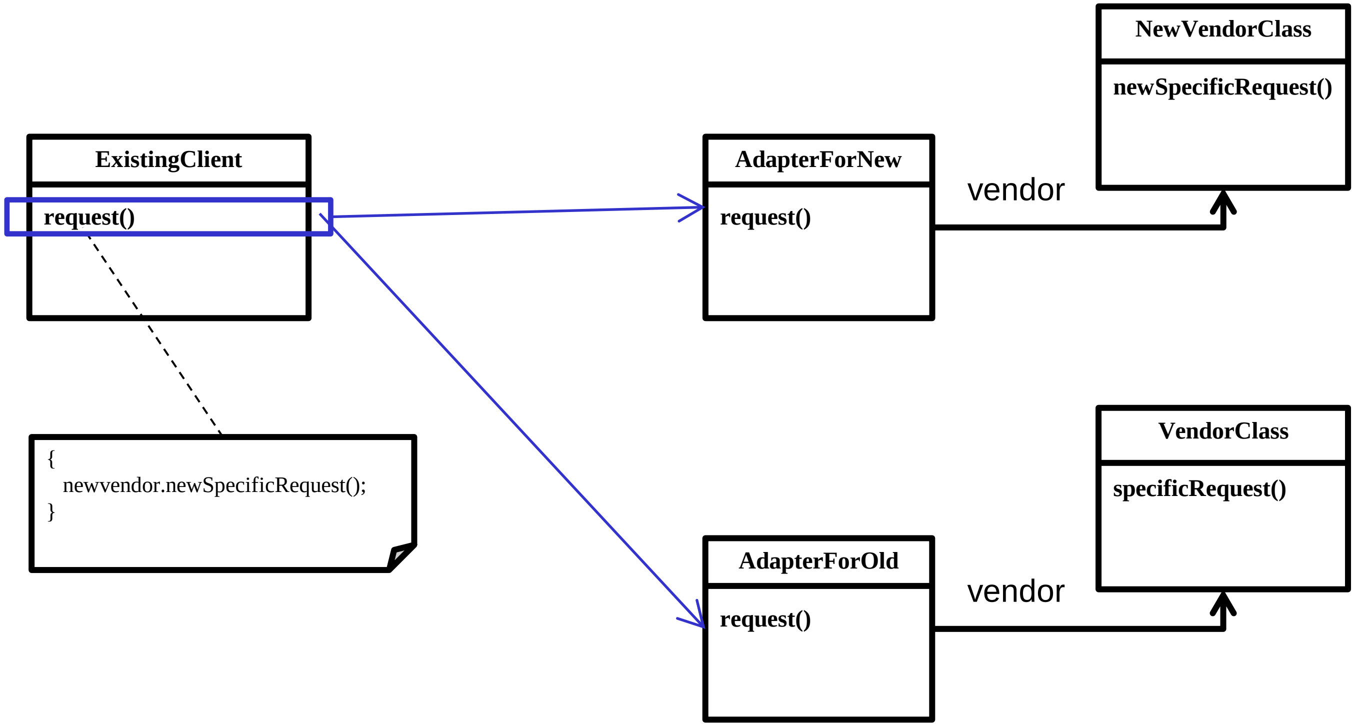
```
{
  newSpecificRequest(i);
}
```

Problem: The request method of the existing software should be modified once it changes its vendor class with a new interface.

Refactoring by Design Principles

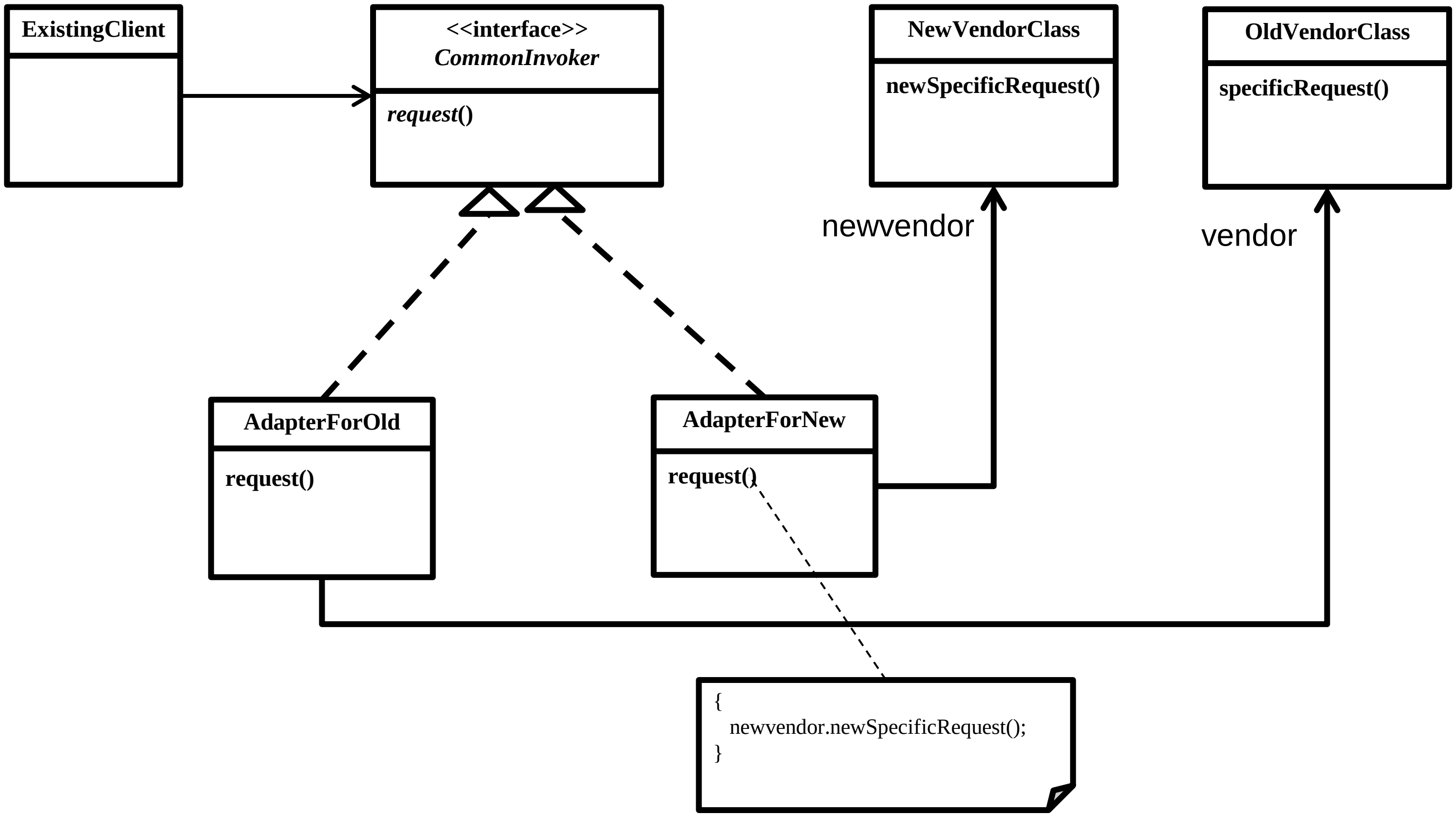
1. Encapsulate what varies
2. Generalize common features

Encapsulate What Varies



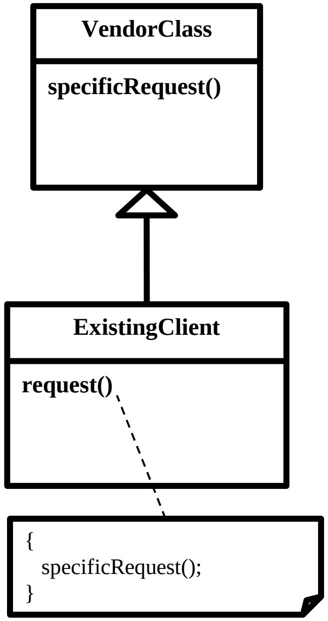
Generalize common features

Object Adapter Structure



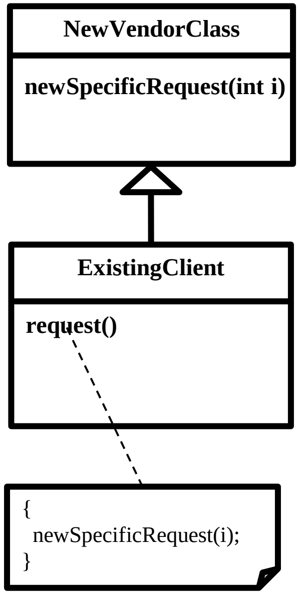
Requirements Statement$_1$

- You’ve got an existing client class that use a vendor class library by inheritance.



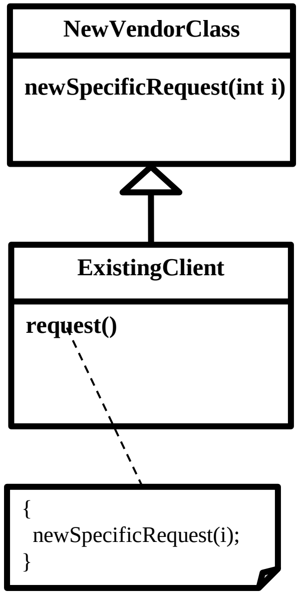
Requirements Statement$_2$

- After a while you found another vendor class library is better , but the new vendor designed their interfaces differently.

