

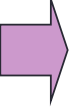
DESIGN PRINCIPLES AND PATTERNS

04. DESIGN PATTERN 1

Nguyen Thi Thu Trang
trangntt@soict.hust.edu.vn

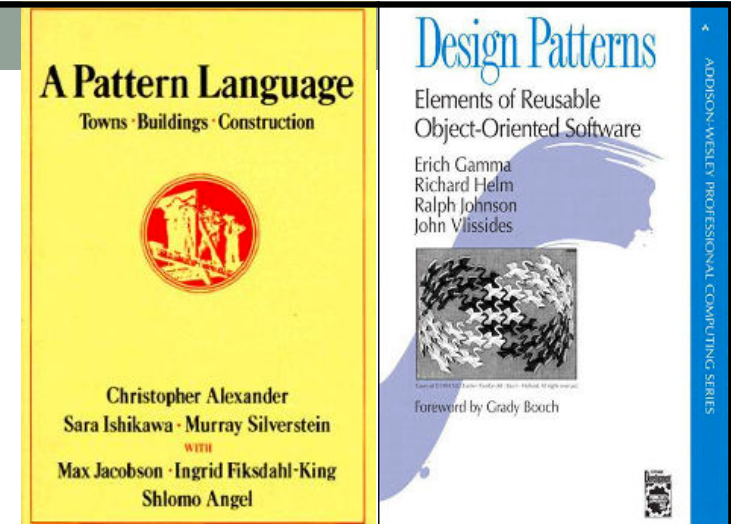


Content

- 
- 1. Introduction to Design Patterns
 - 2. Singleton
 - 3. Template Method
 - 4. Factory Method

Design Patterns

- Published in 1994
- “Each pattern describes a **problem** which occurs over and over **again** in our environment, and then describes the **core of the solution** to that problem, in such a way that you can use this solution **a million times over**, without ever doing it the same way twice”
 - Christopher Alexander
- Today’s amazon.com stats



Amazon Best Sellers Rank: #2,069 in Books ([See Top 100 in Books](#))

#1 in [Books](#) > [Computers & Internet](#) > [Computer Science](#) > [Software Engineering](#) > [Design Tools & Techniques](#)
#1 in [Books](#) > [Computers & Internet](#) > [Programming](#) > [Software Design, Testing & Engineering](#) > [Software Reuse](#)
#3 in [Books](#) > [Nonfiction](#) > [Foreign Language Nonfiction](#) > [French](#)

What and why design patterns?

- A standard solution to a common programming problem
 - a design or implementation structure that achieves a particular purpose
 - a high-level programming idiom
- A technique for making code more flexible
 - reduce coupling among program components
- Short-hand for describing program design
 - a description of connections among program components (static structure)
 - the shape of a heap snapshot or object model (dynamic behaviour)

Whence design patterns?



- The Gang of Four (GoF)
 - Gamma, Helm, Johnson, Vlissides
- Each an aggressive and thoughtful programmer
- Empiricists, not theoreticians
- Found they shared a number of “tricks” and decided to codify them – a key rule was that nothing could become a pattern unless they could identify at least three real examples

GoF patterns: three categories

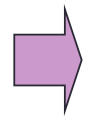
- *Creational Patterns* – these abstract the object-instantiation process
 - Factory Method, Abstract Factory, Singleton, Builder, Prototype
- *Structural Patterns* – these abstract how objects/classes can be combined
 - Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy
- *Behavioral Patterns* – these abstract communication between objects
 - Command, Interpreter, Iterator, Mediator, Observer, State, Strategy, Chain of Responsibility, Visitor, Template Method

Design Patterns classification

Purpose Scope	Creation	Structure	Behaviour
Class	<i>Factory method</i>	Adapter (class)	Interpreter, Template Method
Object	<i>Abstract Factory</i>	Adapter (object)	Chain of Responsibility
	<i>Builder</i>	Bridge	Command
	<i>Prototype</i>	Composite	Iterator
	<i>Singleton</i>	Decorator	Mediator
		Façade	Memento
		Flyweight	Observer
		Proxy	State, Strategy, Visitor

Content

1. Introduction to Design Patterns



2. Singleton

3. Template Method

4. Factory Method

An example of a GoF pattern

- Given a class A, what if you want to guarantee that there is precisely one instance of A in your program? And you want that instance globally available?
 - First, why might you want this?
 - Second, how might you achieve this?

Implementing Singleton

- Make constructor(s) **private** so that they can not be called from outside by clients.
- Declare a single **private static** instance of the class.
- Write a public **getInstance ()** or similar method that allows access to the single instance.
 - May need to protect / synchronize this method to ensure that it will work in a multi-threaded program.

Several solutions

```
public class Singleton {  
    private static final Singleton instance  
        = new Singleton(); // Private constructor prevents  
                           // instantiation from other classes  
    private Singleton() { }  
    public static Singleton getInstance() {  
        return instance;  
    }  
}
```

```
public class Singleton {  
    private static Singleton _instance;  
    private Singleton() { }  
    public static synchronized Singleton getInstance() {  
        if (null == _instance) {  
            instance = new Singleton();  
        } return _instance;  
    }  
}
```

Possible reasons for Singleton

- Make it easier to ensure some **key invariants**
- Make it easier to control when that **single** instance is created – can be important for **large objects**
- E.g.
 - One **RandomNumber** generator
 - One **Restaurant**, one **ShoppingCart**
 - One **KeyboardReader**, etc...

An Alternative: Static Property/Class

Why Singleton???

An Alternative: Static Property/Class

- Testability
 - Multiple tests might effect each other
 - Hard or impossible to mock
- Extensibility
 - Not possible to inherit from a static class
 - Not possible to write an extension method to a static class

Practice: Applying Singleton in Codebase

- Can we apply **Singleton** in any part of the codebase for a specific requirement/a better design?



Content

1. Introduction to Design Patterns

2. Singleton

→ 3. Template Method

4. Factory Method

In a Beverage Machine

TeaMaker

- getCup()
- boilWater()
- steepTeaBag()
- pourInCup()
- addSugar()
- addLemon()

CoffeeMaker

- getCup()
- boilWater()
- brewCoffee()
- pourInCup()
- addSugar()
- addMilk()

