Software Design and Architecture

Command and Template Patterns

Design principles

High-level principles

- **S**ingle Responsibility
- **O**pen/Closed
- **L**iskov Substitution Principle
- **I**nterface Segregation
- **D**ependency Inversion

Low-level principles

- Encapsulate what varies
- Program to interfaces, not implementations
- Favor composition over inheritance
- Strive for loose coupling

Command

Behavioral Patterns

- observer
- decorator
- strategy
- command

Creational Patterns

- factory method
- abstract factory
- singleton

Problem

Your program is in charge of all action events, implementing these would lead to huge if-elseif or switch blocks.

```
public void actionPerformed(ActionEvent e)
{
  Object o = e.getSource();
  if (o instanceof fileNewMenuItem)
    doFileNewAction();
  else if (o instanceof fileOpenMenuItem)
    doFileOpenAction();
  else if (o instanceof fileOpenRecentMenuItem)
    doFileOpenRecentAction();
  else if (o instanceof fileSaveMenuItem)
    doFileSaveAction();
  // and more ...
}
```

Command Pattern solution

```
public interface Command
{
  public void execute();
}
```

```
public class FileOpenMenuItem extends JMenuItem
implements Command
{
  public void execute()
  {
    // your business logic goes here
  }
}
```

```
public void actionPerformed(ActionEvent e)
{
  Object o = e.getSource();
  if (o instanceof fileNewMenuItem)
    doFileNewAction();
  else if (o instanceof fileOpenMenuItem)
    doFileOpenAction();
  else if (o instanceof fileOpenRecentMenuItem)
    doFileOpenRecentAction();
  else if (o instanceof fileSaveMenuItem)
    doFileSaveAction();
  // and more ...
}
```

```
public void actionPerformed(ActionEvent e)
{
  Command command = (Command)e.getSource();
  command.execute();
}
```

