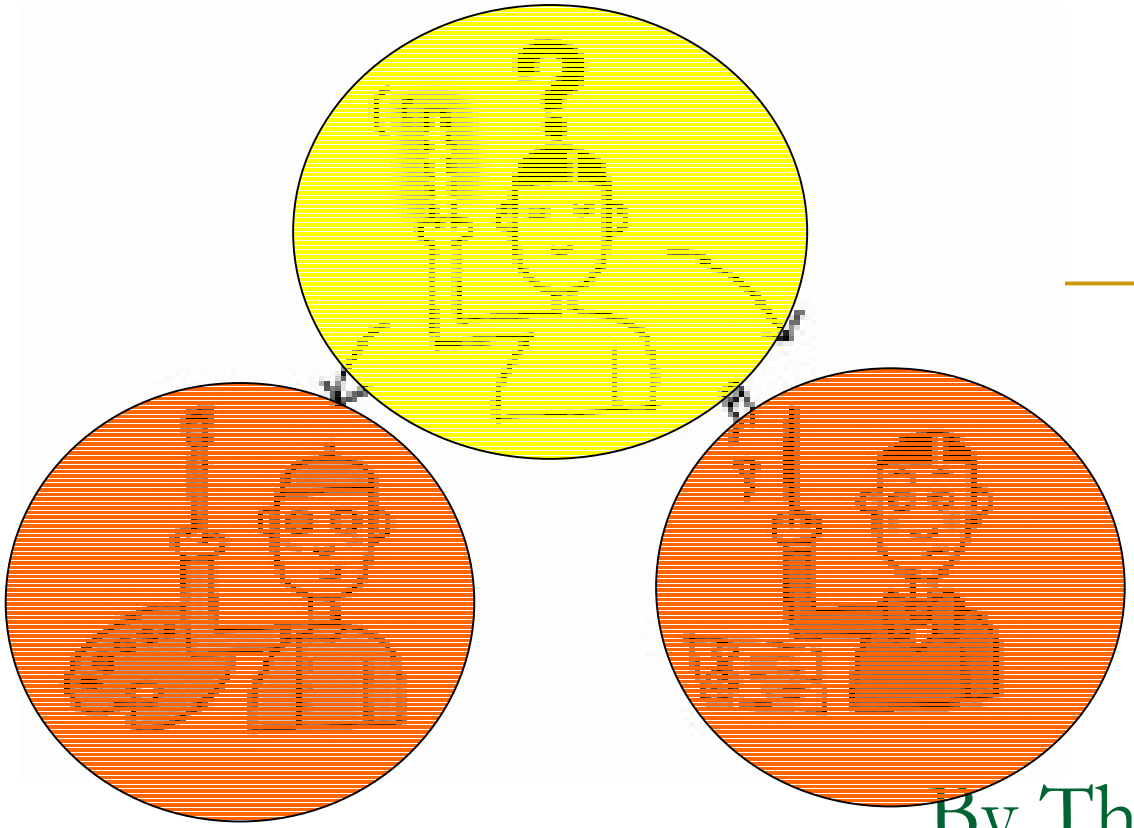
Template Method Pattern

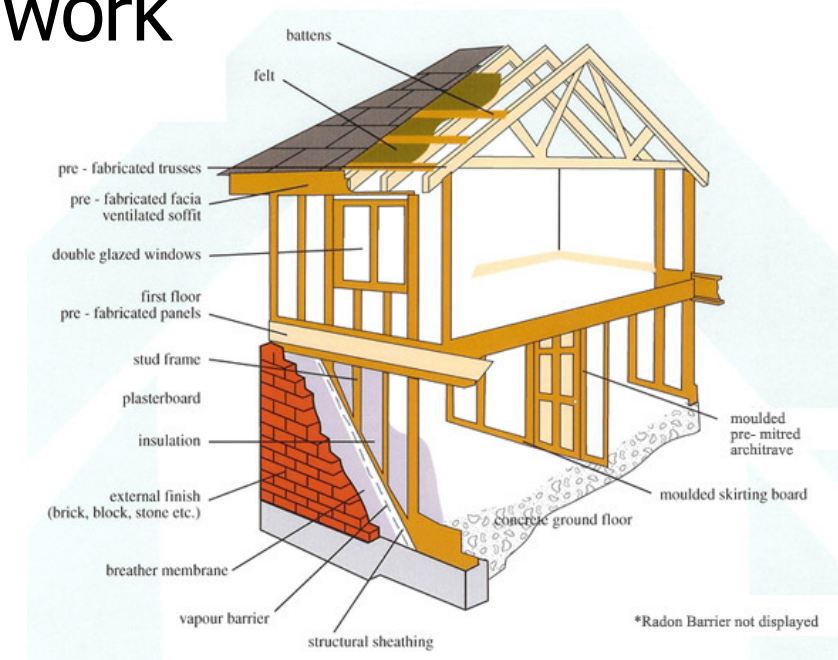


By Thai-Swiss Connections

Pretest : create class diagram