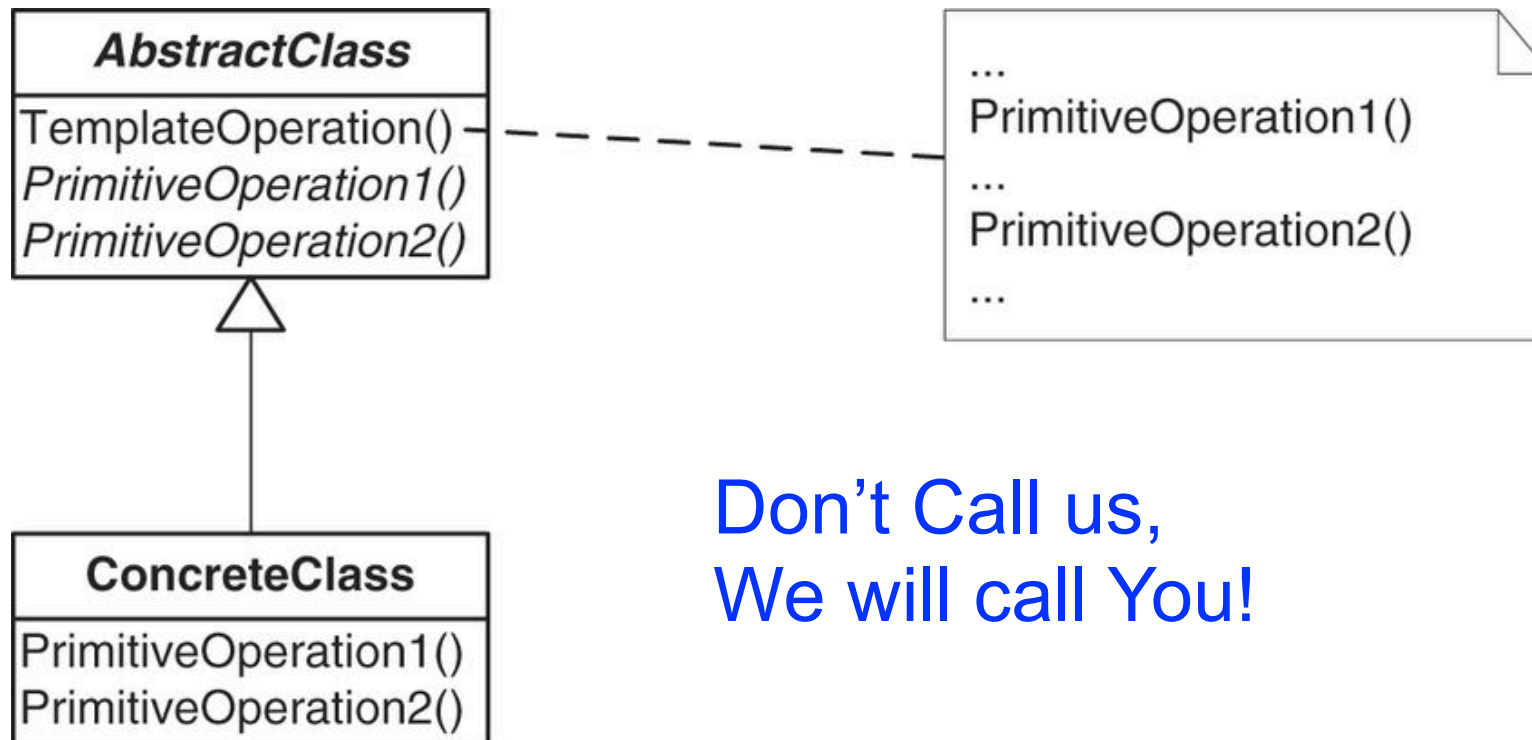
CoffeeVerkeerdMaker

- getCup()
- boilWater()
- brewCoffee()
- pourInCup()
- addMilk()
- addSugar()

- Duplicated properties
- Optional behaviors

➔ How to generalize these beverage makers for a single behavior to client?

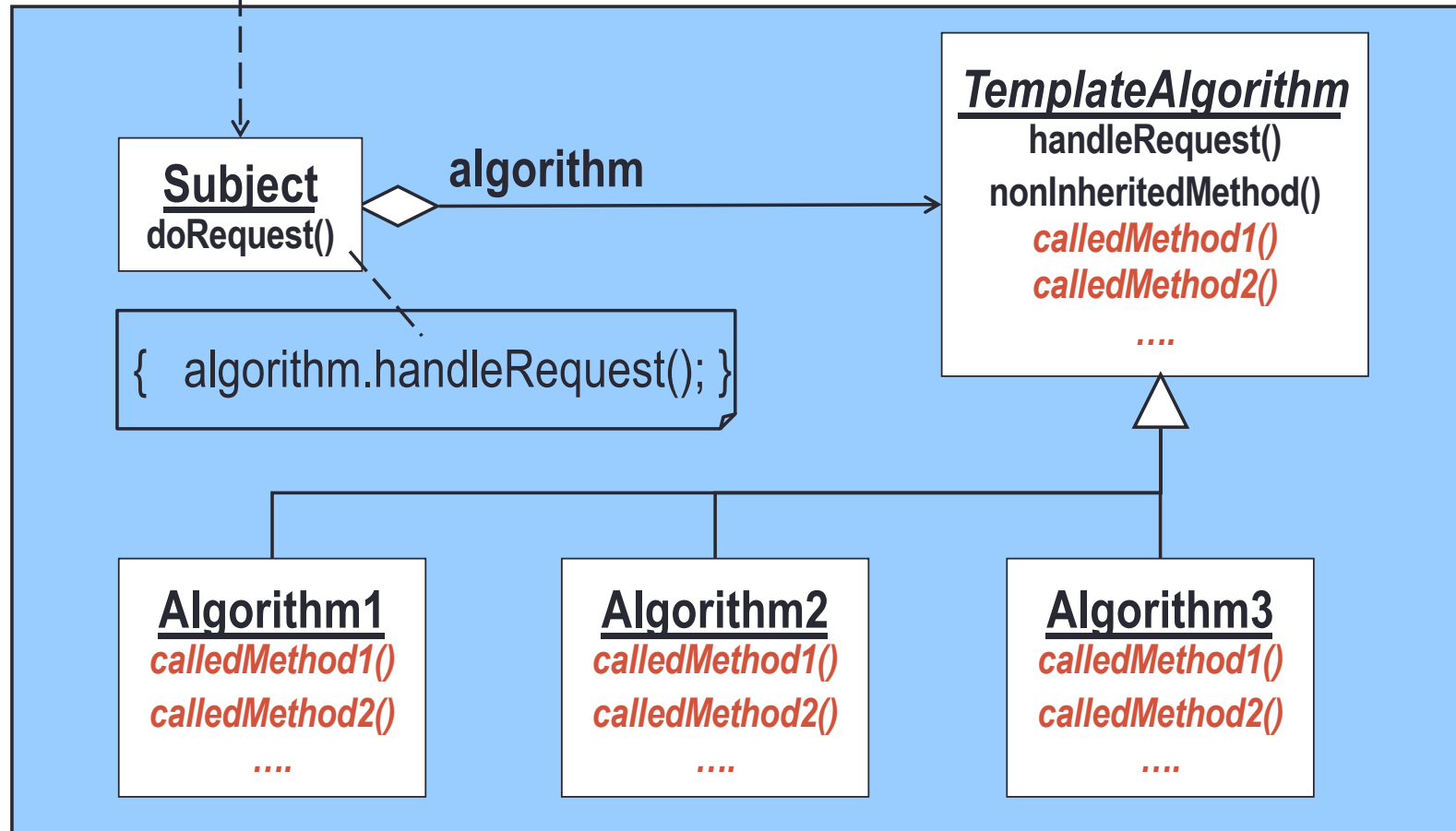
Template Method: Structure



Don't Call us,
We will call You!

