COMP 2511
Object Oriented Design & Programming

Week 05

© Aarthi Natarajan, 2018

So far,

OO design principles

- Encapsulate what varies
- Program to an interface, not an implementation
- Favour composition over inheritance

Design Patterns

- Strategy and State Patterns

This week,

OO design principles

- Open Closed Principle
- Don’t call, we’ll call you (Hollywood principal)

Design Patterns

- Template Method Pattern (Encapsulating algorithms)

© Aarthi Natarajan, 2018

Star Buzz Barista Training Manual

Starbuzz Coffee Recipe

- Boil some water
- Brew coffee in boiling water
- Pour coffee in cup
- Add sugar and boil

Starbuzz Tea Recipe

- Boil some water
- Place tea in boiling water
- Pour tea in cup
- Add lemon

The recipe for coffee looks a lot like the recipe for tea, doesn't it?

© Aarthi Natarajan, 2018

Code for Star Buzz Barista Recipe

```java
public class Coffee {
    // Each step in the recipe implemented as a separate method
    public void prepareRecipe() {
        boilWater();
        brewCoffee();
        pourInCup();
        addSugarInMilk();
    }
    // Each method implements one step of the algorithm
    private void boilWater() {
        System.out.println("Boiling Water");
    }
    private void brewCoffee() {
        System.out.println("Brew Coffee");
    }
    private void pourInCup() {
        System.out.println("Pouring in cup");
    }
    private void addSugarInMilk() {
        System.out.println("Adding sugar and milk");
    }
}
```

```java
public class Tea {
    // Each step in the recipe implemented as a separate method
    public void prepareRecipe() {
        boilWater();
        steepTea();
        pourInCup();
        addLemon();
    }
    // Each method implements one step of the algorithm
    private void boilWater() {
        System.out.println("Boiling Water");
    }
    private void steepTea() {
        System.out.println("Steeping the tea");
    }
    private void pourInCup() {
        System.out.println("Pouring in cup");
    }
    private void addLemon() {
        System.out.println("Adding lemon");
    }
}
```

`prepareRecipe()` in both classes look similar. The second and third steps are different

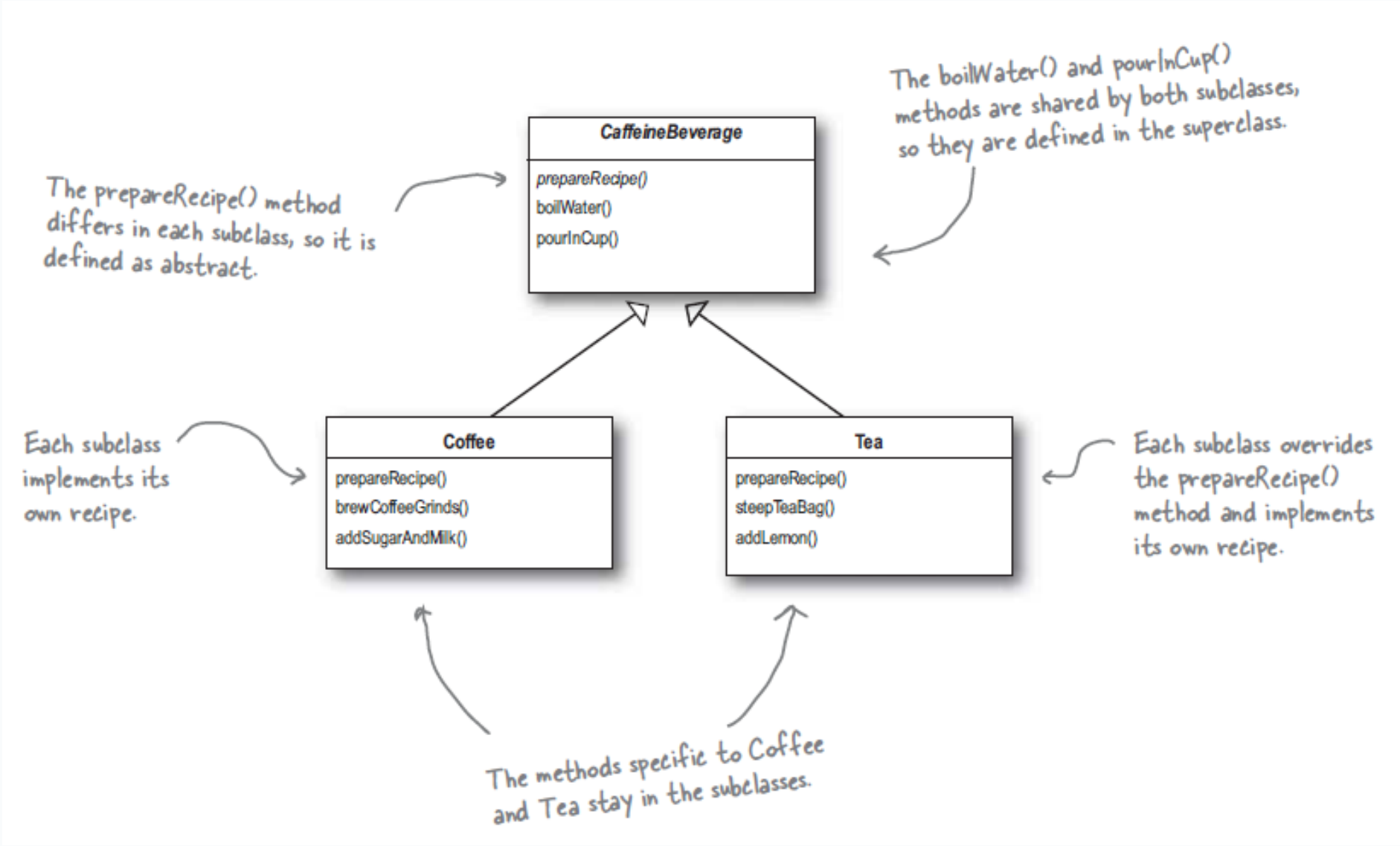
`brewCoffee()` and `addSugarInMilk()` are specialised to Coffee