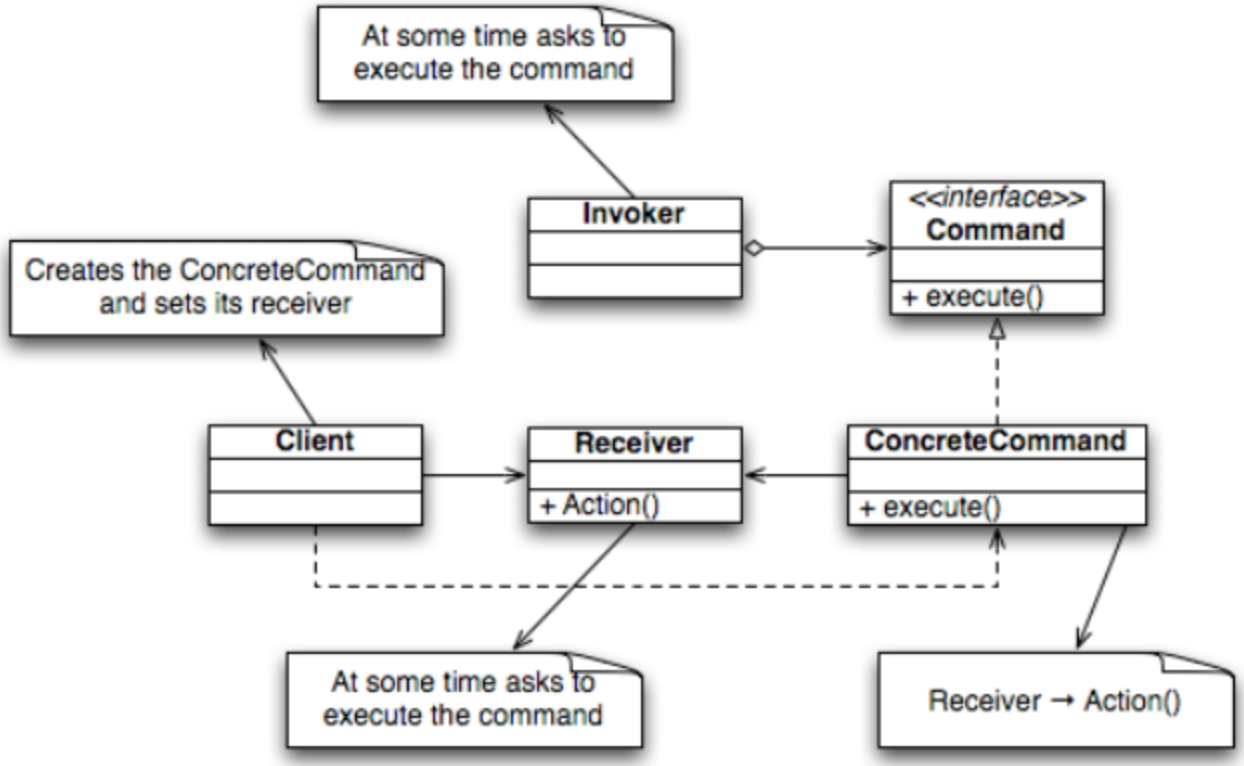
Command Pattern

- Move the code for each individual action into it's own separate class.
- Each of these classes implements the same Interface, allowing the code that uses them to interact solely with the Interface and not know or care about the individual classes.
- This increases Cohesion because each class is responsible for one discrete set of logic.
- This decreases Coupling because the code calling the command only deals with one type, the Interface.

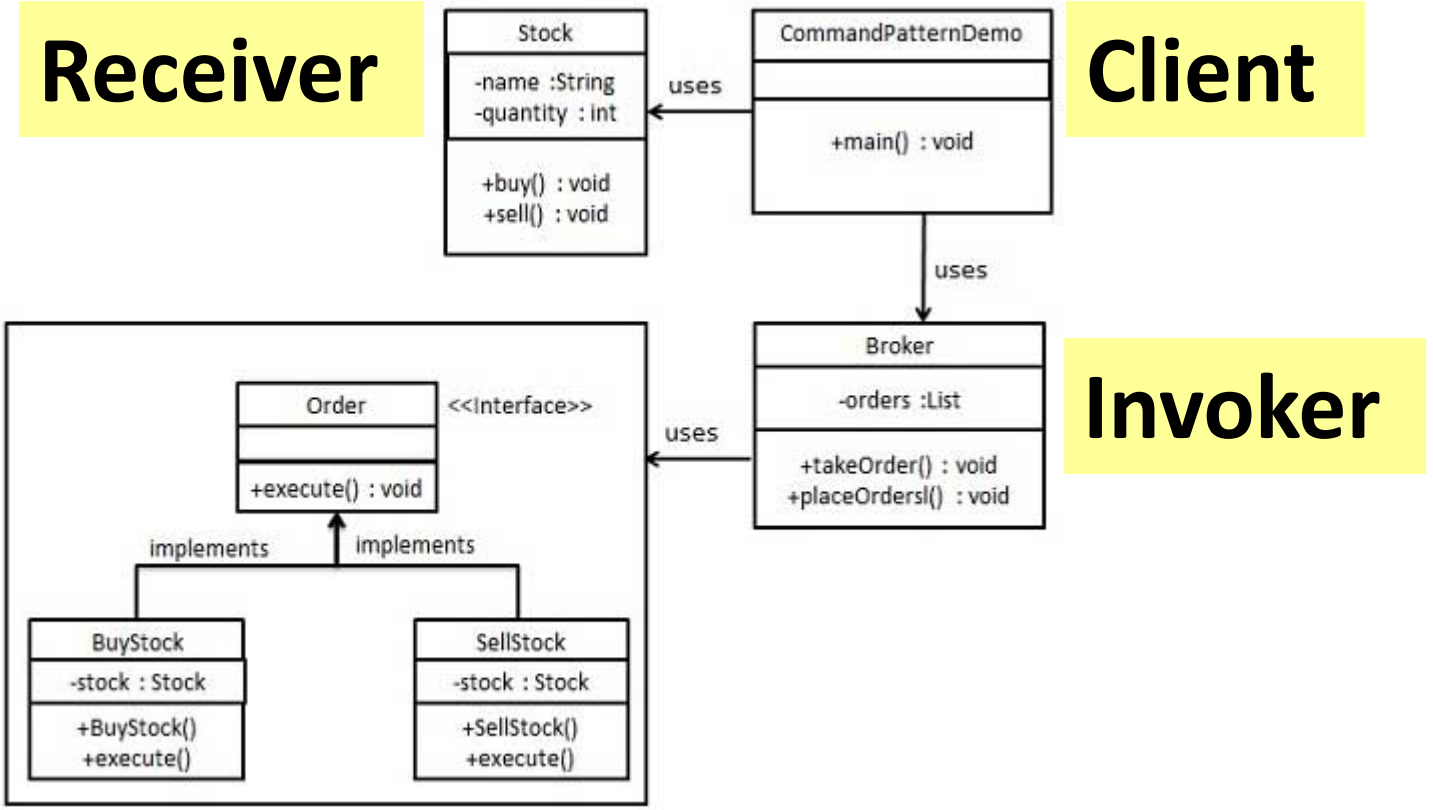
Command Pattern

The Command Pattern encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations