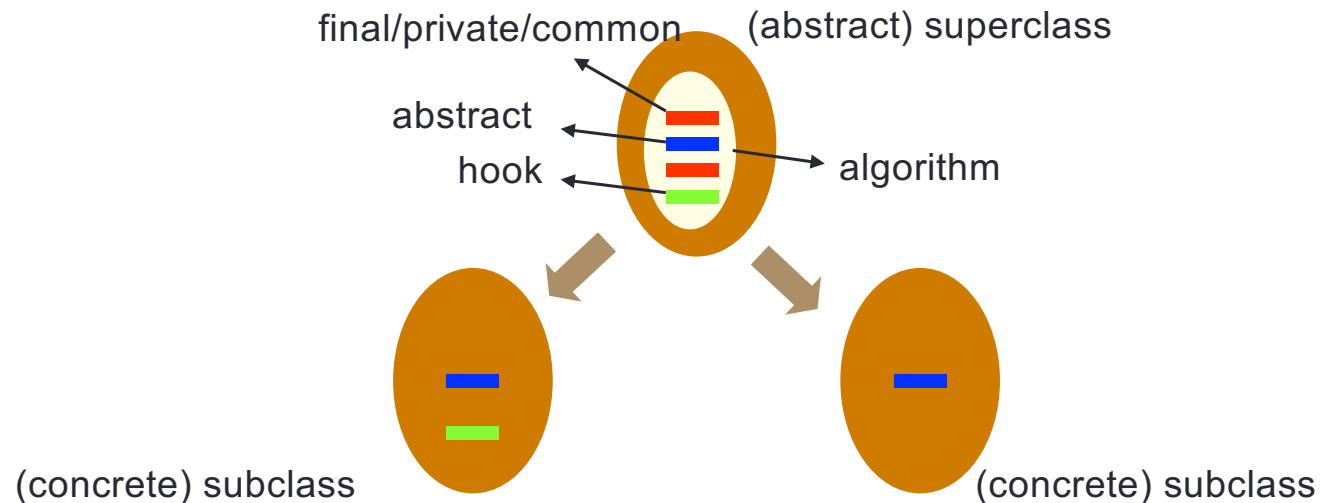
Client

Template Method: More Structure



Template Method: Intent

- Define the skeleton of an algorithm in an operation, deferring some steps to subclasses
 - Subclasses redefine certain steps of an algorithm without changing the algorithm's structure

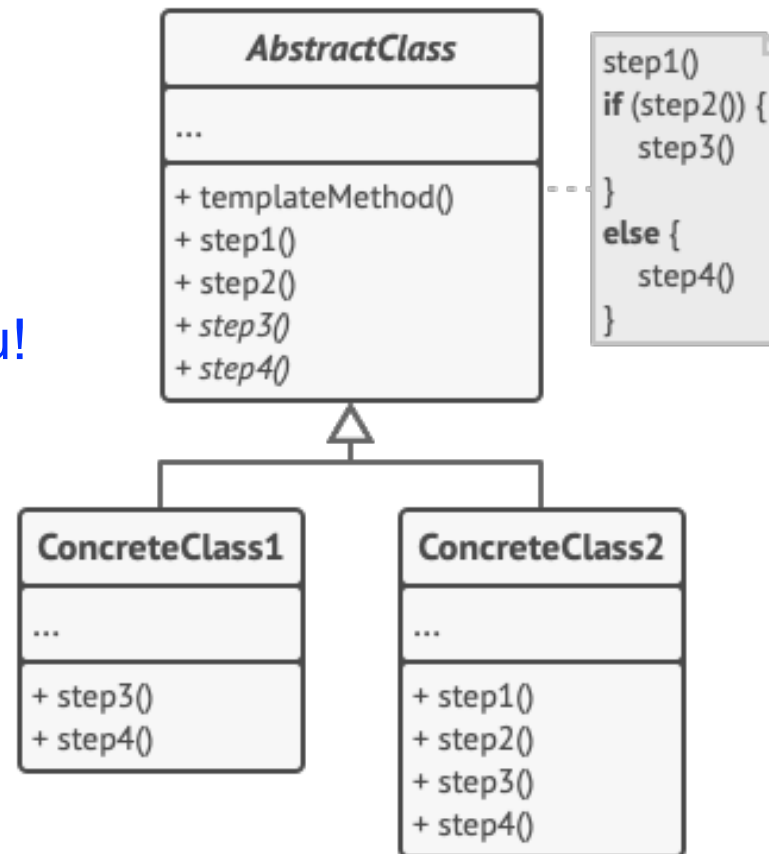


Template Pattern: Intent (2)

- Three modes
 - Forcing subclasses to implement certain steps: **abstract** or **interface**
 - Allowing subclasses to complement certain steps (through overridable hooks): **non-abstract**
 - Allowing subclasses to redefine steps (through overridable algorithm methods): **protected**
- Non-overridable steps: **final**
- Non-inherited steps: **private**

Template Method: More Structure

Don't Call us,
We will call You!



In a Beverage Machine

TeaMaker

- getCup()
- boilWater()
- steepTeaBag()
- pourInCup()
- addSugar()
- addLemon()

CoffeeMaker

- getCup()
- boilWater()
- brewCoffee()
- pourInCup()
- addSugar()
- addMilk()