`steepTea()` and `addLemon()` are specialised to Tea

Code (Behaviour) duplication – a code smell (breeding ground for bugs)!

© Aarthi Natarajan, 2018

Solution 1…



© Head First Design Patterns By E Freeman et al
© Aarthi Natarajan, 2018

Can we improve the design ?

Coffee Recipe

1. *Boil the water*
2. *Brew coffee in boiling water*
3. *Pour coffee in cup*
4. *Add sugar, milk*

Beverage Recipe

1. *Boil the water*
2. *Use hot water to extract tea or coffee*
3. *Pour beverage in cup*
4. *Add condiments*

Tea Recipe

1. *Boil the water*
2. *Steep tea in boiling water*
3. *Pour tea in cup*
4. *Add lemon*

(1) and (2) are already abstracted into based class

(3) and (4) are not abstracted but are the same, just apply to different beverages

© Aarthi Natarajan, 2018

Can we improve the design ?

- Coffee uses `brewCoffeeGrinds()` and `addSugarAndMilk()`
- Tea uses `steepTeaBags()` and `addLemon()`

Coffee

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

Tea

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

```
void prepareRecipe() {
    boilWater();
    brew();
    pourInCup();
    addCondiments();
}
```

© Aarthi Natarajan, 2018

Can we improve the design ?

Now, modify `prepareRecipe()` to fit into the code

CaffeineBeverage is abstract, just like in the class design.

```
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");
    }
}
```

Now, the same prepareRecipe() method will be used to make both Tea and Coffee. prepareRecipe() is declared final because we don't want our subclasses to be able to override this method and change the recipe! We've generalized steps 2 and 4 to brew() the beverage and addCondiments().

Because Coffee and Tea handle these methods in different ways, they're going to have to be declared as abstract. Let the subclasses worry about that stuff!

Remember, we moved these into the CaffeineBeverage class (back in our class diagram).

© Aarthi Natarajan, 2018

Now, fix our Coffee and Tea classes

Now, modify `Coffee` and `Tea` classes

```
public class Tea extends CaffeineBeverage {
    public void brew() {
        System.out.println("Steeping the tea");
    }
    public void addCondiments() {
        System.out.println("Adding Lemon");
    }
}
```

As in our design, Tea and Coffee now extend CaffeineBeverage.

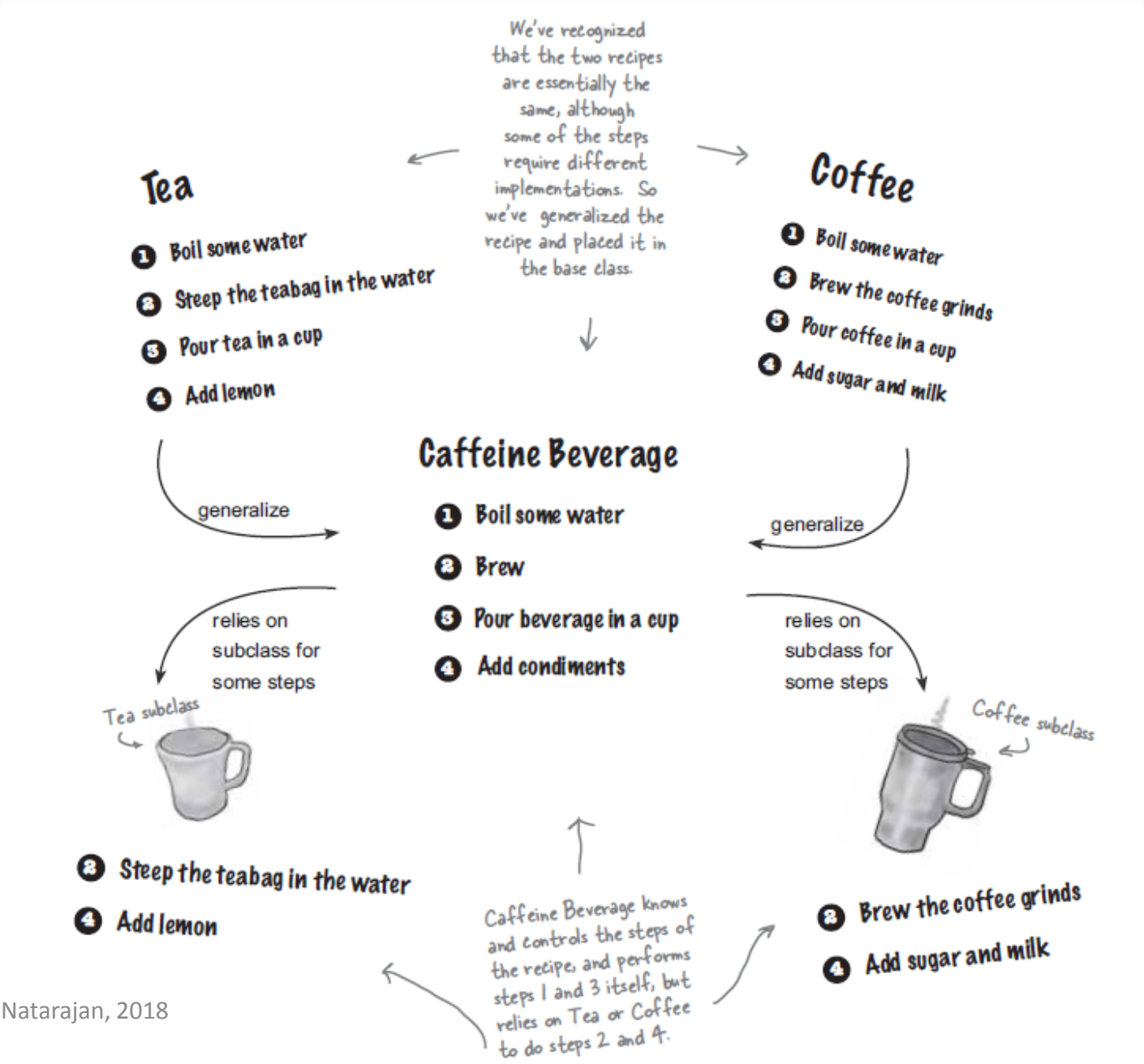
Tea needs to define brew() and addCondiments() – the two abstract methods from Beverage.