Command Pattern



Command Pattern



Receiver

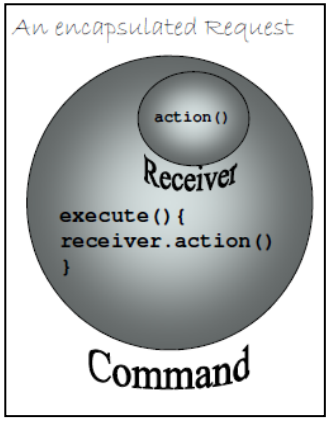
```
public class Stock {

  private String name = "ABC";
  private int quantity = 10;

  public void buy(){
    System.out.println("Stock [ Name: "+name+",
      Quantity: " + quantity +" ] bought");
  }
  public void sell(){
    System.out.println("Stock [ Name: "+name+",
      Quantity: " + quantity +" ] sold");
  }
}
```

Command

```
public interface Order {
   void execute();
}
```

```
public class BuyStock implements Order {
   private Stock abcStock;//Receiver

   public BuyStock(Stock abcStock){
      this.abcStock = abcStock;
   }

   public void execute() {
      abcStock.buy();
   }
}
```

```
public class SellStock implements Order {
   private Stock abcStock;//Receiver

   public SellStock(Stock abcStock){
      this.abcStock = abcStock;
   }

   public void execute() {
      abcStock.sell();
   }
}
```



Invoker

```
import java.util.ArrayList;
import java.util.List;

  public class Broker {
  private List<Order> orderList = new
ArrayList<Order>();

  public void takeOrder(Order order){
    orderList.add(order);
  }

  public void placeOrders(){

    for (Order order : orderList) {
      order.execute();
    }
    orderList.clear();
  }
}
```

Client

```
public class CommandPatternDemo {
   public static void main(String[] args) {
      Stock abcStock = new Stock();

      BuyStock buyStockOrder = new
BuyStock(abcStock);
      SellStock sellStockOrder = new
SellStock(abcStock);

      Broker broker = new Broker();
      broker.takeOrder(buyStockOrder);
      broker.takeOrder(sellStockOrder);

      broker.placeOrders();
   }
}
```

Command Pattern

```
public interface Command
{
  public void execute();
}
```

You can put any “generic” method in here:

- log()
- **undo()**
- delete()
- load()

Receiver

```
public class TV() {
        public TV() {
                system.out.println("TV Created");
        }

        public on() {
                system.out.println("TV On");
        }

        public off() {
                system.out.println("TV Off");
        }
}
```

Command

```
public interface Command {
        public void execute();
        public void undo();
}
```

```
public class TVOnCommand implements
Command {
        TV tv;
        public TVOnCommand(TV tv){
                this.tv = tv;
        }
        public void execute() {
                tv.on();
        }
        public void undo() {
                tv.off()
        }
}
```

```
public class TVOffCommand implements
Command {
        TV tv;
        public TVOffCommand(TV tv){
                this.tv = tv;
        }
        public void execute() {
                tv.off();
        }
        public void undo() {
                //undo the off
                tv.on()
        }
}
```

Invoker

```
public class SimpleRemote {
        Command onButton;
        Command offButton;
        Command undoCommand;
        public SimpleRemote() {}
        public void setOnCommand(Command command) {
                onButton = command;
        }
        public void setOffCommand(Command command) {
                offButton = command;
        }
        // The remote doesn't care what device it's turning on, it just issues the command
        // set the undo command object to the last object called
        public void buttonOnWasPressed() {
                onButton.execute();
                undoCommand = onButton;
        }
        public void buttonOffWasPressed() {
                offButton.execute();
                undoCommand = offButton;
        }
        public void buttonUndoWasPressed() {
                undoCommand.undo();
        }
}
```

Client

```
public class RemoteTest {

        public static void main(String[] args) {
                SimpleRemote remote = new SimpleRemote();
                TV tv = new TV();
                TVOnCommand tvOn = new TVOnCommand(tv);
                TVOffCommand tvOff = new TVOffCommand(tv);

                //program the remote to turn the TV on and off
                remote.setOnCommand(tvOn);
                remote.setOffCommand(tvOff);

                remote.buttonOnWasPressed();
                remote.buttonOffWasPressed();

                //turn the TV back on by undoing the off call
                remote.buttonUndowWasPressed();
        }
}
```