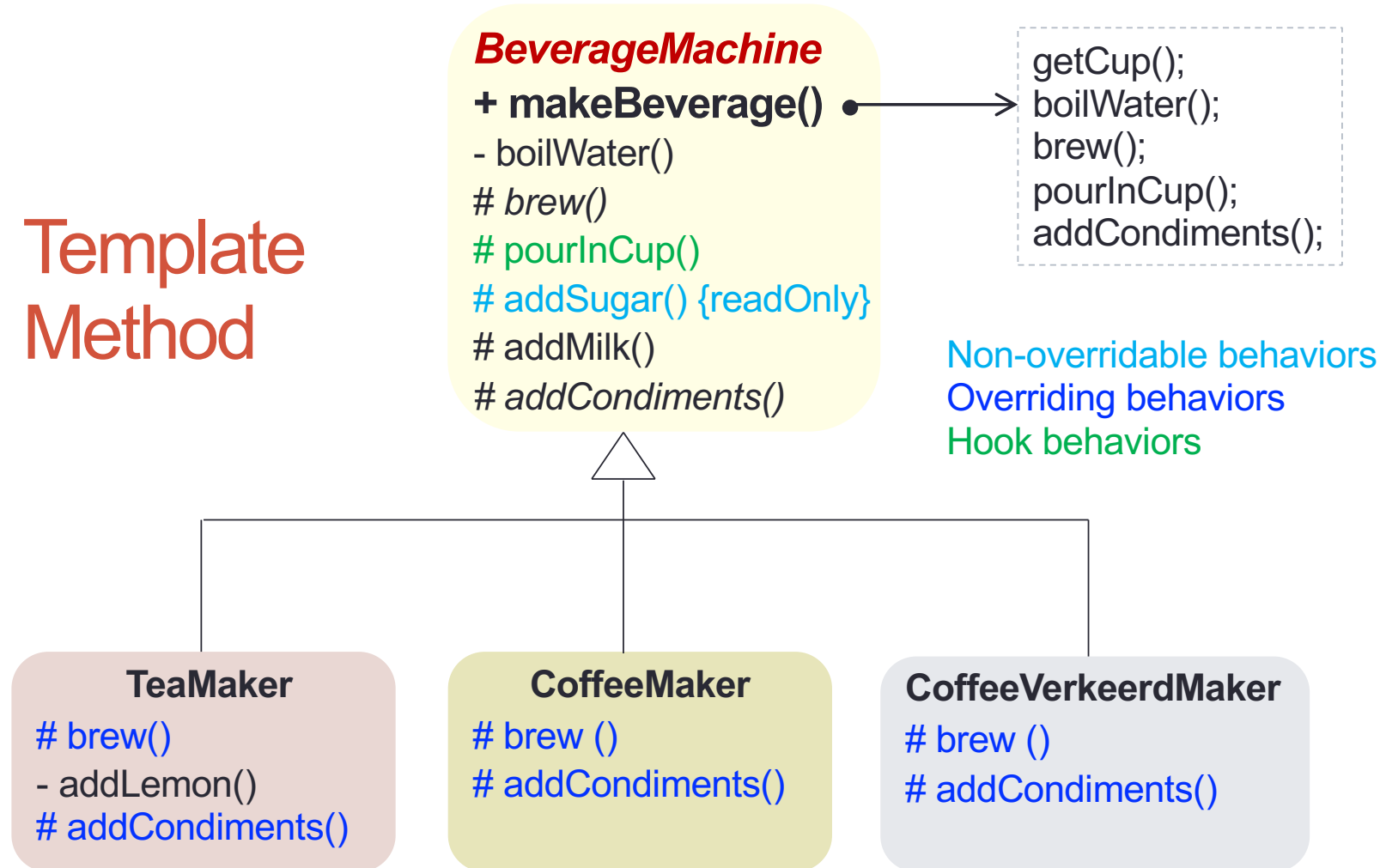
CoffeeVerkeerdMaker

- getCup()
- boilWater()
- brewCoffee()
- pourInCup()
- addMilk()
- addSugar()

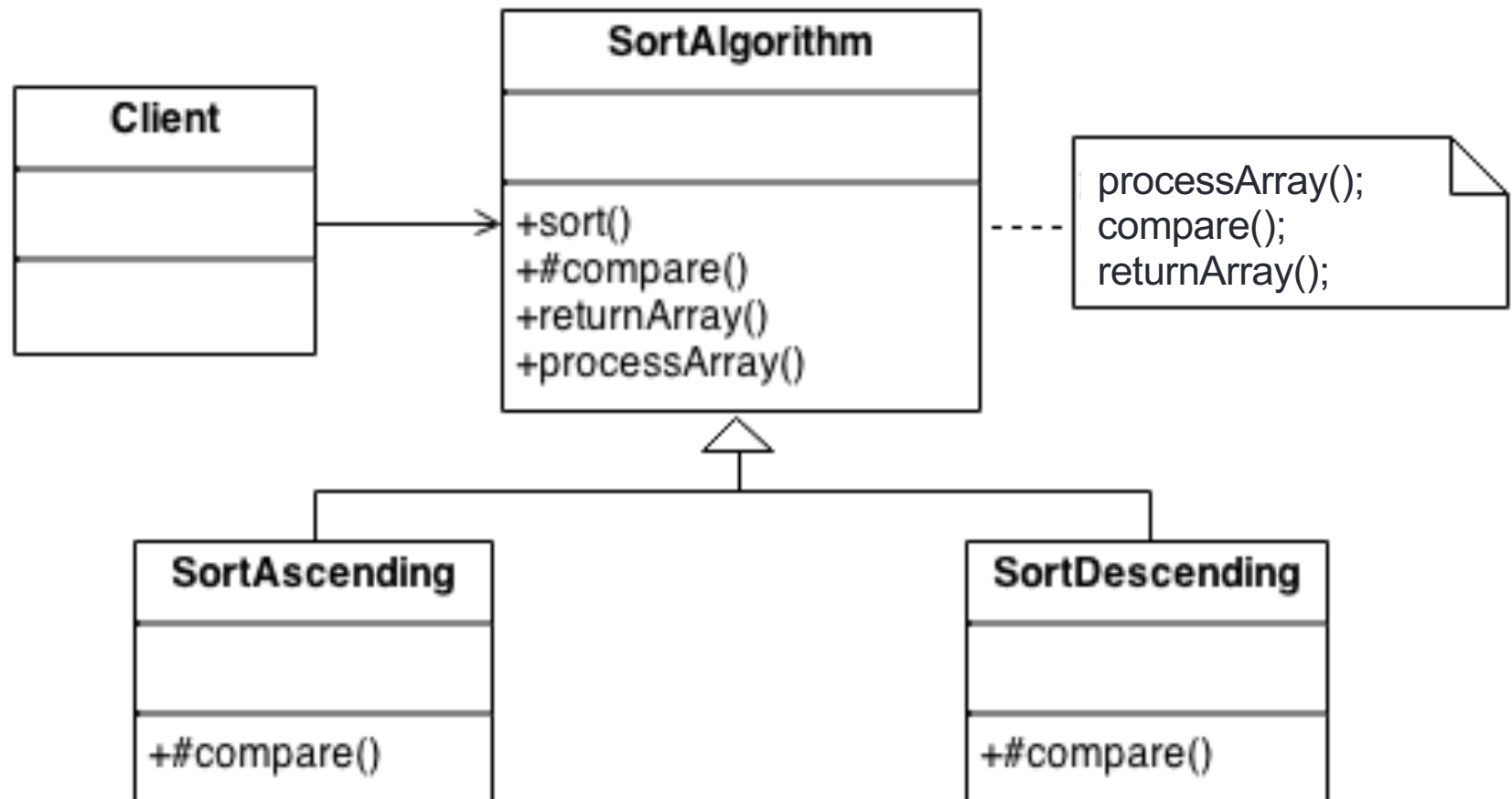
- Duplicated properties
- Optional behaviors

➔ How to generalize these beverage makers for a single behavior to client?