Same for Coffee, except Coffee deals with coffee, and sugar and milk instead of tea bags and lemon.

```
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");
    }
}
```

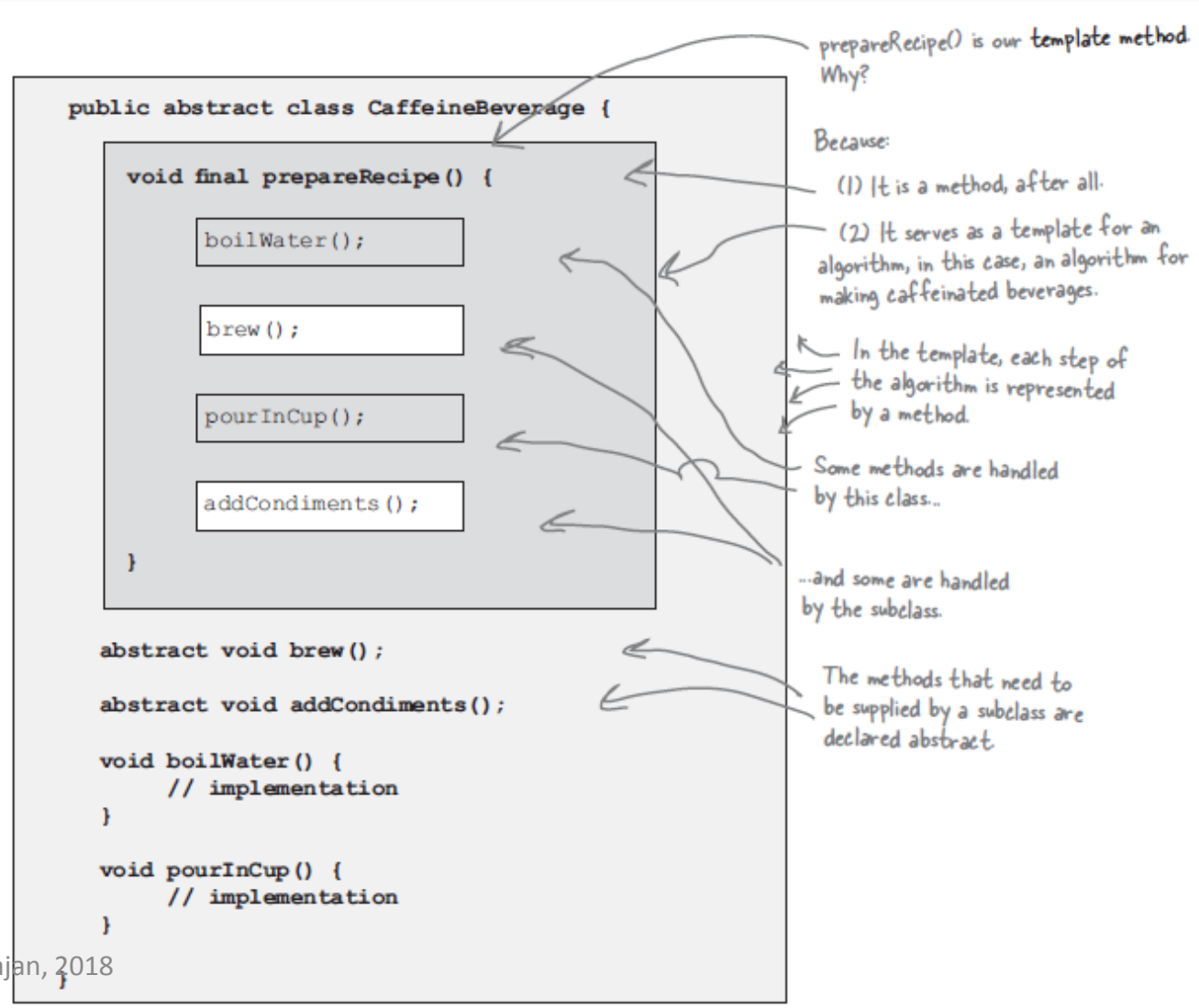
© Aarthi Natarajan, 2018

What have we done ?