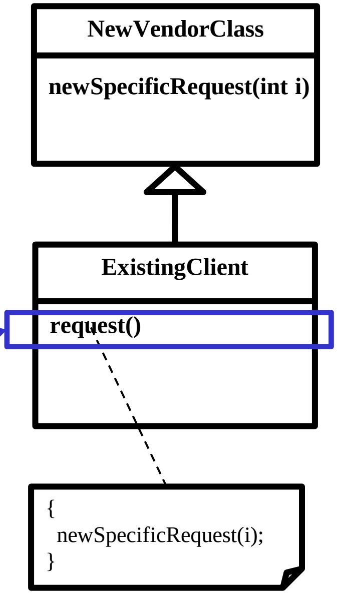


```
{
  newSpecificRequest(i);
}
```

Initial Design



Problems with Initial Design

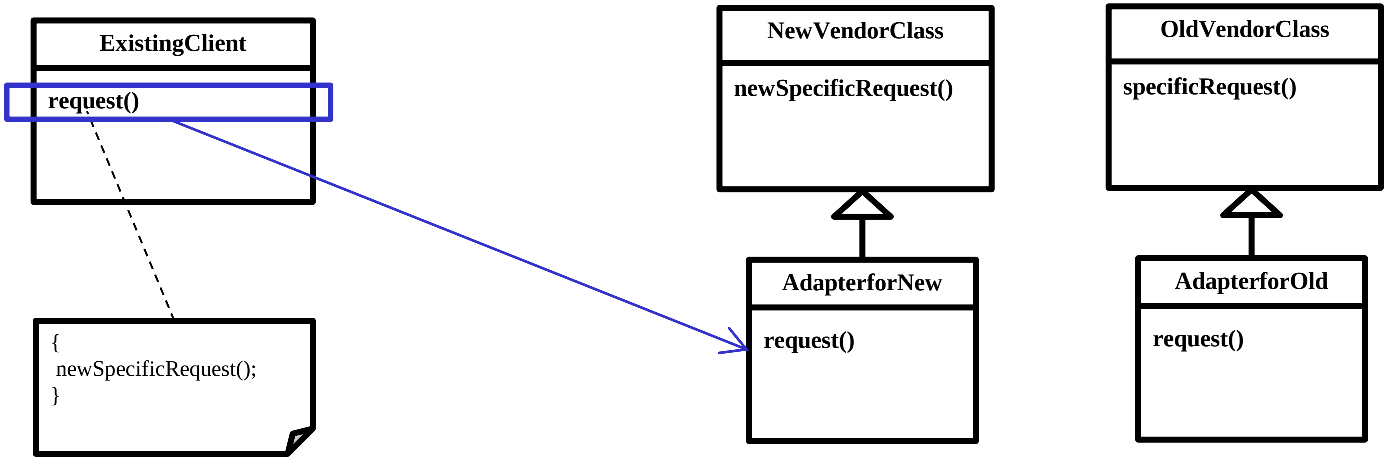


Problem: The request method of the existing software should be modified once it changes its vendor class with a new interface.

Refactoring by Design Principles

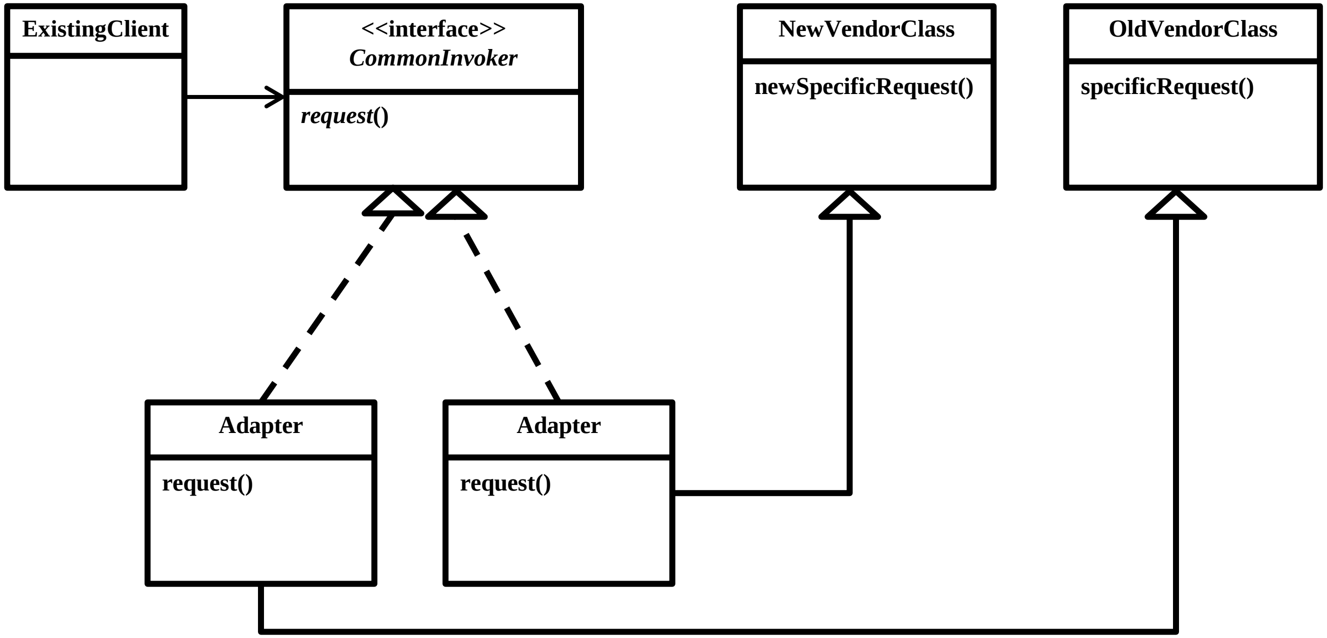
1. Encapsulate what varies
2. Generalize common features

Encapsulate what varies



Generalize common features

Class Adapter Structure



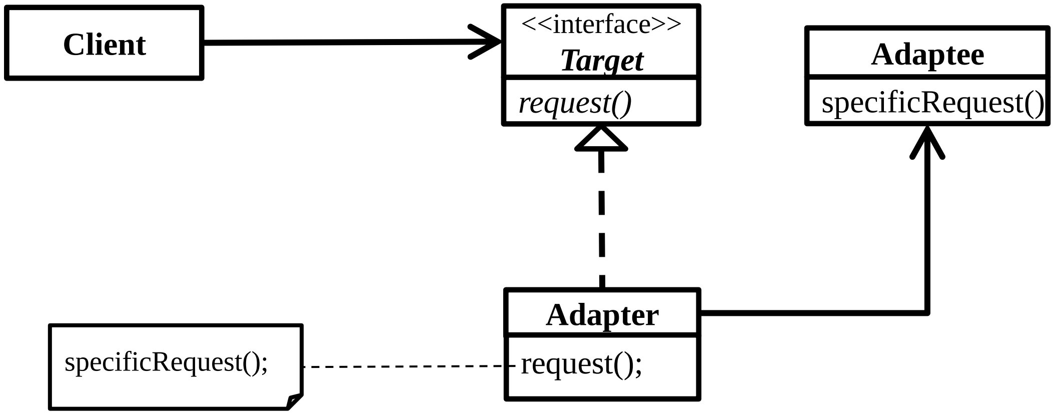
Recurrent Problem

- The request method of the requester object should be modified once it changes its receiver class with a new interface.
  - Sometimes a toolkit class that's designed for reuse isn't reusable only because its interface doesn't match the domain-specific interface an application requires.

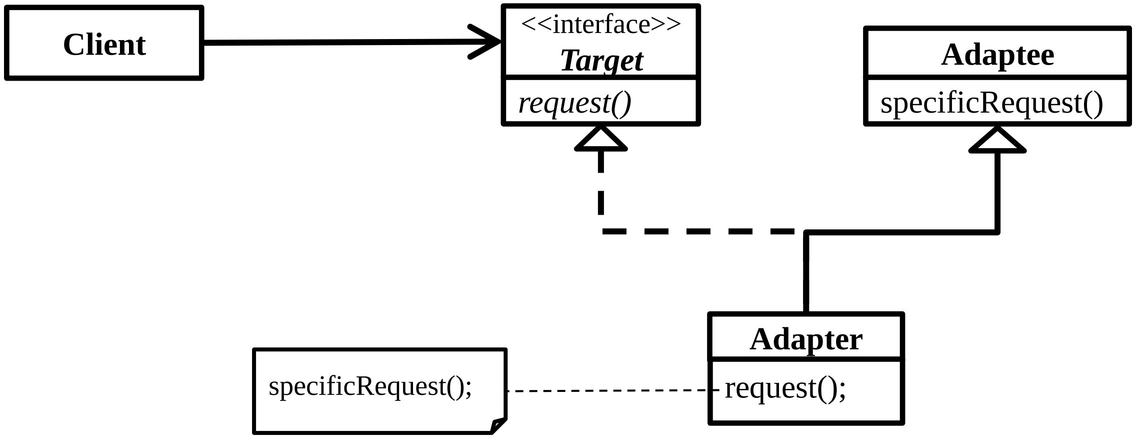
Adapter Pattern

- Intent
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Object Adapter Pattern Structure$_1$



Class Adapter Pattern Structure$_1$



Object Adapter vs. Class Adapter

	Object Adapter	Class Adapter
Design Concept	Use object composition.	Based on the concept of inheritance.
Adapt subclasses of the Adaptee	The Adaptee itself and all of its subclasses (if any).	A class adapter won't work when we want to adapt a class *and* all its subclasses.
Overriding	Can not override Adaptee methods.	It is possible to override some of Adaptee’s behaviors.