Template Method



Template Method: More Example



Template Method: SocialNetwork

Subclasses:

- Facebook
- Twitter

```

public abstract class SocialNetwork {
    String userName; String password;
    Network() {}
    /** * Publish the data to whatever network. */
    public boolean post(String message) {
        // Authenticate before posting. Every network uses
        // a different authentication method.
        if (logIn(this.userName, this.password)) {
            // Send the post data
            boolean result = sendData(message.getBytes());
            logOut();
            return result;
        }
        return false;
    }
    abstract boolean logIn(String userName, String password);
    abstract boolean sendData(byte[] data);
    abstract void logOut();
}

```

Template Pattern Advantages

- Separates what changes from what does not
- Bulk of unchanging code can be easily reused
- Detailed differences are all that the subclasses worry about
- This pattern is being used every time when you extend a class in Java
 - E.g. Override paint() method (hook)
 - *public class MyFrame extends JFrame*
 - *public class MyApplet extends Applet*

```
public class MyFrame extends JFrame {  
    public MyFrame(String title) {  
        super(title);  
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        this.setSize(300, 300);  
        this.setVisible(true);  
    }
```

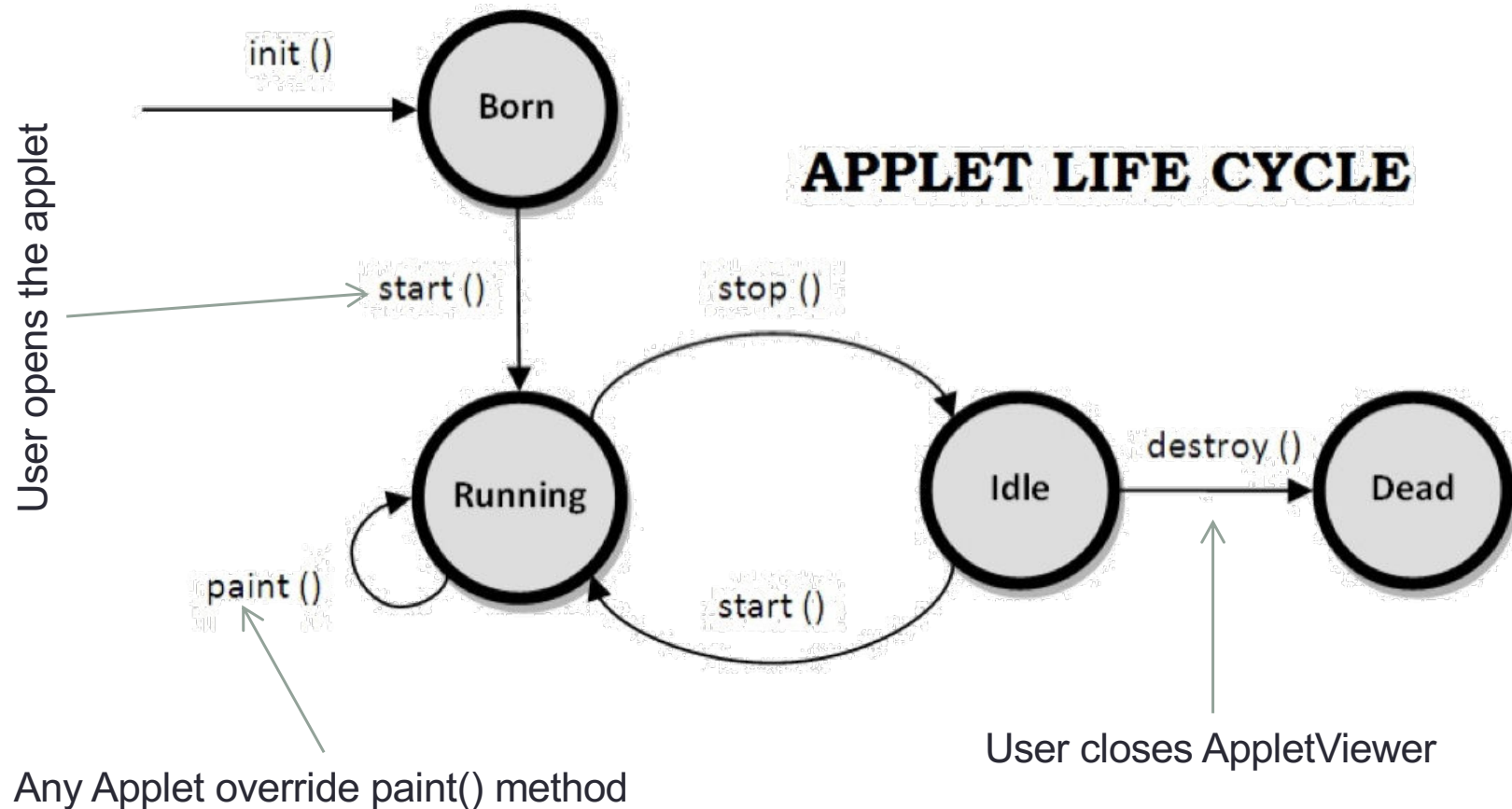
JFrame of Java Swing

```
    public void paint(Graphics graphics) {  
        super.paint(graphics);  
        String msg = "Hello world!";  
        graphics.drawString(msg, 100, 100);  
    }
```

JFrame's update()
algorithm calls paint()

```
    public static void main(String[] args) {  
        MyFrame myFrame = new MyFrame("My first frame");  
    }  
}
```

Template Method for Applet Lifecycle



Practice: Applying Template Method in Codebase

- Can we apply **Template Method** in any part of the codebase for a specific requirement or a better design?



Content

1. Introduction to Design Patterns
2. Singleton
3. Template Method
- ➔ 4. Factory Method

```

1 public class PizzaStore {
2
3     Pizza orderPizza(String type) {
4
5         Pizza pizza;
6
7         if (type.equals("cheese")) {
8             pizza = new CheesePizza();
9         } else if (type.equals("greek")) {
10            pizza = new GreekPizza();
11        } else if (type.equals("pepperoni")) {
12            pizza = new PepperoniPizza();
13        }
14
15        pizza.prepare();
16        pizza.bake();
17        pizza.cut();
18        pizza.box();
19
20        return pizza;
21    }
22
23 }
24

```

Despite using "interface", this code depends on "CheesePizza" or any other types of pizza

Creation

Problems:

- Mixing creation and preparing/ordering pizza: If we change the way to create pizza