Template method

Behavioral Patterns

- observer
- decorator
- strategy
- command
- template method

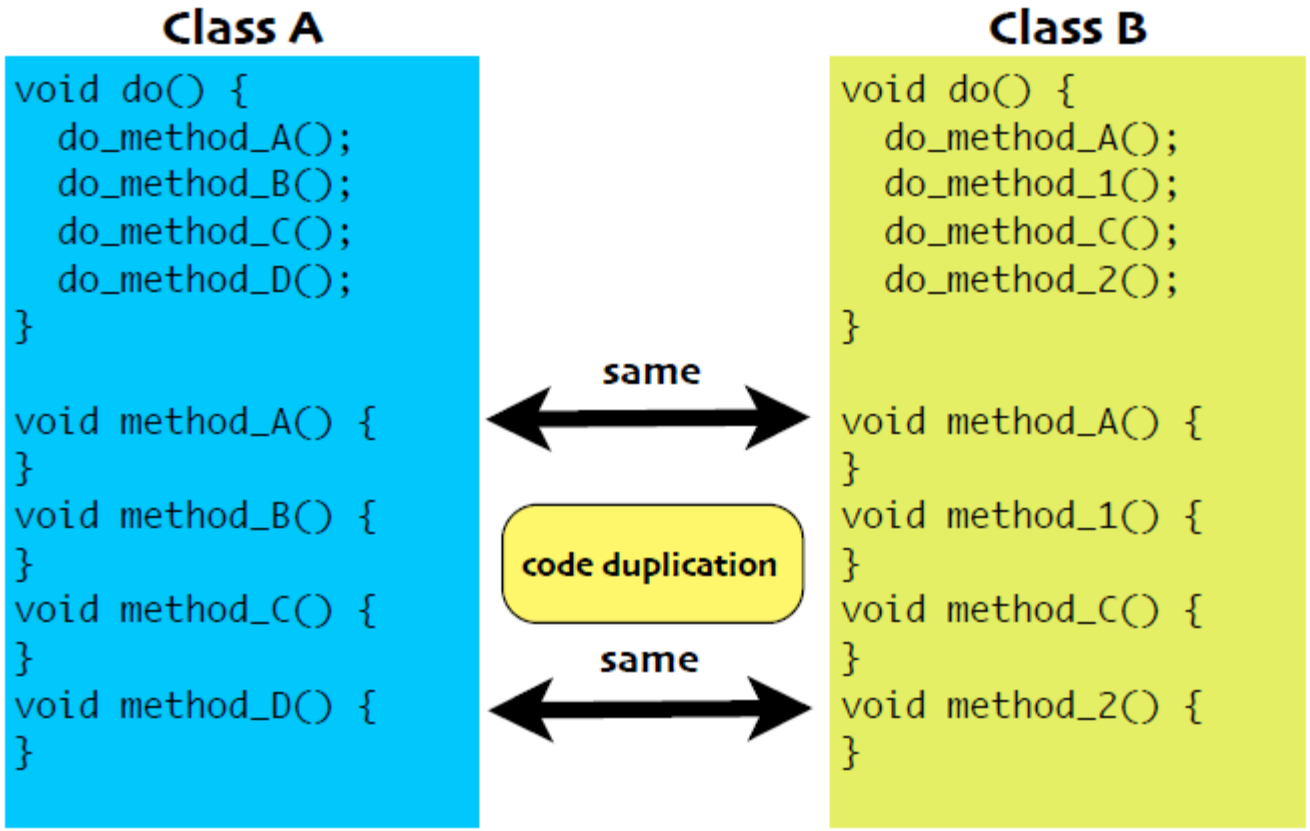
Creational Patterns

- factory method
- abstract factory
- singleton

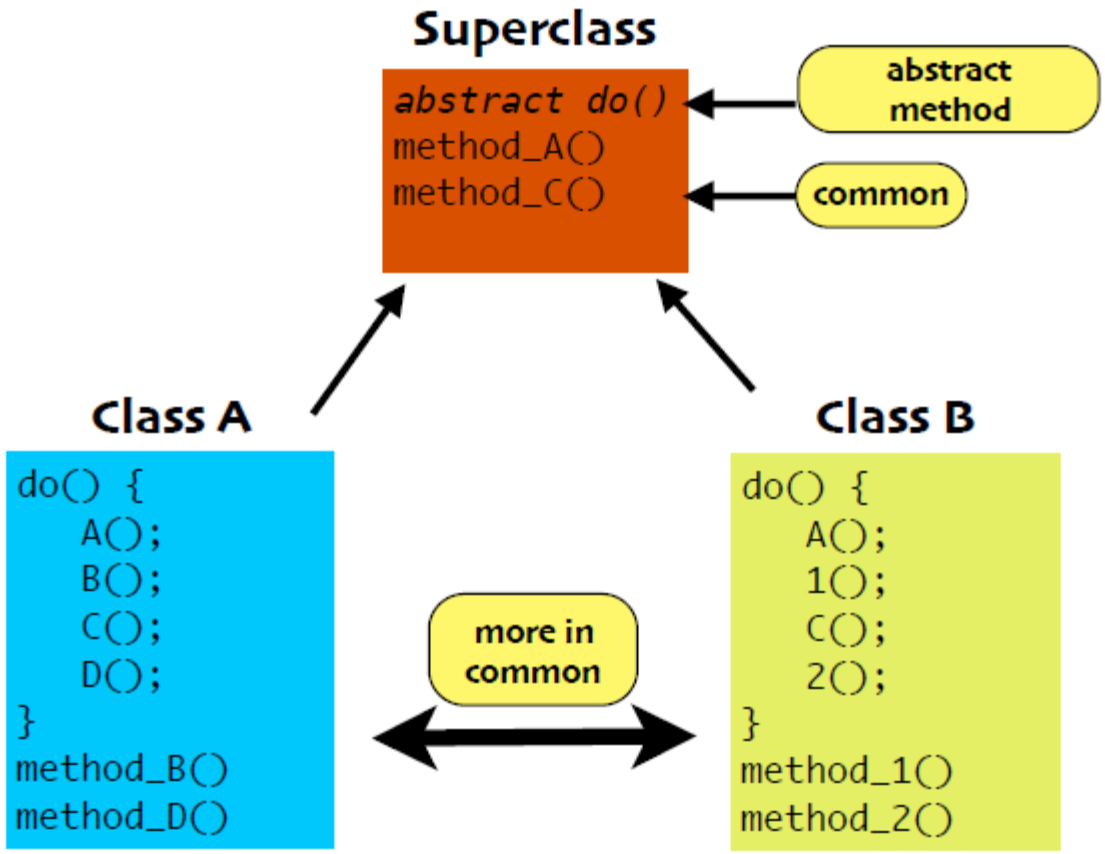
Problem

Duplicated code is difficult to change, maintain or extend

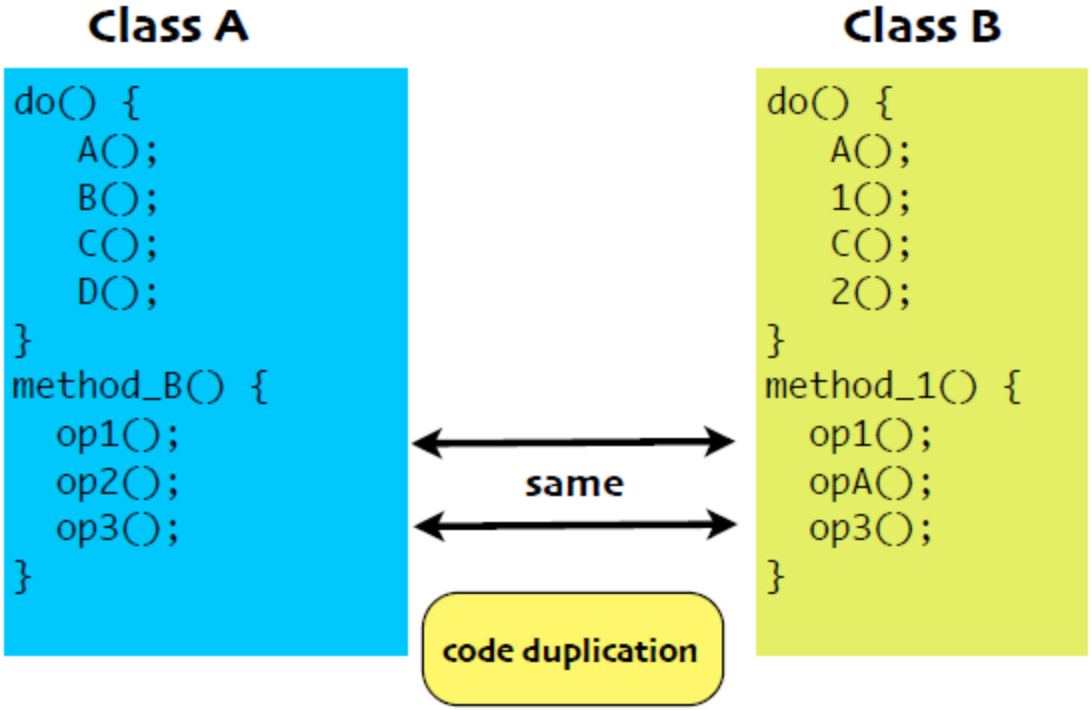
Example


Class A
void do() {
do_method_A();
do_method_B();
do_method_C();
do_method_D();
}
void method_A() {
}
void method_B() {
}
void method_C() {
}
void method_D() {
}
same
code duplication
same
Class B
void do() {
do_method_A();
do_method_1();
do_method_C();
do_method_2();
}
void method_A() {
}
void method_1() {
}
void method_C() {
}
void method_2() {
}