State Pattern

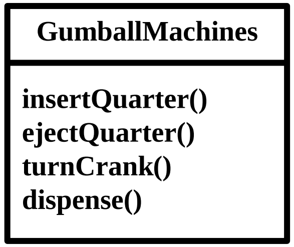
Design Aspect of State

states of an object

A Gumball Machine

Requirements Statement$_1$

- ❑ A GumballMachine has four actions: Insert Quarter, Eject Quarter, Turn Crank, and Dispense.

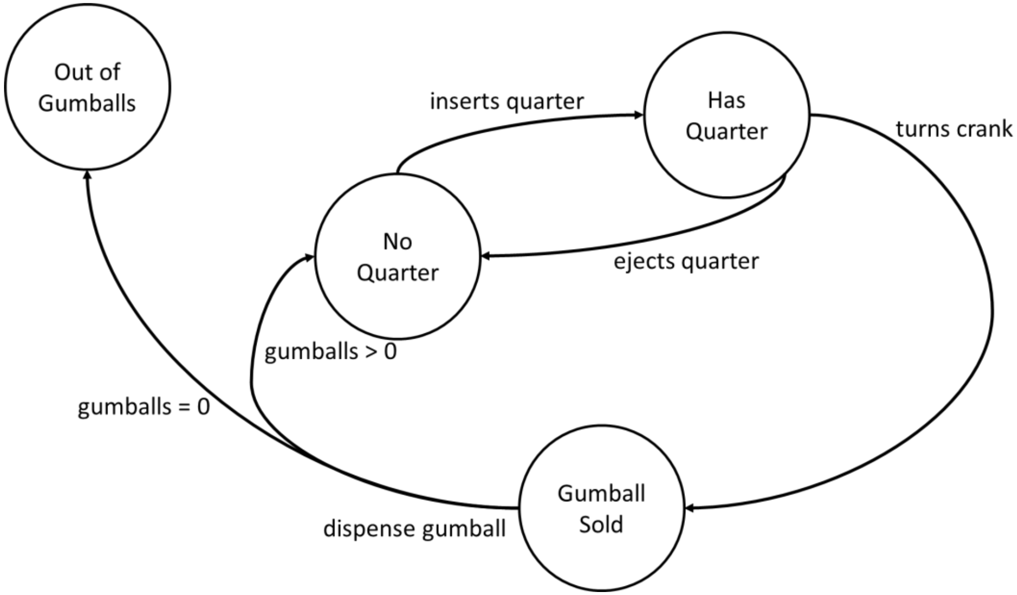


GumballMachines

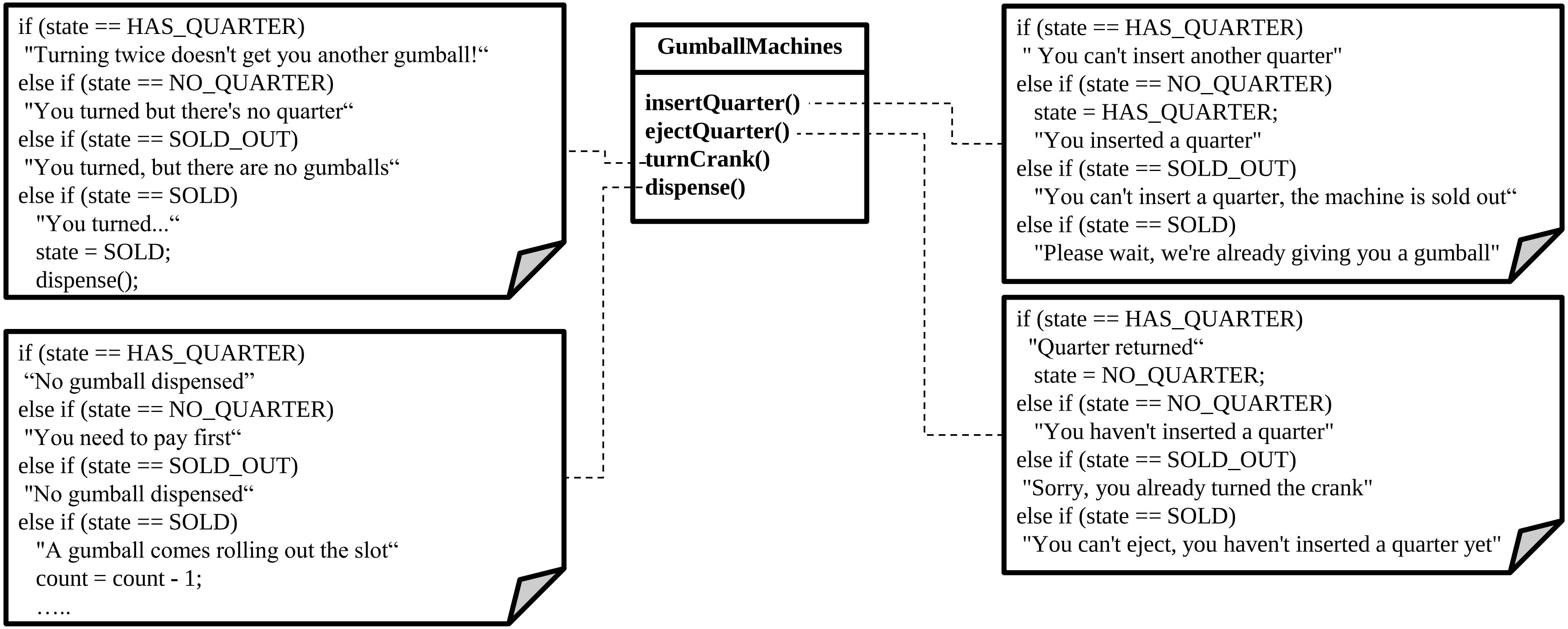
insertQuarter()
ejectQuarter()
turnCrank()
dispense()

Requirements Statement$_2$

- There are four states in the GumballMachine: No Quarter, Has Quarter, Out of Gumballs and Gumball Sold. As the following state diagram.



Requirements Statement$_3$



```
if (state == HAS_QUARTER)
 "Turning twice doesn't get you another gumball!“
else if (state == NO_QUARTER)
 "You turned but there's no quarter“
else if (state == SOLD_OUT)
 "You turned, but there are no gumballs“
else if (state == SOLD)
   "You turned...“
   state = SOLD;
   dispense();
```

```
if (state == HAS_QUARTER)
 “No gumball dispensed”
else if (state == NO_QUARTER)
 "You need to pay first“
else if (state == SOLD_OUT)
 "No gumball dispensed“
else if (state == SOLD)
   "A gumball comes rolling out the slot“
   count = count - 1;
   …..
```