=> **Violate SRP principle**

- If we have a new type or remove an existing type of pizza

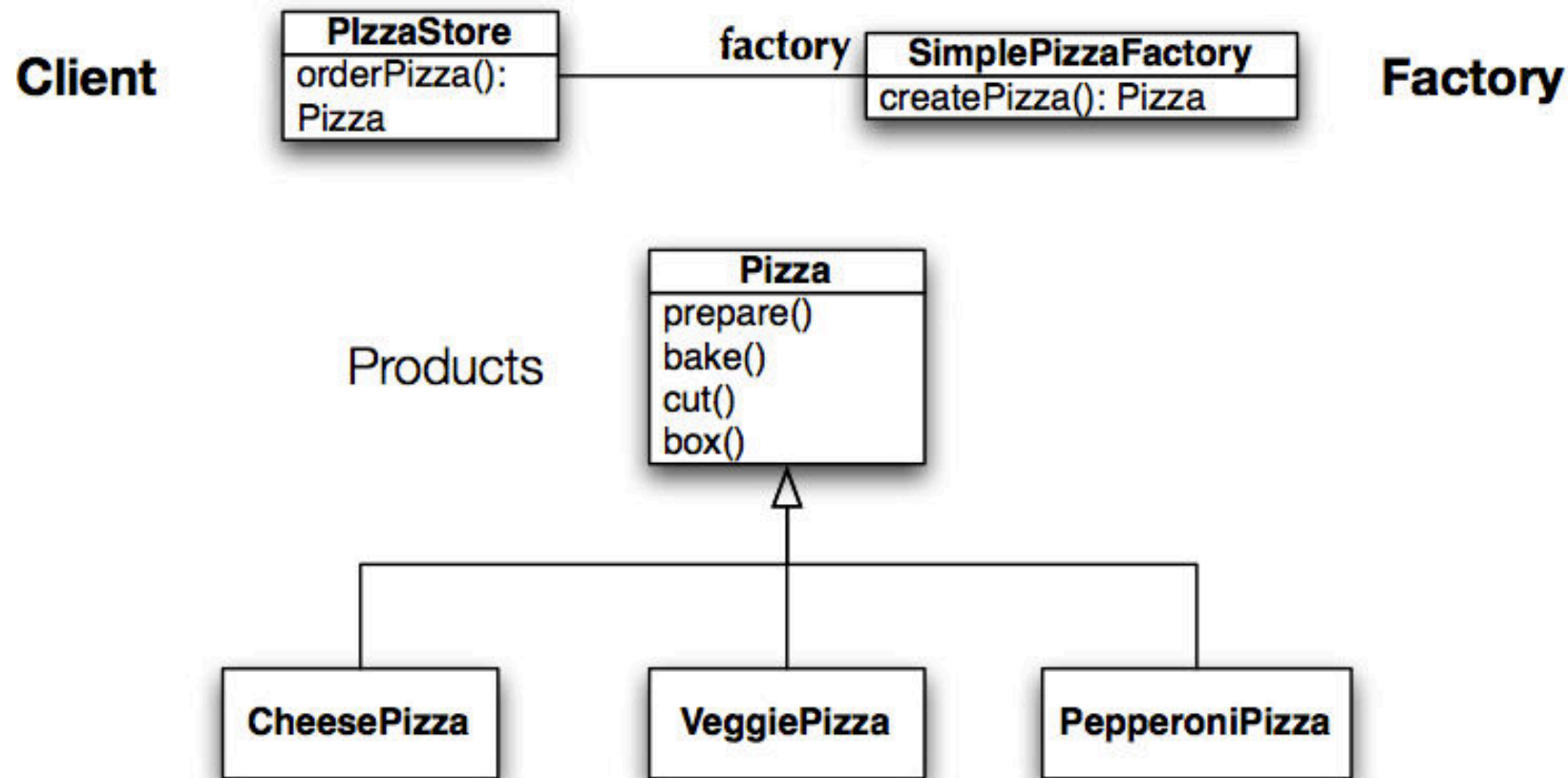
=> **Violate OCP principle**

PizzaStore Example

PizzaStore: Separate Creation and Usage

- Separate the process of creating a pizza from the process of preparing/ordering a pizza
- Encapsulate Creation Code: Put it in a separate class
 - That new class depends on the concrete classes, but those dependencies no longer impact the preparation code

Separating Creation (Factory) and Usage




```

1 public class PizzaStore {
2
3     private SimplePizzaFactory factory;
4
5     public PizzaStore(SimplePizzaFactory factory) {
6         this.factory = factory;
7     }
8
9     public Pizza orderPizza(String type) {
10
11         Pizza pizza = factory.createPizza(type);
12
13         pizza.prepare();
14         pizza.bake();
15         pizza.cut();
16         pizza.box();
17
18         return pizza;
19     }
20 }
21
22

```

✓ Separating the creation and prepare/order pizza



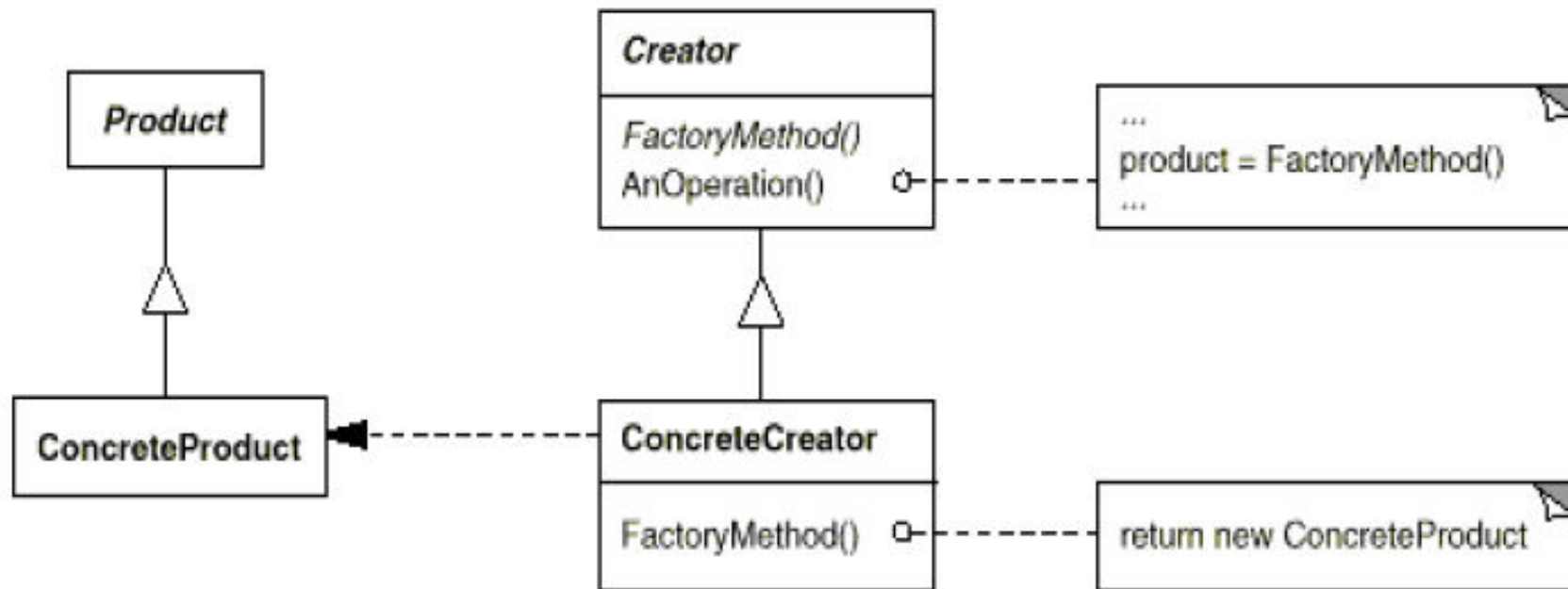
```

1 public class SimplePizzaFactory {
2
3     public Pizza createPizza(String type) {
4         if (type.equals("cheese")) {
5             return new CheesePizza();
6         } else if (type.equals("greek")) {
7             return new GreekPizza();
8         } else if (type.equals("pepperoni")) {
9             return new PepperoniPizza();
10        }
11    }
12
13 }
14

```

✓ But still violates OCP when adding new pizza type

Factory Method: Structure



Exercise: PizzaStore

- Please give a better version of factory method for PizzaStore
 - When having new type or removing an existing one, the factory doesnot violate OCP

Factory Method: More example

- Display images on a graphical window
- The Graphics object has a drawImage method:
 - `public void drawImage(Image img, int x, int y, panel)`
 - `public void drawImage(Image img, int x, int y, int w, int h, panel)`
- Images are hard drive files in a given format:
 - GIF, JPEG, PNG, BMP, TIFF, ...
- So how do we get an Image object to draw?
- Can't simply say `new Image :`
 - `Image img = new Image("bobafett.gif"); // error`