© Aarthi Natarajan, 2018

The `prepareRecipe()` method defines a template for an algorithm, in this case an algorithm for making caffeinated beverages

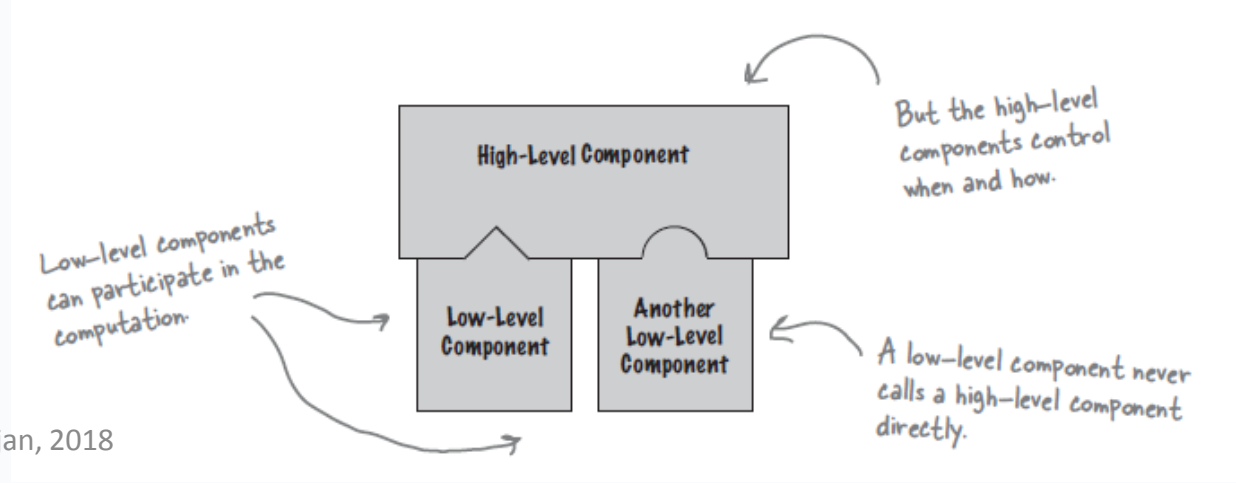


© Aarthi Natarajan, 2018

Design Principle #5: The Hollywood Principle

Don't call us, we'll call you

- Motivation
  - Provides us a way to reduce software rot
  - The high-level components (CaffeineBeverage) give the low-level components (Coffee, Tea) a "don't call us, we'll call you' treatment, i.e., allow low-level components to hook themselves into a system, but the high-level components decide "when" and "how" they are needed