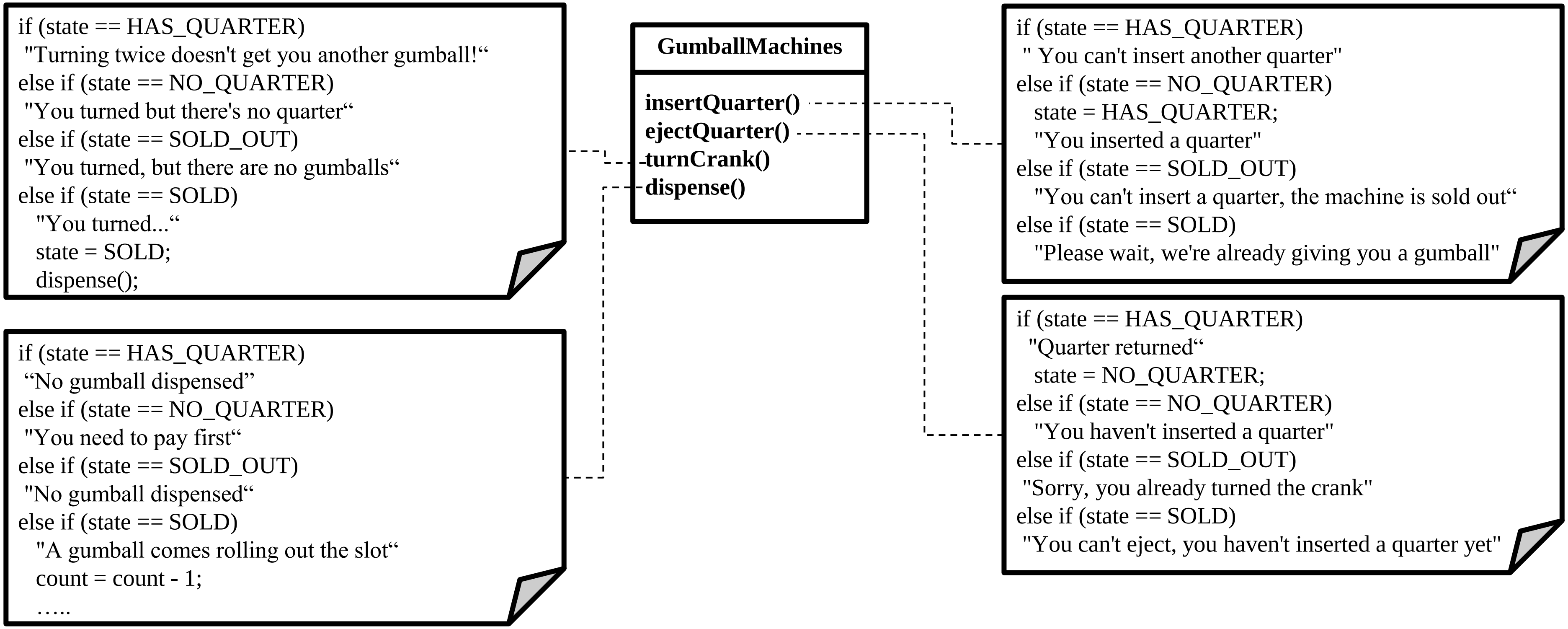
GumballMachines

insertQuarter()
ejectQuarter()
turnCrank()
dispense()

```
if (state == HAS_QUARTER)
 " You can't insert another quarter"
else if (state == NO_QUARTER)
   state = HAS_QUARTER;
   "You inserted a quarter"
else if (state == SOLD_OUT)
   "You can't insert a quarter, the machine is sold out“
else if (state == SOLD)
   "Please wait, we're already giving you a gumball"
```

```
if (state == HAS_QUARTER)
  "Quarter returned“
   state = NO_QUARTER;
else if (state == NO_QUARTER)
   "You haven't inserted a quarter"
else if (state == SOLD_OUT)
 "Sorry, you already turned the crank"
else if (state == SOLD)
 "You can't eject, you haven't inserted a quarter yet"
```

Initial Design - Class Diagram



GumballMachines

insertQuarter()
ejectQuarter()
turnCrank()
dispense()

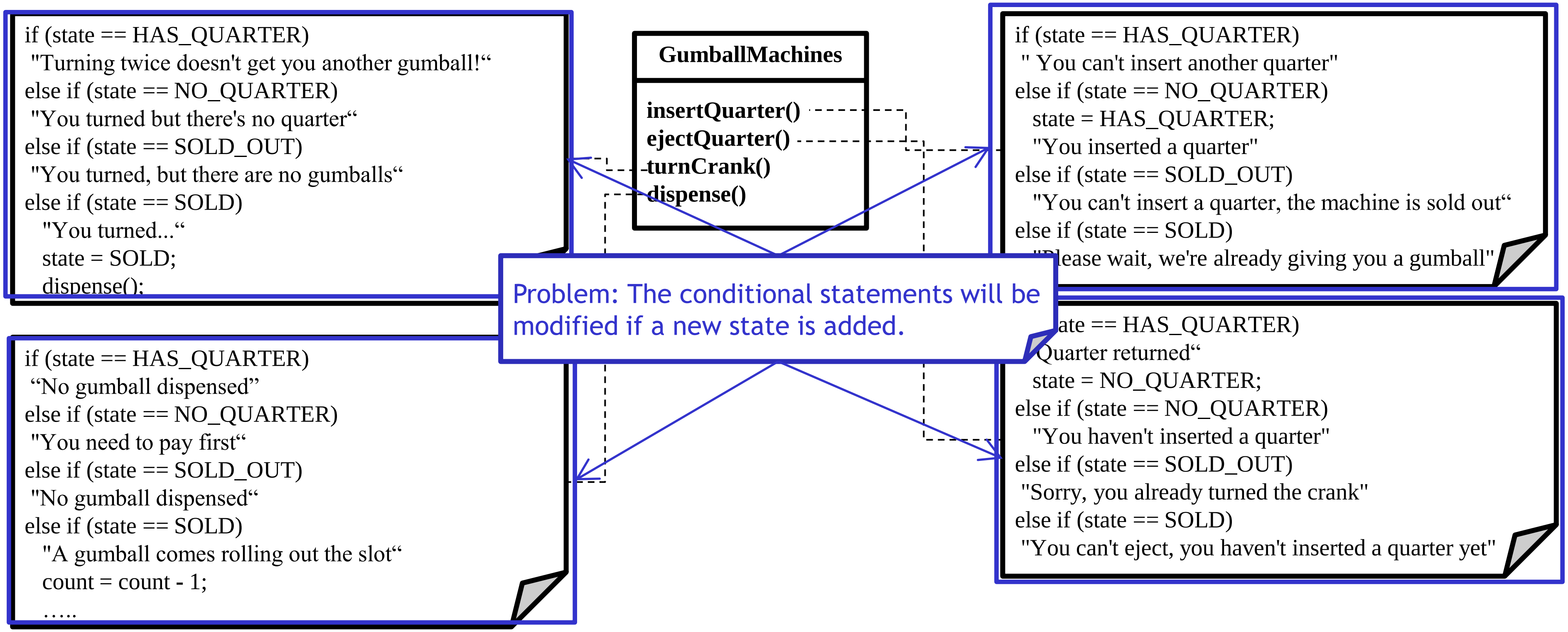
```
if (state == HAS_QUARTER)
 "Turning twice doesn't get you another gumball!“
else if (state == NO_QUARTER)
 "You turned but there's no quarter“
else if (state == SOLD_OUT)
 "You turned, but there are no gumballs“
else if (state == SOLD)
   "You turned...“
   state = SOLD;
   dispense();
```

```
if (state == HAS_QUARTER)
 “No gumball dispensed”
else if (state == NO_QUARTER)
 "You need to pay first“
else if (state == SOLD_OUT)
 "No gumball dispensed“
else if (state == SOLD)
   "A gumball comes rolling out the slot“
   count = count - 1;
   …..
```

```
if (state == HAS_QUARTER)
 " You can't insert another quarter"
else if (state == NO_QUARTER)
   state = HAS_QUARTER;
   "You inserted a quarter"
else if (state == SOLD_OUT)
   "You can't insert a quarter, the machine is sold out“
else if (state == SOLD)
   "Please wait, we're already giving you a gumball"
```

```
if (state == HAS_QUARTER)
  "Quarter returned“
   state = NO_QUARTER;
else if (state == NO_QUARTER)
   "You haven't inserted a quarter"
else if (state == SOLD_OUT)
 "Sorry, you already turned the crank"
else if (state == SOLD)
 "You can't eject, you haven't inserted a quarter yet"
```

Problems with Initial Design



GumballMachines

insertQuarter()
ejectQuarter()
turnCrank()
dispense()

```
if (state == HAS_QUARTER)
 "Turning twice doesn't get you another gumball!“
else if (state == NO_QUARTER)
 "You turned but there's no quarter“
else if (state == SOLD_OUT)
 "You turned, but there are no gumballs“
else if (state == SOLD)
  "You turned...“
  state = SOLD;
  dispense();
```

```
if (state == HAS_QUARTER)
 " You can't insert another quarter"
else if (state == NO_QUARTER)
  state = HAS_QUARTER;
  "You inserted a quarter"
else if (state == SOLD_OUT)
  "You can't insert a quarter, the machine is sold out“
else if (state == SOLD)
  "Please wait, we're already giving you a gumball"
```

Problem: The conditional statements will be modified if a new state is added.