Toolkits

- Toolkit is a class for GUI system info and resource loading
- Java handles loading of images through Toolkits:
 - `public Image getImage(String filename)`
 - `public Image getImage(URL url)`
- Can't simply say `new Toolkit`:
 - `Toolkit tk = new Toolkit(); // error`
- Have to call a static method to get a toolkit (Why? What is this?):
 - `public static Toolkit getDefaultToolkit()`
 - `=> Toolkit tk = Toolkit.getDefaultToolkit(); // ok`

Buggy image client

- The following well-intentioned client does not show the images:

```
public static void main(String[] args) {  
    Toolkit tk = Toolkit.getDefaultToolkit();  
    Image img1 = tk.getImage("calvin.gif");  
    Image img2 = tk.getImage("cuteicecream.jpg");  
    Image img3 = tk.getImage("tinman.png");  
  
    DrawingPanel panel = new DrawingPanel(600, 500);  
    Graphics g = panel.getGraphics();  
    g.drawImage(img1, 0, 0, panel);  
    g.drawImage(img2, 200, 50, panel);  
    g.drawImage(img3, 400, 200, panel);  
}
```

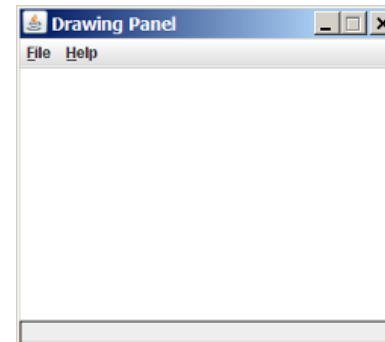


Image loading factory

- The preceding code is too cumbersome to write every time we want to load an image.
 - Let's make a factory method to load images more easily

```
public static Image loadImage(  
    String filename, DrawingPanel panel) {  
    Toolkit tk = Toolkit.getDefaultToolkit();  
    Image img = tk.getImage(filename);  
  
    MediaTracker mt = new MediaTracker(panel);  
    mt.addImage(img, 0);  
    try {  
        mt.waitForAll();  
    } catch (InterruptedException e) {}  
    return img;  
}
```

A Factory Class

- Factory methods are often put into their own class for reusability:

```
public class ImageFactory {  
    public static Image loadImage(  
        String filename, DrawingPanel panel) {  
        Toolkit tk = Toolkit.getDefaultToolkit();  
        Image img = tk.getImage(filename);  
        MediaTracker mt = new MediaTracker(panel);  
        mt.addImage(img, 0);  
        try {  
            mt.waitForAll();  
        } catch (InterruptedException e) {}  
        return img;  
    }  
    public static Image loadImage(  
        File file, DrawingPanel panel) {  
        return loadImage(file.toString(), panel);  
    }  
}
```

Factory Method: DateFormat as a factory

- `DateFormat` class knows how to format dates/times as text
 - Options: Just date? Just time? Date+time? Where in the world?
 - Instead of passing all options to constructor, use factories.
 - The subtype created doesn't need to be specified.

```
DateFormat df1 = DateFormat.getDateInstance();  
DateFormat df2 = DateFormat.getTimeInstance();  
DateFormat df3 = DateFormat.getDateInstance(  
    DateFormat.FULL, Locale.FRANCE);  
  
Date today = new Date();  
System.out.println(df1.format(today)); // "Apr 20, 2011"  
System.out.println(df2.format(today)); // "10:48:00 AM"  
System.out.println(df3.format(today)); // "mcredi 20 avril  
    2011"
```

How to implement Factory Method

- The factory itself should not be instantiated
 - Make constructor private
- Factory only uses static methods to construct components
- Factory should offer as simple an interface to client code as possible
 - Don't demand lots of arguments; possibly overload factory methods to handle special cases that need more arguments
- Factories are often designed for reuse on a later project or for general use throughout your system

Factory Method: Using Existing Factories

- Setting borders on buttons and panels
 - use built-in `BorderFactory` class

```
myButton.setBorder (
```

```
BorderFactory.createRaisedBevelBorder() ) ;
```

- Setting hot-key "accelerators" on menus
 - use built-in `KeyStroke` class

```
menuItem.setAccelerator (
```

```
45 KeyStroke.getKeyStroke ( 'T' ,  
KeyEvent.ALT_MASK ) ) ;
```

Border Factory in Java

- Java graphical components like `DrawingPanel` can have borders:

```
public void setBorder(Border border)
```

- But `Border` is an interface; cannot construct a new `Border`.
 - There are many different kinds of borders (classes).
- Instead, use the provided `BorderFactory` class to create them:

```
public static Border createBevelBorder(...)  
public static Border createEtchedBorder(...)  
public static Border createLineBorder(...)  
public static Border createMatteBorder(...)  
public static Border createTitledBorder(...)
```

- Avoids a constructor that takes too many "option / flag" arguments.

When using Factory Method?

- Clients don't want or don't be allowed to know how to create objects
- Complex information to creation objects or cumbersome initialization
 - ⇒ Simplify the interface to client code
 - ⇒ Localize the logic to instantiate a complex object
- A class requires its subclasses to specify the objects it creates
- A class cannot anticipate the type of objects it needs to create beforehand

Practice: Applying Factory Method in Codebase

- Can we apply **Factory Method** in any part of the codebase for a specific requirement or a better design?