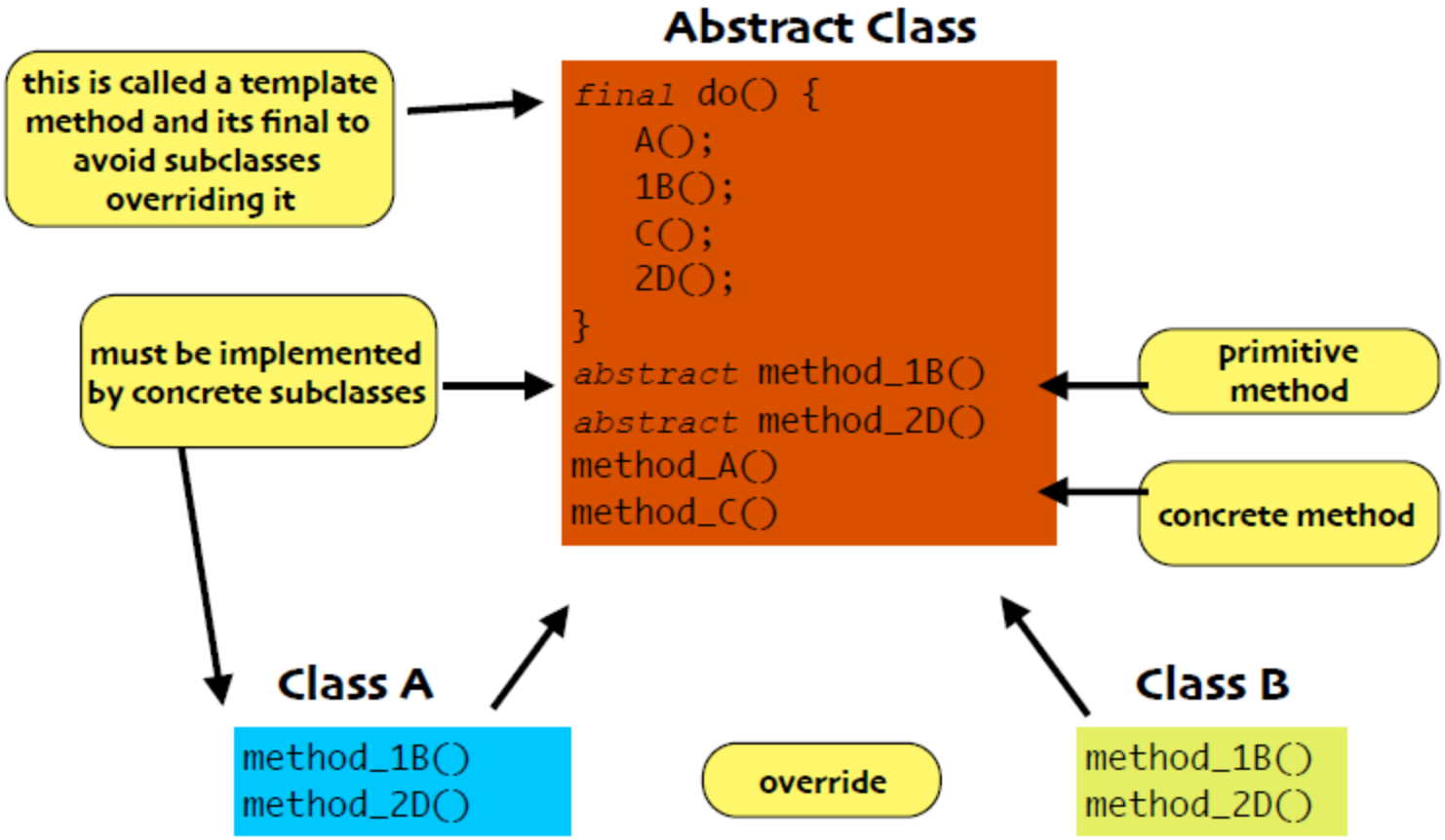
Remove Redundancy



Remove Redundancy



Define Template Method



Template Method

The Template Method Pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses.