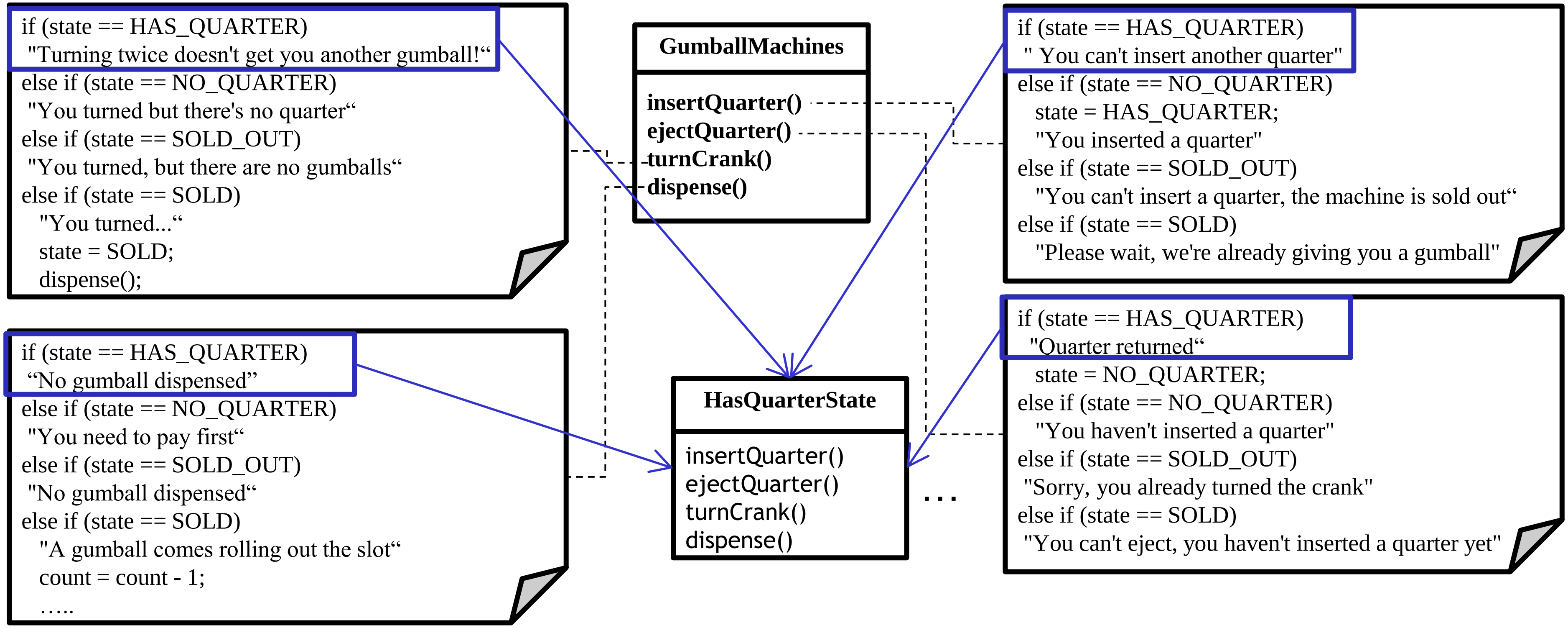
```
if (state == HAS_QUARTER)
 “No gumball dispensed”
else if (state == NO_QUARTER)
 "You need to pay first“
else if (state == SOLD_OUT)
 "No gumball dispensed“
else if (state == SOLD)
  "A gumball comes rolling out the slot“
  count = count - 1;
  …..
```

```
      ate == HAS_QUARTER)
 "Quarter returned“
  state = NO_QUARTER;
else if (state == NO_QUARTER)
  "You haven't inserted a quarter"
else if (state == SOLD_OUT)
 "Sorry, you already turned the crank"
else if (state == SOLD)
 "You can't eject, you haven't inserted a quarter yet"
```

Refactoring by Design Principles

1. Encapsulate what varies
2. Generalize common features
3. Program to an interface, not an implementation

Encapsulate what varies



```
if (state == HAS_QUARTER)
 "Turning twice doesn't get you another gumball!“
else if (state == NO_QUARTER)
 "You turned but there's no quarter“
else if (state == SOLD_OUT)
 "You turned, but there are no gumballs“
else if (state == SOLD)
   "You turned...“
   state = SOLD;
   dispense();
```

GumballMachines

insertQuarter()
ejectQuarter()
turnCrank()
dispense()

```
if (state == HAS_QUARTER)
 " You can't insert another quarter"
else if (state == NO_QUARTER)
   state = HAS_QUARTER;
   "You inserted a quarter"
else if (state == SOLD_OUT)
   "You can't insert a quarter, the machine is sold out“
else if (state == SOLD)
   "Please wait, we're already giving you a gumball"
```

```
if (state == HAS_QUARTER)
 “No gumball dispensed”
else if (state == NO_QUARTER)
 "You need to pay first“
else if (state == SOLD_OUT)
 "No gumball dispensed“
else if (state == SOLD)
   "A gumball comes rolling out the slot“
   count = count - 1;
   …..
```

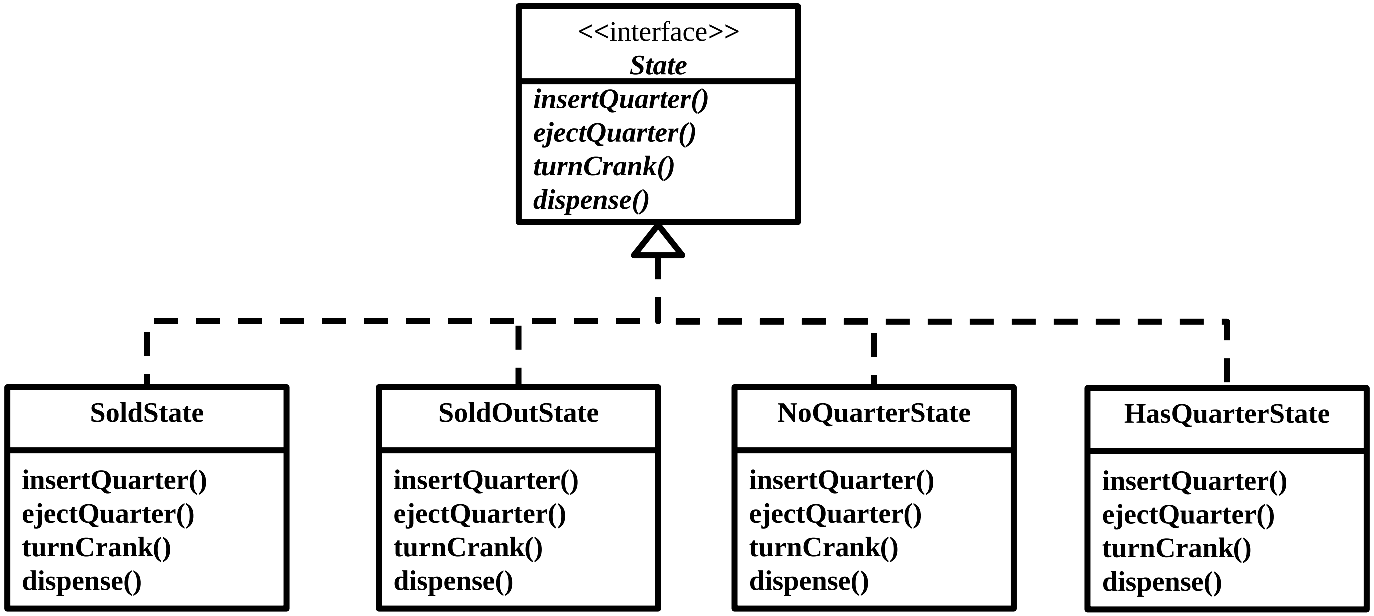
HasQuarterState

insertQuarter()
ejectQuarter()
turnCrank()
dispense()

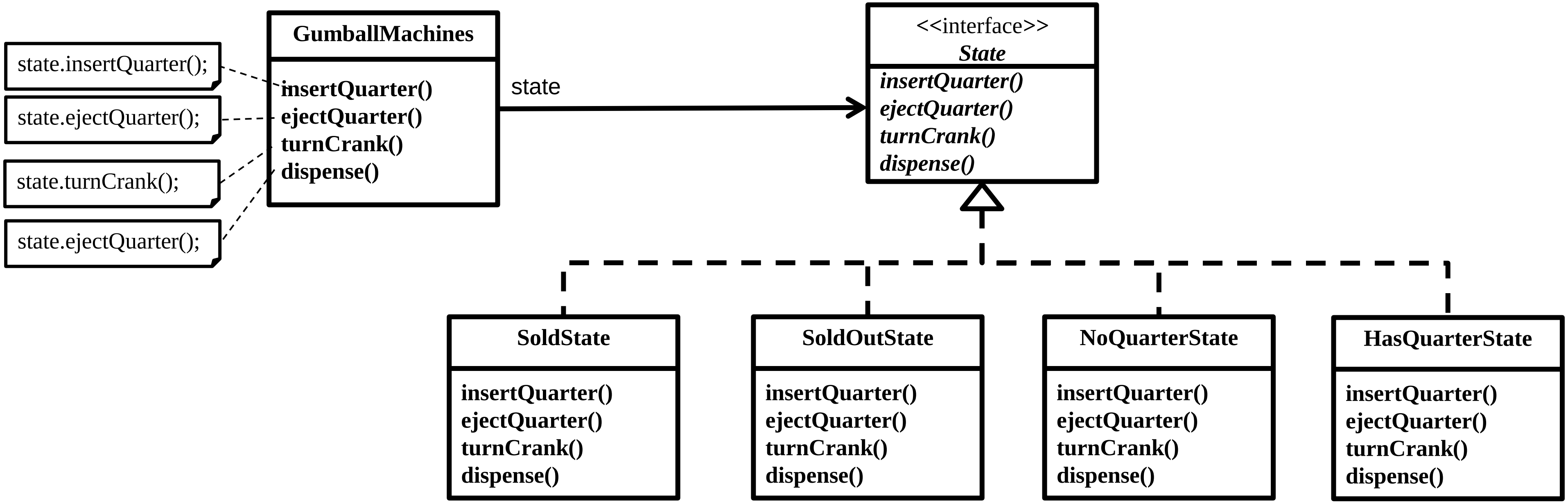
…

```
if (state == HAS_QUARTER)
  "Quarter returned“
   state = NO_QUARTER;
else if (state == NO_QUARTER)
   "You haven't inserted a quarter"
else if (state == SOLD_OUT)
 "Sorry, you already turned the crank"
else if (state == SOLD)
 "You can't eject, you haven't inserted a quarter yet"
```