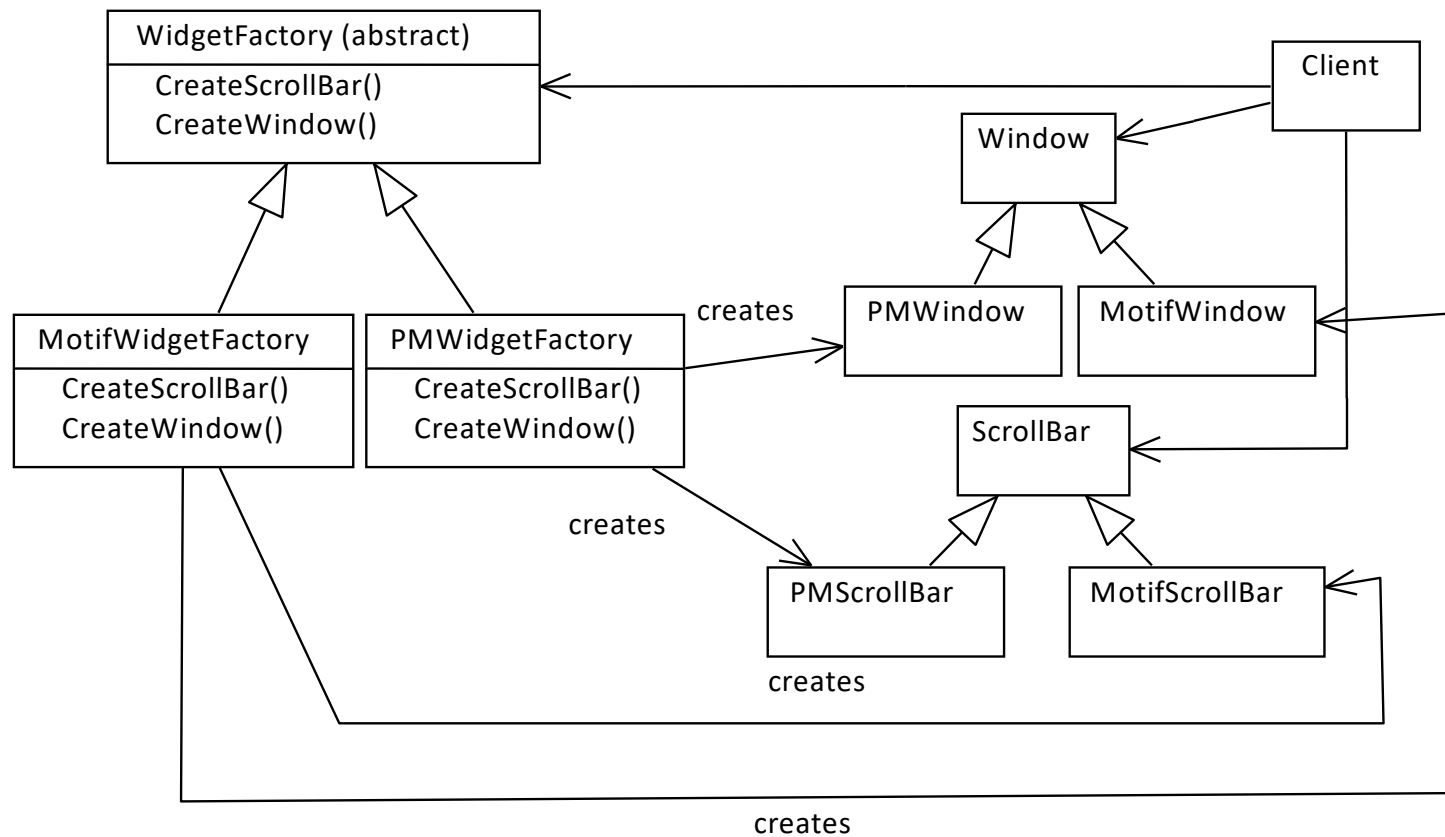
More discussion: Abstract Factory

- Factory Method pattern: a factory that can be constructed and has an overridable method to create its objects
 - can be subclassed to make new kinds of factories
- Abstract Factory pattern: when the topmost factory class and its creational method are abstract

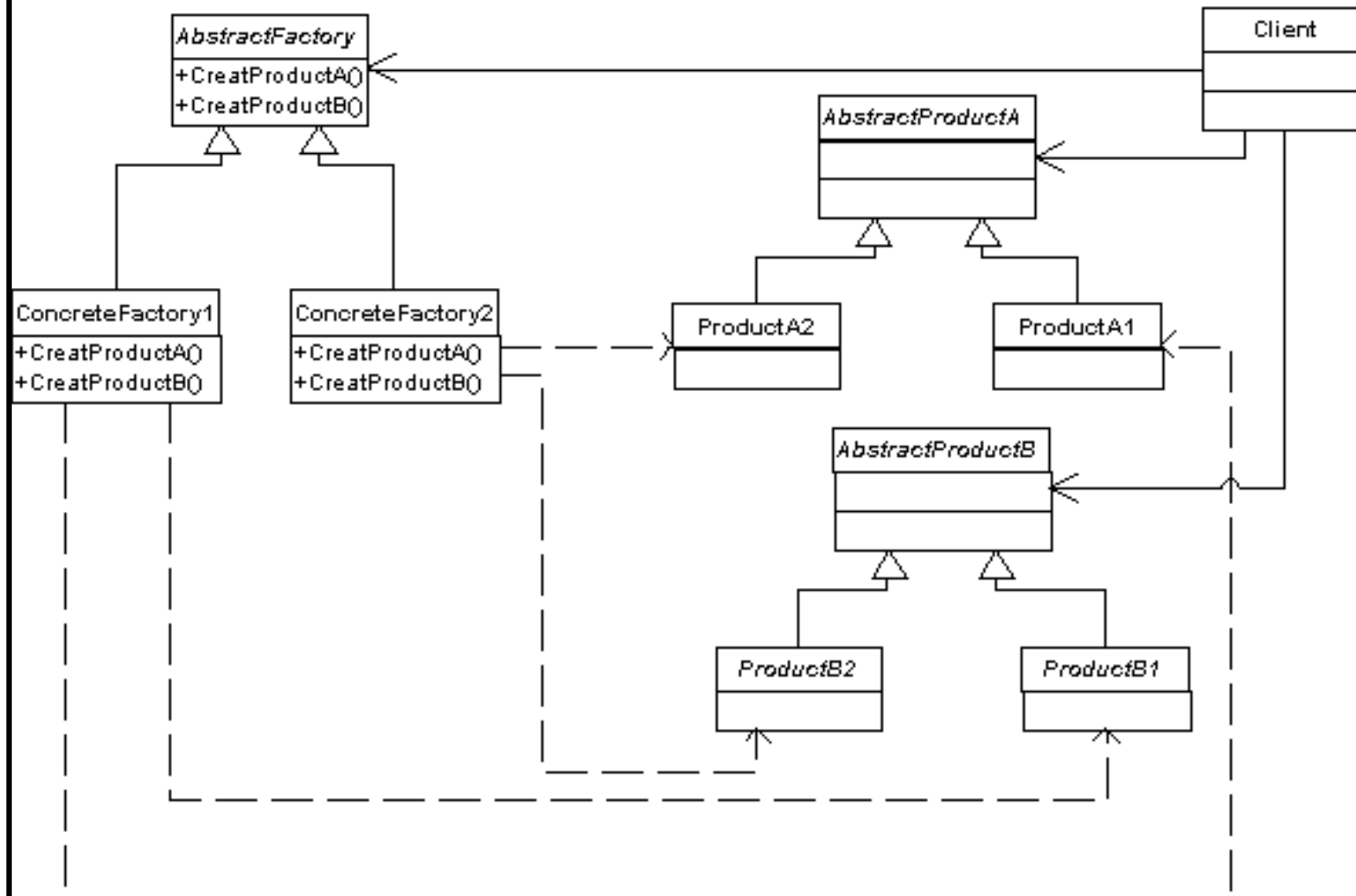
More discussion: When Abstract Factory?

- A system should be independent of how its products are created, composed and represented
- A system should be configured with one of multiple families of products
- A family of related product objects is designated to be used together, and you need to enforce this constraint
- You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations

Abstract Factory: Example



Abstract Factory: Structure



Abstract Factory: Consequences

- Good:
 - Isolates concrete classes
 - All manipulation on client-side done through abstract interfaces
 - Makes exchanging product families easy
 - Just change the ConcreteFactory
 - Enforces consistency among products
- Bad
 - Supporting new kinds of products is difficult
 - Have to reprogram Abstract Factory and all subclasses
 - But it's not so bad in dynamically typed languages