Code

```
public class Coffee {

  void prepareRecipe() {
    boilWater();
    brewCoffeeGrinds();
    pourInCup();
    addSugarAndMilk();
  }

  public void boilWater() {
    System.out.println("Boiling water");
  }

  public void brewCoffeeGrinds() {
    System.out.println("Dripping Coffee through filter");
  }

  public void pourInCup() {
    System.out.println("Pouring into cup");
  }

  public void addSugarAndMilk() {
    System.out.println("Adding Sugar and Milk");
  }

}
```

```
public class Tea {

  void prepareRecipe() {
    boilWater();
    steepTeaBag();
    pourInCup();
    addLemon();
  }

  public void boilWater() {
    System.out.println("Boiling water");
  }

  public void steepTeaBag() {
    System.out.println("Steeping the tea");
  }

  public void addLemon() {
    System.out.println("Adding Lemon");
  }

  public void pourInCup() {
    System.out.println("Pouring into cup");
  }

}
```

Code

```
abstract class AbstractClass {
    /* A template method : */
    final void TemplateMethod() {
        primitiveOperation1();
        primitiveOperation2();
        concreteOperation();
    }
    abstract void primitiveOperation1();
    abstract void primitiveOperation2();
    final void concreteOperation() {
     doSomething();
    }
}
```

generic

must be implemented by concrete subclasses

generic

Code

```
public abstract class CaffeineBeverage {
  final void prepareRecipe() {
    boilWater();
    brew();
    pourInCup();
    addCondiments();
  }
  abstract void brew();
  abstract void addCondiments();
  void boilWater() {
    System.out.println("Boiling water");
  }
  void pourInCup() {
    System.out.println("Pouring into cup");
  }
}
```

```
public class Coffee extends CaffeineBeverage {

  public void brew() {
    System.out.println("Dripping Coffee through filter");
  }

  public void addCondiments() {
    System.out.println("Adding Sugar and Milk");
  }

}
```

```
public class Tea extends CaffeineBeverage {

  public void brew() {
    System.out.println("Steeping the tea");
  }

  public void addCondiments() {
    System.out.println("Adding Lemon");
  }

}
```

Add a Hook

```
abstract class AbstractClass {
    /* A template method : */
    final void TemplateMethod() {
        primitiveOperation1();
        primitiveOperation2();
        concreteOperation();
        hook();
    }
    abstract void primitiveOperation1();
    abstract void primitiveOperation2();
    final void concreteOperation() {
     doSomething();
    }
    void hook() { }
}
```

must implement

optional

useful for logging or whatever

stub

in the concrete class you could do a test to decide what the hook should do

Code

```
public abstract class CaffeineBeverageWithHook {

  void prepareRecipe() {
    boilWater();
    brew();
    pourInCup();
    if (customerWantsCondiments()) {
      addCondiments();
    }
  }

  abstract void brew();

  abstract void addCondiments();

  void boilWater() {
    System.out.println("Boiling water");
  }

  void pourInCup() {
    System.out.println("Pouring into cup");
  }

  boolean customerWantsCondiments() {
    return true;
  }

}
```

prepareRecipe() altered to have a hook method: customerWantsCondiments()