We need to create a house in many styles with this algorithm

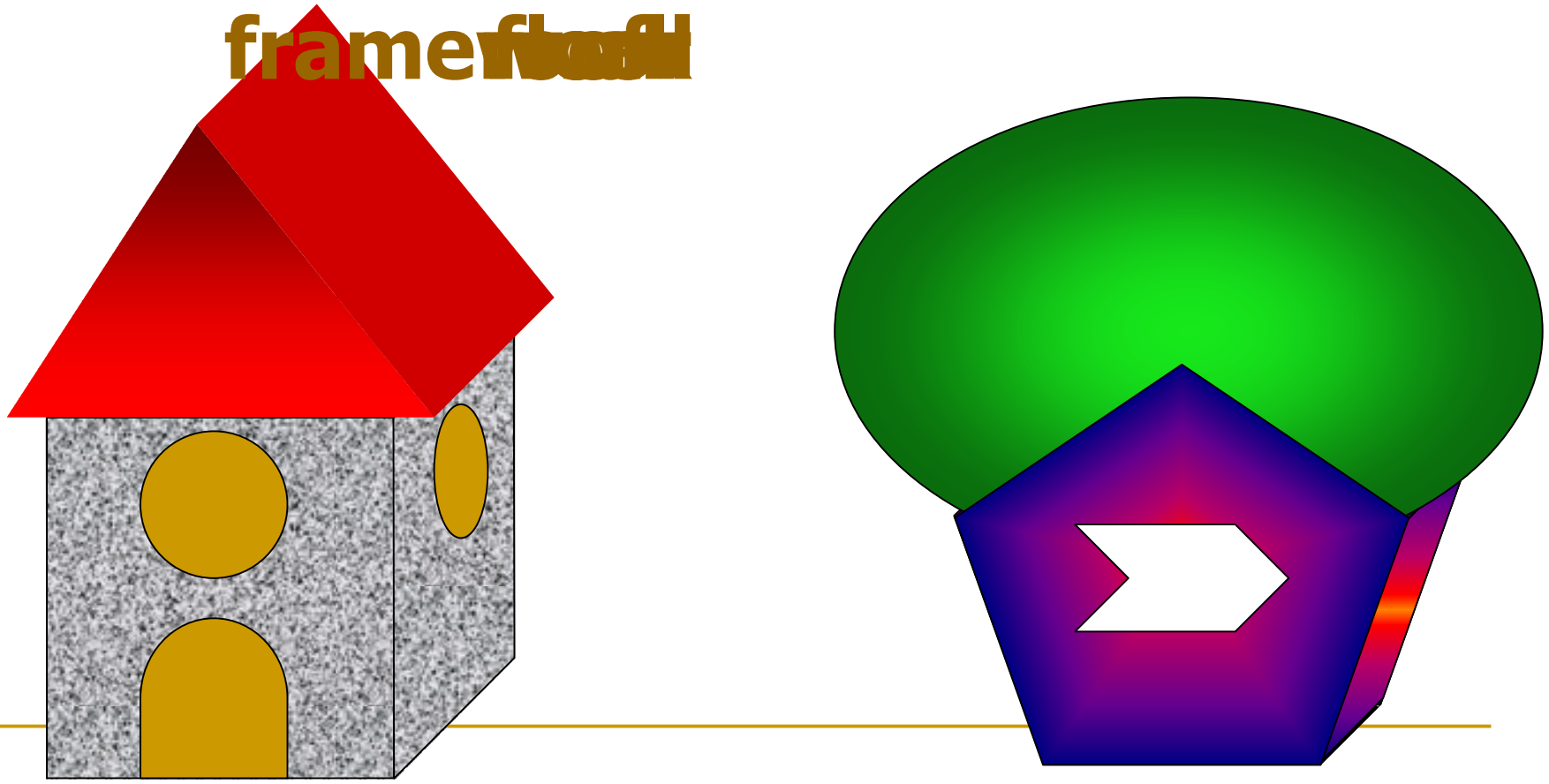
- Create building framework
- Create floor
- Create wall
- Create roof