Generalize common features



Program to an interface



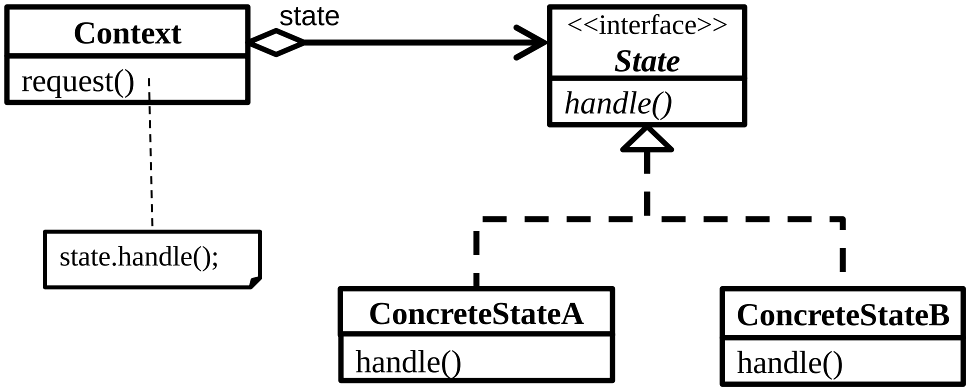
Recurrent Problem

- An object’s behavior depends on its state, and it must change its behavior at run-time depending on that state.

State Pattern

- ❑ Intent
  - ➢ Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

State Pattern Structure$_1$



Visitor Pattern

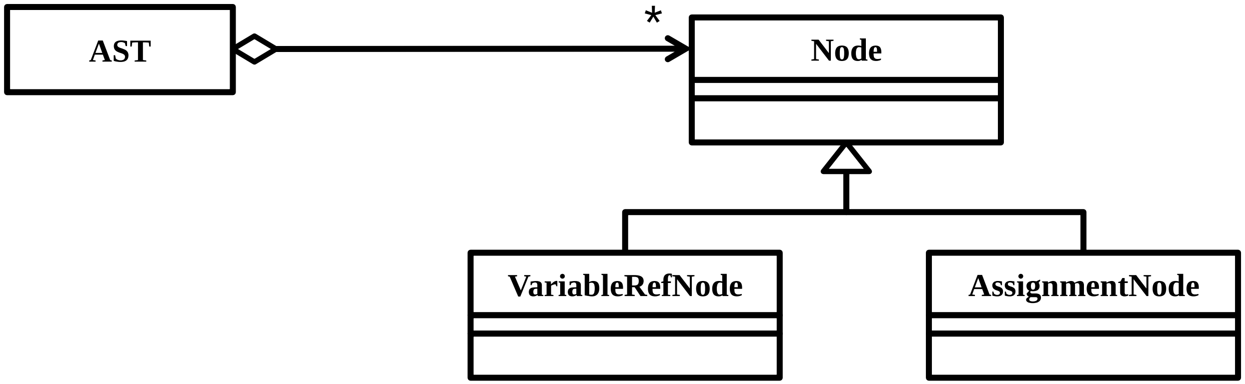
Design Aspect of Visitor

Operations that can be applied to objects without changing their classes

Compiler and AST

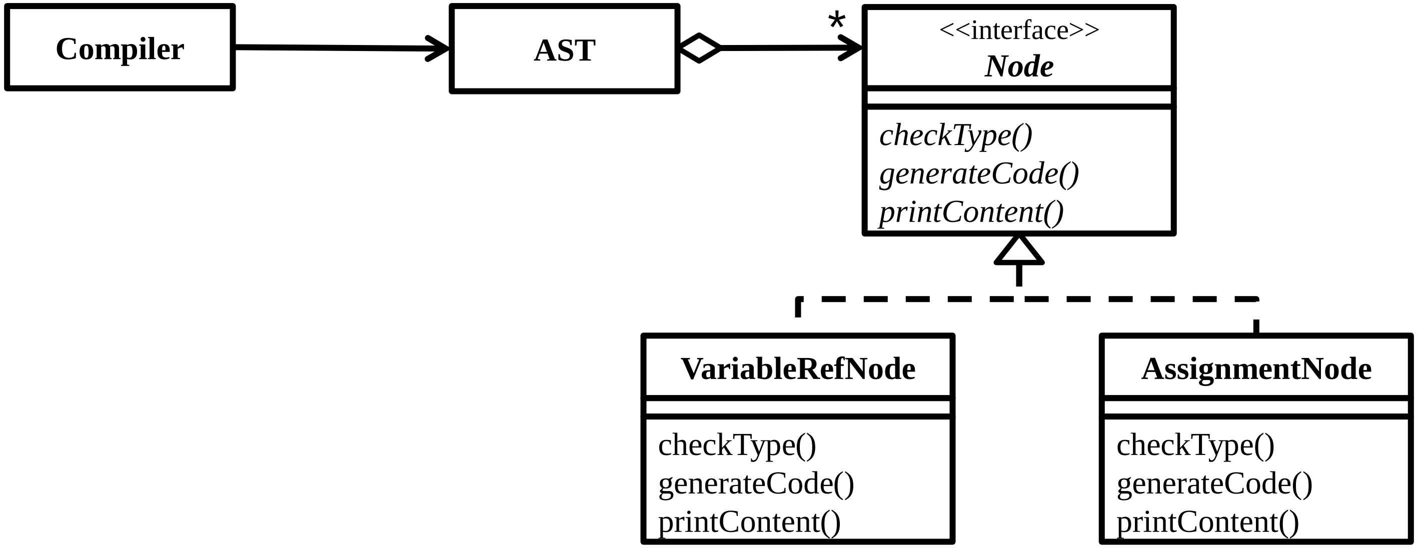
Requirements Statements$_1$

- There are several nodes in an abstract syntax tree (AST), such as VariableRefNode and AssignmentNode, which represent respective parts in source code and keep the code information.