© Aarthi Natarajan, 2018

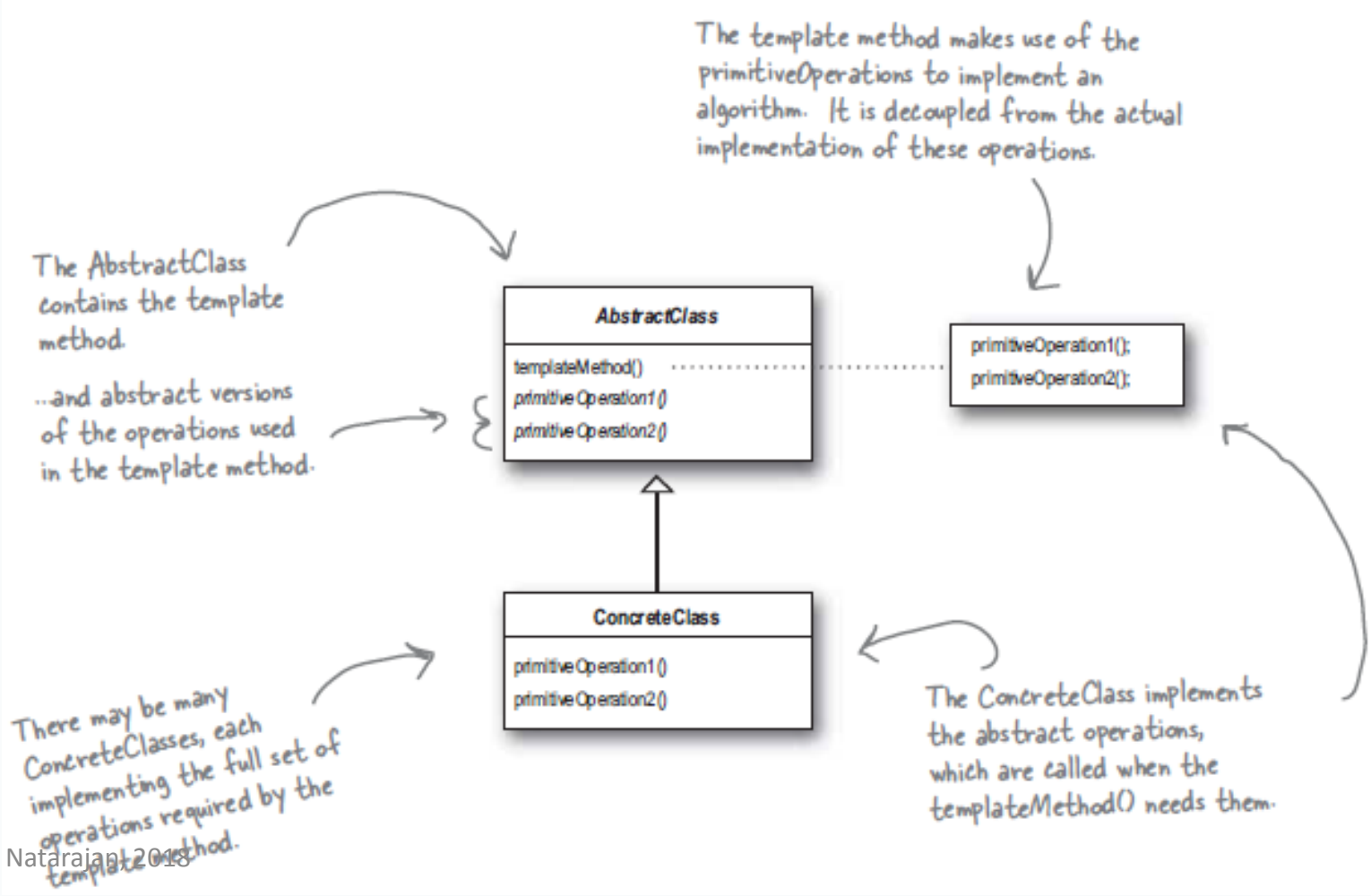
Pattern #3: Template Method Pattern

- Motivation
  - Two different components exhibit significant similarity in behaviour, but demonstrate no re-use of common implementation.
  - Need a way to remove the duplicated effort following a change to the common functionality
- Intent
  - The Template Method pattern is a behavioural design pattern that defines the skeleton of an algorithm in a method deferring some steps to sub-classes. The template method lets sub-classes redefine certain steps of an algorithm without changing the algorithm structure

© Aarthi Natarajan, 2018

The Template Method

Design Pattern #3: The `Template Method Pattern` defines the skeleton of an algorithm in a method deferring some steps to sub-classes. The template method lets sub-classes redefine certain steps of an algorithm without changing the algorithm structure



© Aarthi Natarajan, 2013

The Template Method

Before Template Method

- Coffee and Tea control the algorithm
- Code (and hence behaviour) is duplicated across both
- Changes to the algorithm requires duplicated effort
- Design is rigid to add a new caffeine beverage (Design smell – Rigidity
- Knowledge of the algorithm is distributed across multiple classes

After Template Method

- Beverage class controls the algorithm and protects it
- Beverage class maximises code (behaviour) reuse
- Algorithm lives in one place, changes localised to one place
- Provides a flexible framework to add new caffeine beverages
- Knowledge of the algorithm is concentrated in Beverage class that relies on sub-classes to provide complete implementations

© Aarthi Natarajan, 2018

The Template Method

Before Template Method

- Coffee and Tea control the algorithm
- Code (and hence behaviour) is duplicated across both
- Changes to the algorithm requires duplicated effort
- Design is rigid to add a new caffeine beverage (Design smell – Rigidity
- Knowledge of the algorithm is distributed across multiple classes

After Template Method

- Beverage class controls the algorithm and protects it
- Beverage class maximises code (behaviour) reuse
- Algorithm lives in one place, changes localised to one place
- Provides a flexible framework to add new caffeine beverages
- Knowledge of the algorithm is concentrated in Beverage class that relies on sub-classes to provide complete implementations

© Aarthi Natarajan, 2018

Hooked on the Template Method

- A hook is a method in the abstract class, with an empty or default implementation
- A sub-class may or may not override the hook

```
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;
    }
}
```

We've added a little conditional statement that bases its success on a concrete method, customerWantsCondiments(). If the customer WANTS a condiments, only then do we call addCondiments().

Here we've defined a method with a (mostly) empty default implementation. This method just returns true and does nothing else.

This is a **hook** because the subclass can override this method, but doesn't have to.

© Aarthi Natarajan, 2018

Using the hook

- A hook gives subclasses the ability to "hook into" the algorithm at various points or a sub-class may choose "not to"

```
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;
    }
}
```

Here's where you override the hook and provide your own functionality.

Get the user's input on the condiment decision and return true or false depending on the input.

This code asks the user if he'd like milk and sugar and gets his input from the command line.

© Aarthi Natarajan, 2018

The Open Closed Principle

Design Principle #4: Classes should be open for extension but closed for modification

- Closed for modification implies that new changes must be implemented by new code instead of altering existing code
- Classes must be closed for modification
  - This reduces the possibility of breaking existing tried and tested code
- Classes should be open for extension
  - Allow classes to be easily extended to incorporate new behaviour
- OCP promotes designs that are resilient to change but flexible to take on new requirements

© Aarthi Natarajan, 2018

The Open Closed Principle

How does the strategy, state and template patterns support OCP?