Notice

- Every home style must follow the algorithm in previous page
- Variety of home style.
- In each step of this algorithm are different in detail for each style

Example



Template Method

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

Introduction

Problem

- In many cases where you have several classes that inherit from the same superclass some of them are likely to implement algorithms that are identical in structure. Refactoring the code inside the algorithm will enable you to move some common code outside the subclasses and into the superclass. The superclass can then be responsible for handling the algorithm and

Template Method

Context

1 A general algorithm can be used in multiple different classes

2 The algorithm can be broken into smaller parts that can be different for each class

3 The executing order of the parts in the algorithm do not depend on the class

but different sub classes can have different functionality after implementing the methods in the super class.

Remind your knowledge

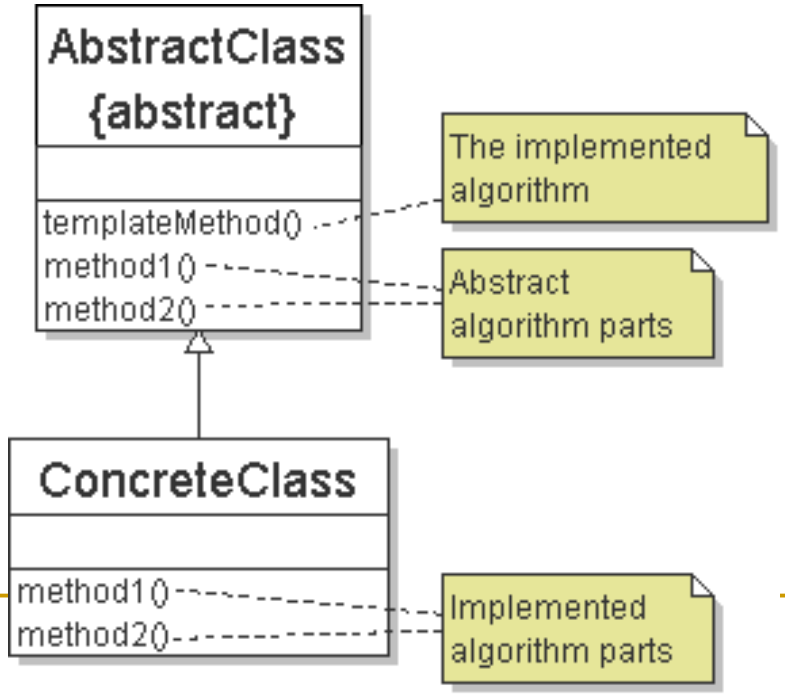
Pattern	**Description**
Template Method	Encapsulate interchangeable behaviors and use delegation to decide which behavior to use
Strategy	
	Subclass decide how to implement steps in an algorithm
Factory Method	
	Subclass decide which concrete classes to create