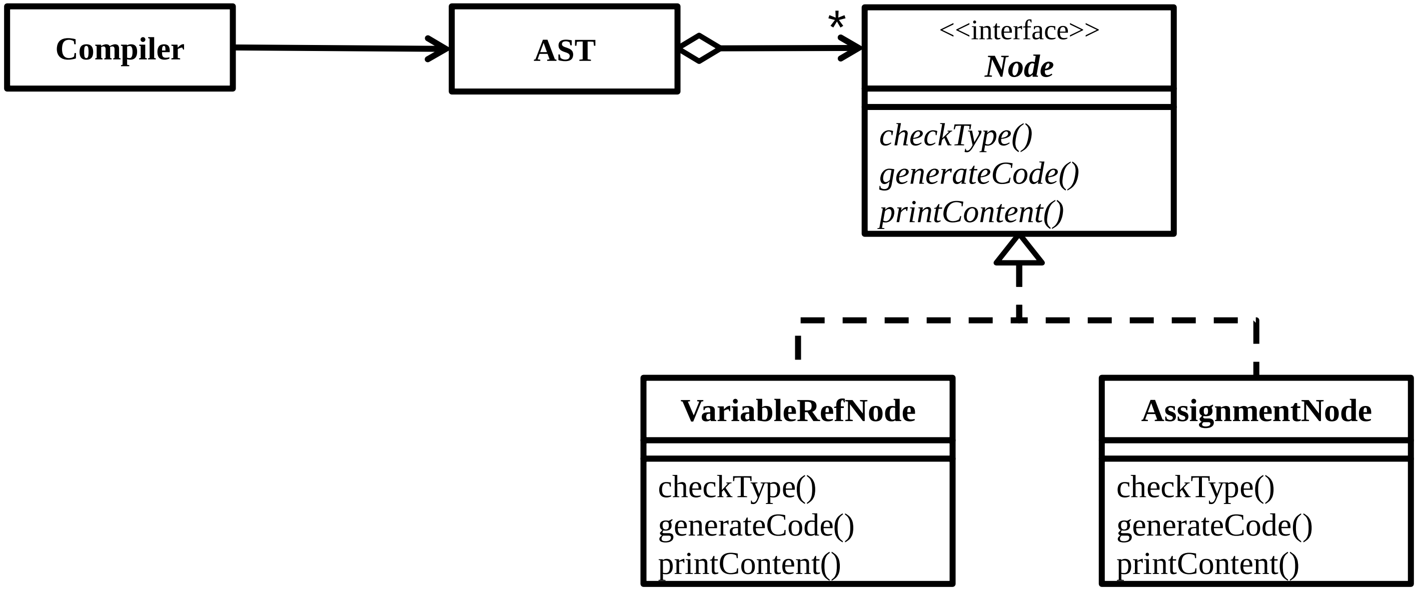


Requirements Statements$_2$

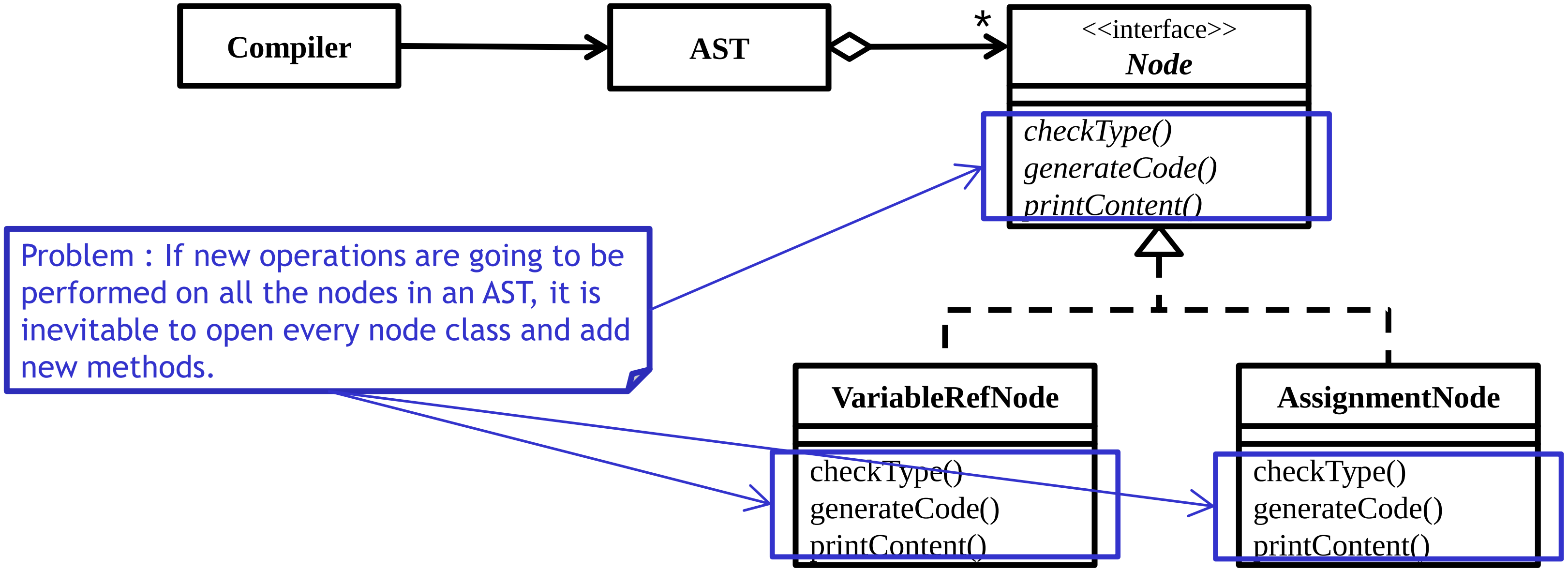
- Each node currently provides three interfaces for the compiler to use in order to check its type, generate code and print out the content.



Initial Design



Problem with the Initial Design

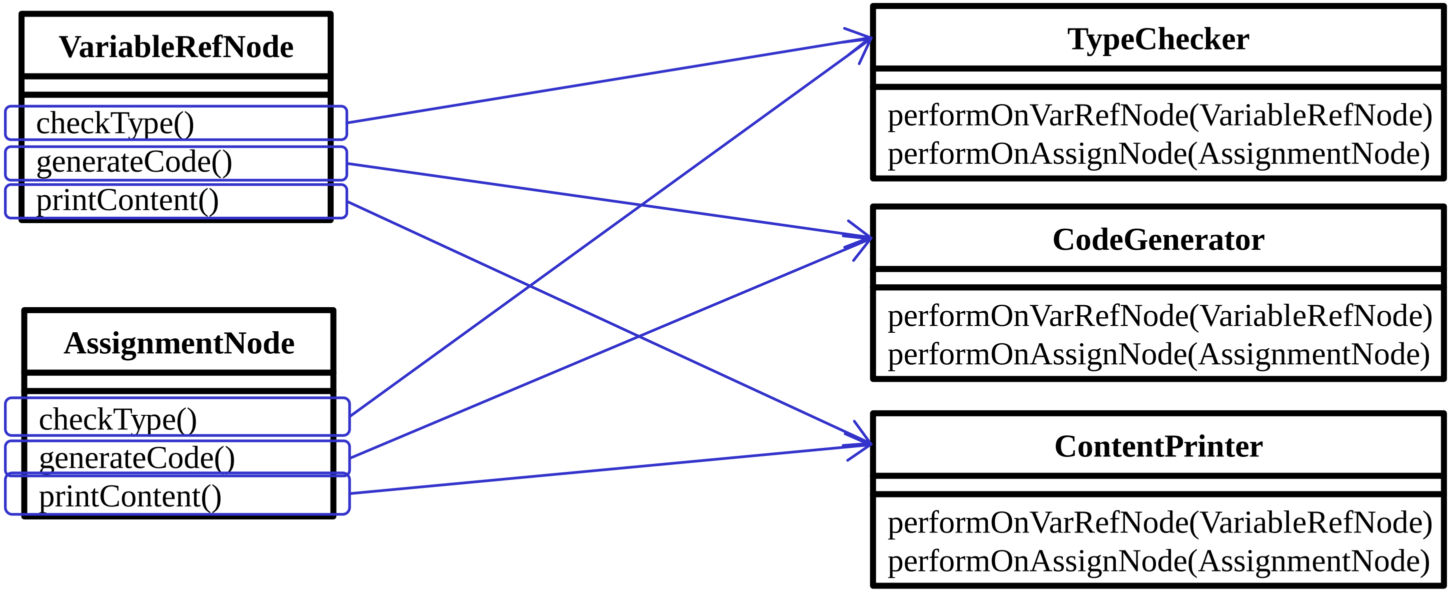


Problem : If new operations are going to be performed on all the nodes in an AST, it is inevitable to open every node class and add new methods.

Refactoring by Design Principles

1. Encapsulate what varies
2. Generalize common features
3. Program to an interface, not an implementation

Encapsulate what varies


VariableRefNode

checkType()
generateCode()
printContent()

AssignmentNode

checkType()
generateCode()
printContent()

TypeChecker

performOnVarRefNode(VariableRefNode)
performOnAssignNode(AssignmentNode)

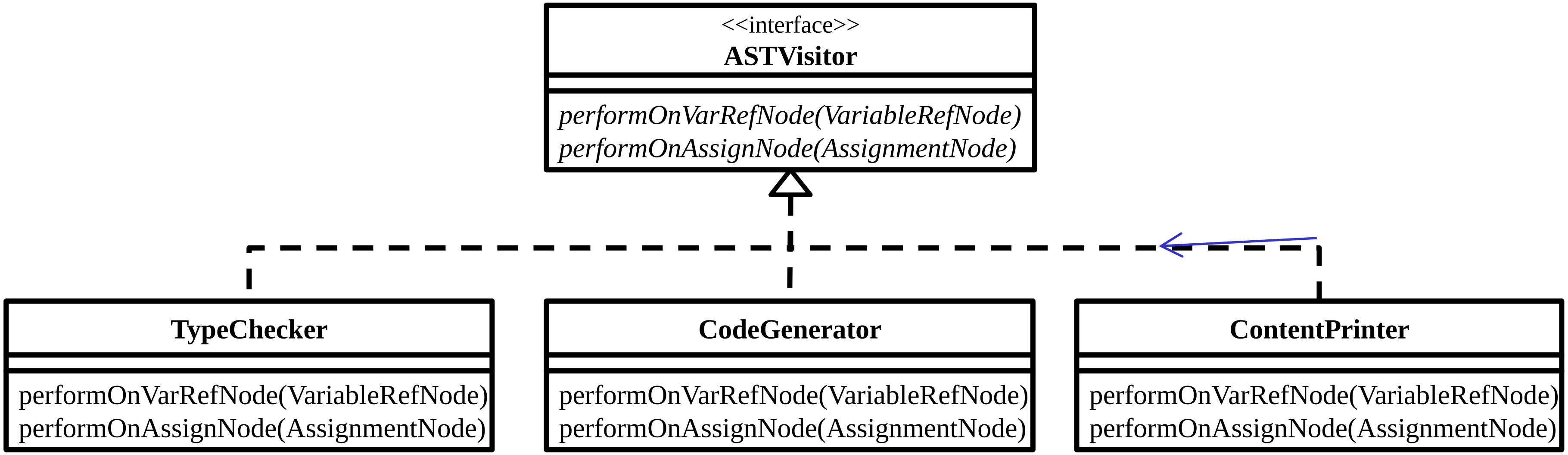
CodeGenerator

performOnVarRefNode(VariableRefNode)
performOnAssignNode(AssignmentNode)