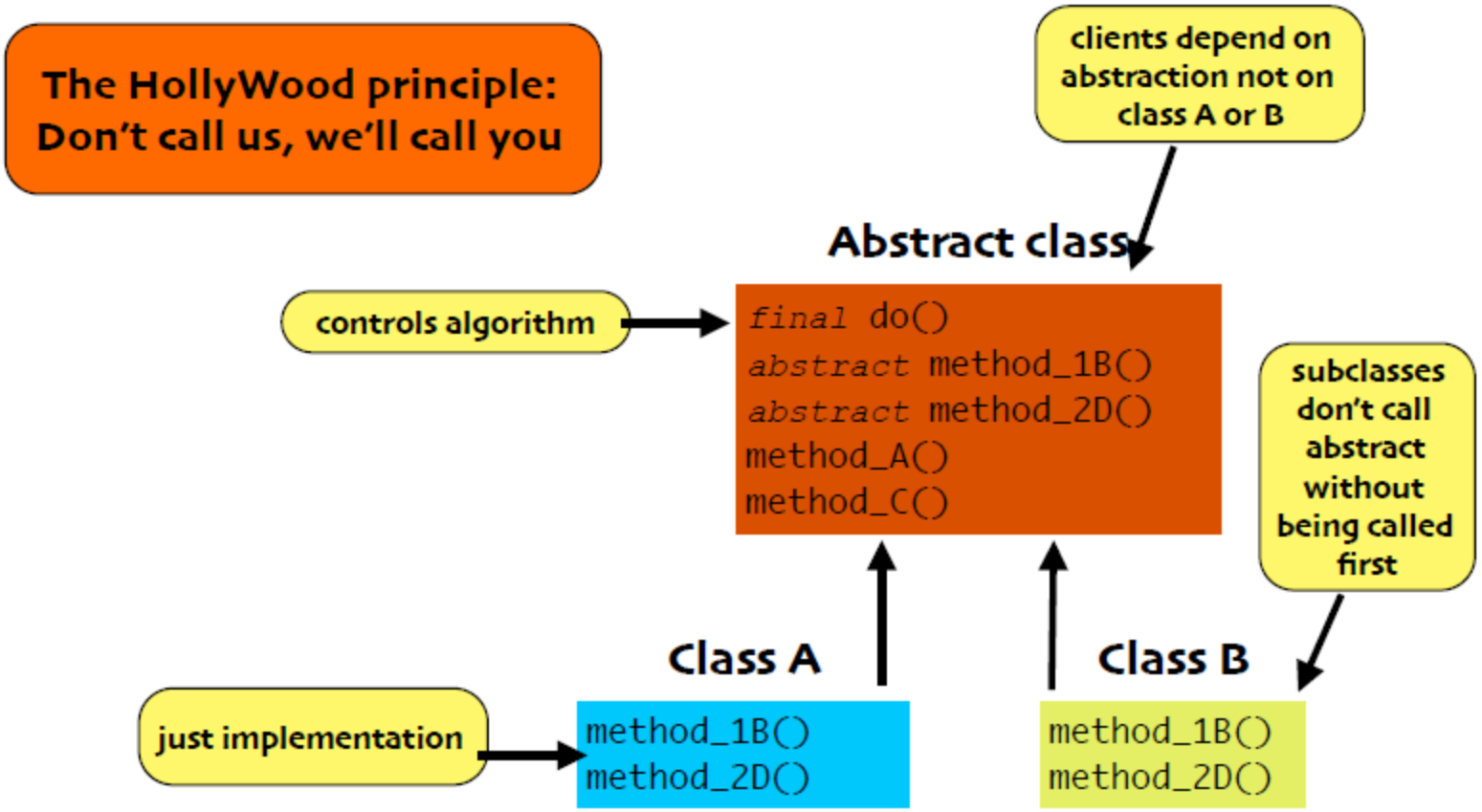
This method provides a method body that subclasses can override

To make the distinction between hook and non-hook methods more clear, you can add the “final” keyword to all concrete methods that you don’t want subclasses to touch

Code

```
import java.io.*;
public class CoffeeWithHook extends CaffeineBeverageWithHook {
  public void brew() {
    System.out.println("Dripping Coffee through filter");
  }
  public void addCondiments() {
    System.out.println("Adding Sugar and Milk");
  }
  public boolean customerWantsCondiments() {
    String answer = getUserInput();
    if (answer.toLowerCase().startsWith("y")) {
    return true;
    } else {
    return false;
    } }
  private String getUserInput() {
    String answer = null;
    System.out.print("Would you like milk and sugar with your coffee (y/n)? ");
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    try {
      answer = in.readLine();
    } catch (IOException ioe) {
      System.err.println("IO error trying to read your answer");
    }
    if (answer == null) {
      return "no";
    }
      return answer;
  } }
```

Design Principle



Hollywood Principle

"Don't call us, we'll call you"

When we design with the template method pattern, we are telling subclasses "Don't call us, we'll call you"

The template method lives in a high-level class and invokes methods that live in its subclasses:

The CaffeineBeverage has control over the algorithm for the recipe. It calls on subclasses only when they are needed for an implementation of a method.

Template Methods in the Wild

Template Method is used a lot since it's a great design tool for creating **frameworks**

- the framework specifies how something should be done with a template method
- that method invokes **abstract hook methods** that allow client-specific subclasses to "hook into" the framework and take advantage of its services

Examples in the Java API

- Sorting using compareTo() method
- Frames in Swing
- Applets

Template Method vs. Strategy

Both Template Method and Strategy deal with the **encapsulation of algorithms**

- **Template Method** focuses **encapsulation on the steps of the algorithm**
- **Strategy** focuses on **encapsulating entire algorithms**
- You can use both patterns at the same time if you want

Template Method vs. Strategy

Template Method encapsulate the details of algorithms using **inheritance**

- **Factory Method** can now be seen as a specialization of the Template Method pattern

In contrast, **Strategy** does a similar thing but uses **composition/delegation**

Template Method vs. Strategy

Because it uses **inheritance**, **Template Method** offers **code reuse benefits** not typically seen with the Strategy pattern

On the other hand, **Strategy** provides **run-time flexibility** because of its use of **composition/delegation**

- You can switch to an entirely different algorithm when using Strategy, something that you can't do when using Template Method

Inclass Exercise

CQRS (Command and Query Responsibility Segregation) Architecture