- The strategy and state pattern are open for extension, by allowing a new strategy or state object to be created that implements an existing interface
- Similarly, the template method enables new sub classes to provide variations of the steps of the algorithm
- However, be careful when choosing the areas of code that need to be extended; applying the OCP EVERYWHERE is wasteful, unnecessary, and can lead to complex, hard to understand code

© Aarthi Natarajan, 2018

Design Toolbox

OO Basics

- *Abstraction*
- *Encapsulation*
- *Inheritance*
- *Polymorphism*

OO Principles

- *Principle of least knowledge – talk only to your friends*
- *Encapsulate what varies*
- *Favour composition over inheritance*
- *Program to an interface, not an implementation*
- ***Classes should be open for extension and closed for modification***
- ***Don't call us, we'll call you***

OO Patterns

- *Strategy*
- *State*
- ***Template Method***

© Aarthi Natarajan, 2018

COMP 2511
Object Oriented Design & Programming

Design By Contract

© Aarthi Natarajan, 2018

Defensive Programming vs Design By Contract

Defensive Programming

- Expect the unexpected:

  " The whole point of defensive programming is guarding against errors you don't expect." [Steve McConnell, Code Complete]

- Promotes putting checks in every module to detect unexpected situations
  - Results in redundant checks (for both caller and callee may check the same condition)
  - Many checks makes the software more complex and harder to maintain

Design by Contract

- Assignment of responsibilities was clear and was integrated as part of the module interface
  - prevents redundant checks
  - easier to maintain

© Aarthi Rajan, 2018

Design By Contract

What should happen if a caller passes a negative amount to deposit?

```
public class BankAccount {
    public void deposit(float amt) {
        balance += amt;
    }
}
```

What should happen if a caller attempts to remove a message from an empty queue?

```
public class MessageQueue {
    public void add(Message someMsg) { ... }
    /**
       Remove message at head of the queue
    */
    public void remove() { ... }
}

public Message remove() {
                return elements. remove(0); }
```

© Aarthi Natarajan, 2018

Design By Contract

- A class that provides a service and its caller should have a **formal contract**

- The **contract** is enforced between the service-provider and caller through **pre-conditions**, **post-conditions** and **class invariants**

© Aarthi Natarajan, 2018

Pre-Conditions

A **pre-condition** is a condition that must be fulfilled and hence true before the service provider promises to act its part

- Pre-conditions do not have to be handled by the component provider
- If pre-conditions hold, error checking is redundant
- Specified in javadoc along with associated parameters in their **@param** tags or with custom tags e.g., **@pre**

```
// Here, deposit makes no promises  to do anything sensible
// when you pass in a negative amount

public class BankAccount {
/**
 * deposit amount into a bank account
 * @param an amount > 0 to be deposited in to the account
*/
public void deposit(float amt)  {
        balance += amt; }
}
```

© Aarthi Natarajan, 2018

Pre-condition (another example)

```
/**
 *  Remove message at head
 *  @pre size() > 0
 */
public Message remove() {
                return elements. remove(0); }
```

- The method ***remove*** **makes** ***no promises*** to do anything sensible when you call ***remove*** on an empty queue.
- Cost of violating the pre-condition is high. Here, an *IndexOutOfBounds* exception is thrown

© Aarthi Natarajan, 2018

Post-condition

- A **post-condition** is a logical condition that the service provider guarantees to hold upon completion of the method, provided that the **pre-condition** was fulfilled and is true when the method was called.
- Then the service-provider guarantees expected behaviour
- e.g., the ***deposit*** method promises that after the deposit, balance is incremented by the amount deposited.

```
public class BankAccount { ...
/**
 * deposit amount into a bank account
 * @param an amount to be deposited into the account
 * @pre amount > 0
 * @post balance = balance + amount
 */
public void deposit(float amt)  {
        balance += amt;
}
```

© Aarthi Natarajan, 2018

Post-Condition (another example)

- The *add()* method promises has a useful post-condition that after adding an element, size() > 0.
- This condition is useful because it *implies the precondition* of the *remove* method.

```
/**
 *  @post q.size() > 0
*/
public void add() {  ... }

// Any client code is guaranteed that after you add an element, it
is always safe to call remove() e.g.,
q.add(m);

// Post-condition of add: q.size() > 0
// Pre-condition of remove: q.size() > 0

m = q.remove()
```

© Aarthi Natarajan, 2018

Class Invariant

*A **class-invariant*** is a logical condition that holds for the state of an object.

To provide a class invariant you must check that:

- the condition is true after the constructor has completed execution (this guarantees that no invalid objects are created
- the condition is true before and after a method call, but it can be temporarily violated during the execution of a method.