ContentPrinter

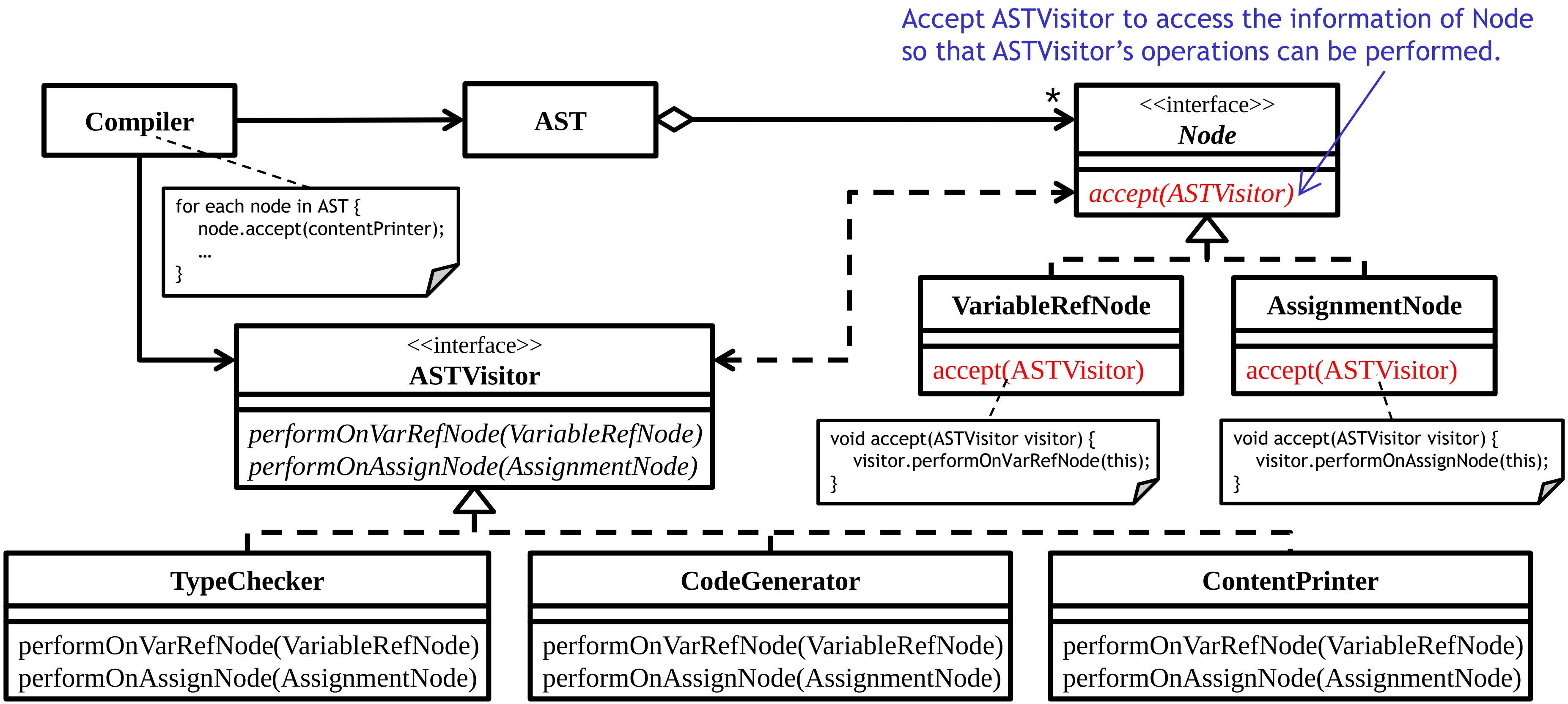
performOnVarRefNode(VariableRefNode)
performOnAssignNode(AssignmentNode)

Generalize common features



<<interface>>
ASTVisitor
performOnVarRefNode(VariableRefNode)
performOnAssignNode(AssignmentNode)
TypeChecker
performOnVarRefNode(VariableRefNode)
performOnAssignNode(AssignmentNode)
CodeGenerator
performOnVarRefNode(VariableRefNode)
performOnAssignNode(AssignmentNode)
ContentPrinter
performOnVarRefNode(VariableRefNode)
performOnAssignNode(AssignmentNode)

Program to an interface



Accept ASTVisitor to access the information of Node so that ASTVisitor's operations can be performed.
Compiler
AST
*
<<interface>>
Node
accept(ASTVisitor)
VariableRefNode
accept(ASTVisitor)
AssignmentNode
accept(ASTVisitor)
<<interface>>
ASTVisitor
performOnVarRefNode(VariableRefNode)
performOnAssignNode(AssignmentNode)
TypeChecker
performOnVarRefNode(VariableRefNode)
performOnAssignNode(AssignmentNode)
CodeGenerator
performOnVarRefNode(VariableRefNode)
performOnAssignNode(AssignmentNode)
ContentPrinter
performOnVarRefNode(VariableRefNode)
performOnAssignNode(AssignmentNode)

```
for each node in AST {
    node.accept(contentPrinter);
    …
}
```

```
void accept(ASTVisitor visitor) {
    visitor.performOnVarRefNode(this);
}
```

```
void accept(ASTVisitor visitor) {
    visitor.performOnAssignNode(this);
}
```

Recurrent Problem

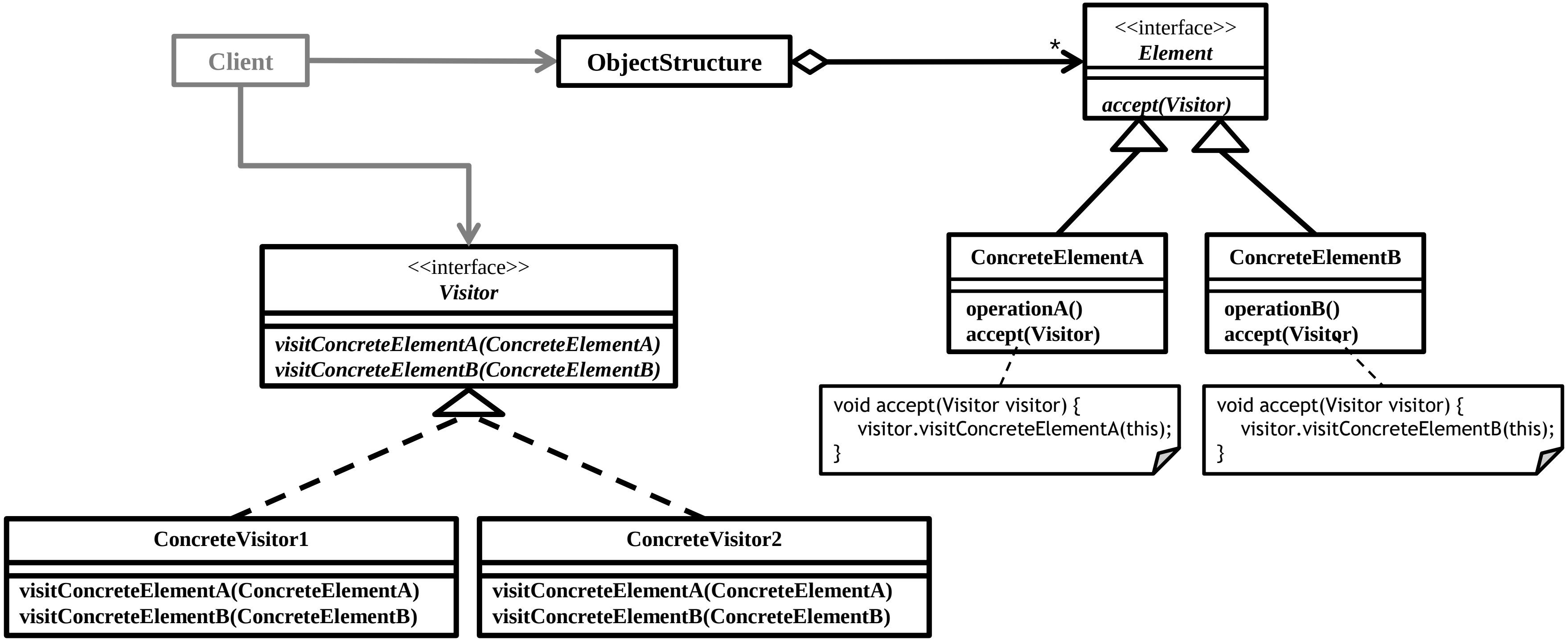
- The problem is that distributing all these operations across the various classes in an object structure leads to a system that's hard to understand, maintain, and change. Moreover, adding a new operation usually requires recompiling all of these classes.

Intent

❑ Represent an operation to be performed on the elements of an object structure.

❑ Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Visitor Pattern Structure$_1$


Client
ObjectStructure
*
<<interface>>
Element
accept(Visitor)
<<interface>>
Visitor
visitConcreteElementA(ConcreteElementA)
visitConcreteElementB(ConcreteElementB)
ConcreteElementA
operationA()
accept(Visitor)
ConcreteElementB
operationB()
accept(Visitor)
ConcreteVisitor1
visitConcreteElementA(ConcreteElementA)
visitConcreteElementB(ConcreteElementB)
ConcreteVisitor2
visitConcreteElementA(ConcreteElementA)
visitConcreteElementB(ConcreteElementB)

```
void accept(Visitor visitor) {
    visitor.visitConcreteElementA(this);
}
```

```
void accept(Visitor visitor) {
    visitor.visitConcreteElementB(this);
}
```