© Aarthi Natarajan, 2018

Class Invariant (example)

```
/*
 * @invariant balance >= 0
 */
public class BankAccount   {

   private float balance = 0;
   /**
     Constructor - constructs a
     bank account.
     @param initial amount in the
            account
     @pre amount > 0
   **/

   public BankAccount(int amount) {
        this.balance = amount;
    }
```

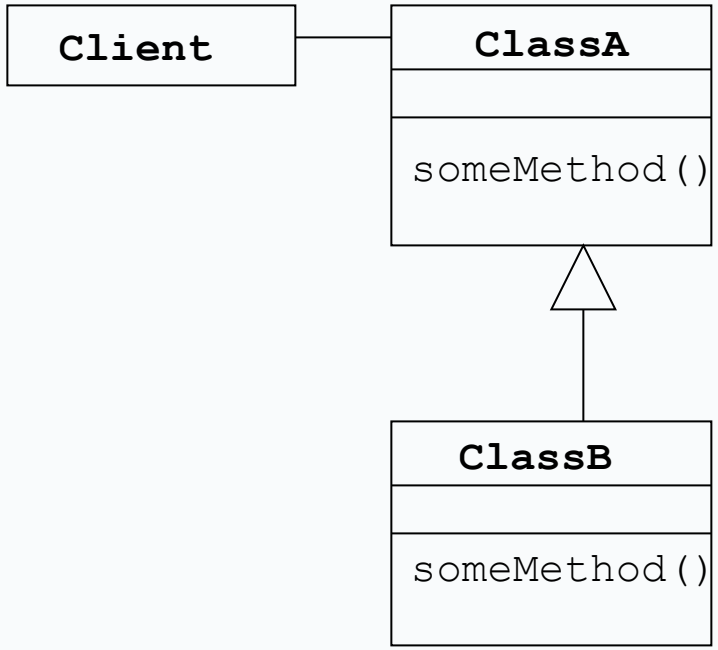
```
/**
 * @param amount to be deposited.
 * @pre amount > 0
 * @post balance is incremented by the
         amount deposited
 */
public void deposit(float amount) {
          this.balance += amount;
}

/**
 * @param amount to be withdrawn
 * @pre amount > 0
 */
public boolean withdraw(float amount) {
   if (amount <= this.balance) {
         this.balance -= amount;
         return true;
   }
   else {   return false;
   }
```

© Aarthi Natarajan, 2018

Inheritance: Pre-conditions

The precondition of the `ClassB.someMethod` must not be stronger than the precondition of the `ClassA.someMethod`

The code for `ClassB` may have been written after `Client` was written, so `Client` has no way of knowing its contractual requirements for `ClassB`

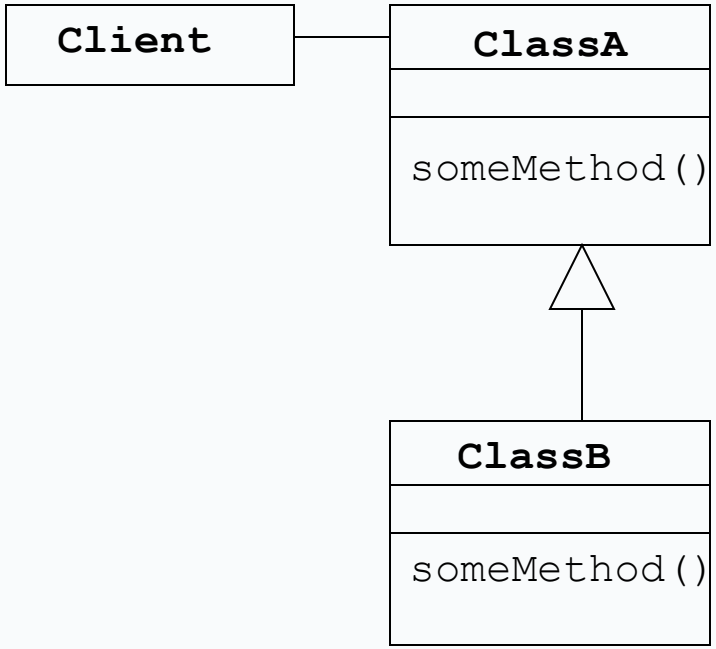


© Aarthi Natarajan, 2018

Inheritance: Post-conditions

- The post-condition of the `ClassB.someMethod` must not be weaker than the postcondition of the `ClassA.someMethod`

- Since `Client` may not have known about `ClassB`, it could have relied on the stronger guarantees provided by the `ClassA.someMethod`



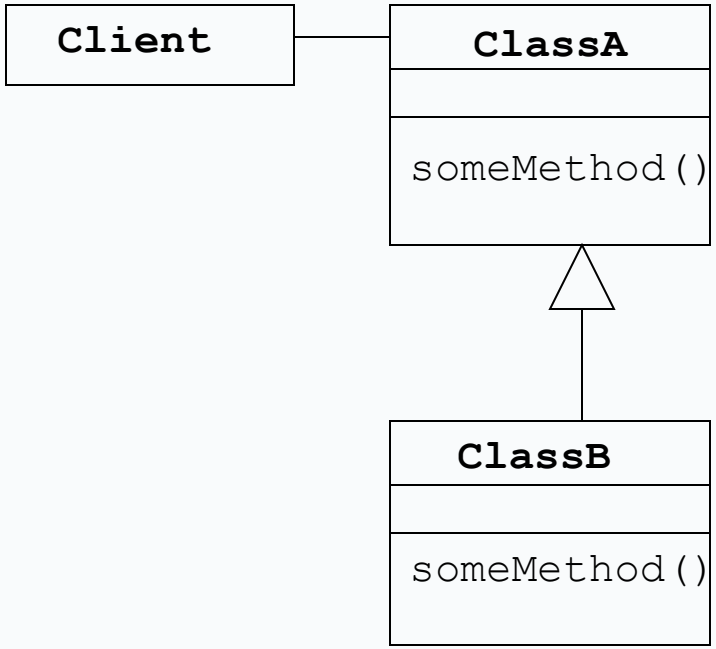
© Aarthi Natarajan, 2018

Inheritance: Invariants

- All class invariants for `ClassA` must be preserved in the sub-type `ClassB`

- If the class invariant for the `ClassB` is weaker than the class invariant for the `ClassA`, then this is not fair to the `Client`

- Since `Client` may not have known about `ClassB`, it could have relied on the stronger guarantees provided by the `ClassA`



© Aarthi Natarajan 2018

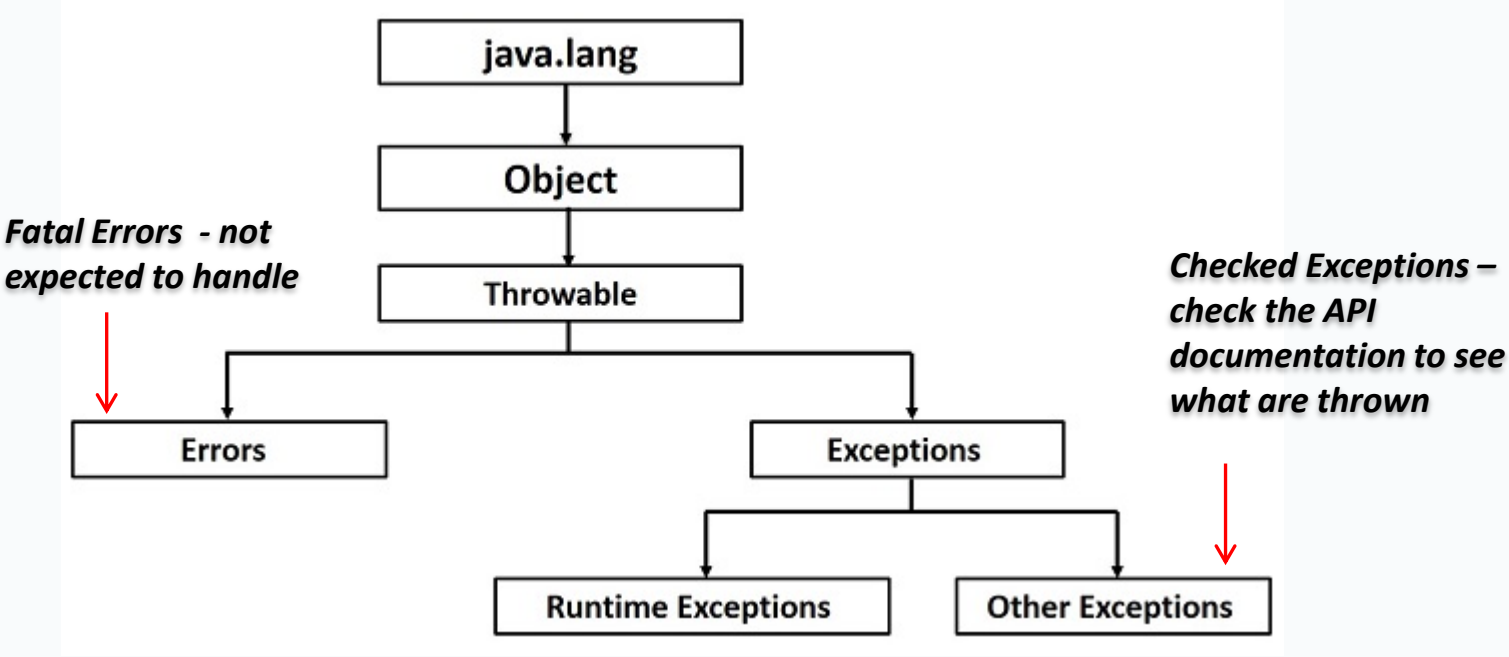
Exceptions

- Are problems that arises during the execution of a program and disrupts the normal flow of the program e.g., a user has entered invalid data, a file does not exist, JVM has run out of memory
- Can be fatal or gracefully handled

© Aarthi Natarajan, 2018

Exception Handling in Java

- Exception handled through a *try-catch* mechanism
- If exception is not handled in the try-catch block, it is propagated to the method caller



© Aarthi Natarajan, 2018

Exceptions in the Contract

- A common strategy for dealing with problem cases is throwing an exception, and the exception can be specified as part of the contract

```
/**
 * Creates a new FileReader,given the name of the file to read from
 * @param  the name of the file to read from
 * @throws FileNotFoundException if the name of the file does not exist, is
a directory rather than a regular file….
*/
public  FileReader(String fileName)  throws FIleNotFoundException {
      ...
}
```

In the example above,

- The constructor has no **pre-condition.** *(fileName must be a valid file is not a pre-condition)*
- Constructor "promises" to throw a *FileNotFoundException* if there is no file with the given name and programmers calling this constructor are entitled to rely on this behaviour.

© Aarthi Natarajan, 2018

Next Week

- More Exception Handling
- Generic Types, Collections
- Iterator, Composite Patterns

© Aarthi Natarajan, 2018