Solution

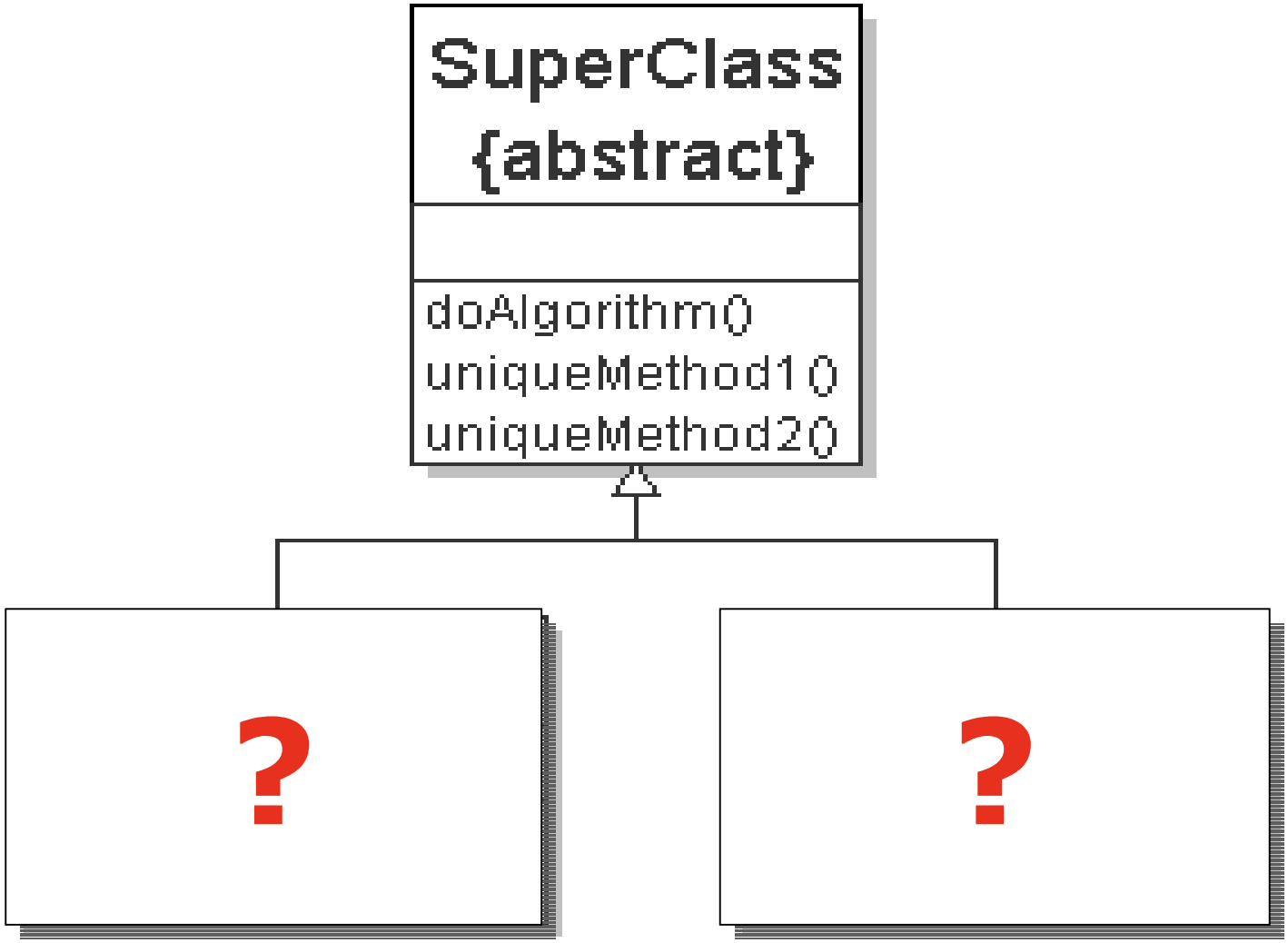
1.Define an abstract class that has a method that executes the algorithm and holds abstract versions of the methods used in the algorithm

2.Implement the algorithm in the abstract class, but do not implement the methods that the algorithms uses

3.Each subclass of the abstract class defines the unimplemented parts of the algorithm, but leaves the algorithm itself untouched



Example Usage - Simple



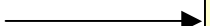
```
class SuperClass{
        public void doAlgorithm() {
                for ( int i = 1; i < 2; i++ ) {
                print( "Loop #" + i );
                uniqueMethod1();
                commonMethod();
                uniqueMehod2();
        }
        private void commonMethod() {
                print( "commonMethod" );
        }
        abstract void uniqueMethod1();
        abstract void uniqueMethod2();
}
```

```
class SubClassOne extends SuperClass {
        public void uniqueMethod1() {
                print( "SubClassOne: uniqueMehod1" );
        }
        public void uniqueMethod1() {
                print( "SubClassOne: uniqueMehod2" );
        }
}
```

```
                print( "SubClassTwo: uniqueMehod2" );
        }
}
```

RUN

```
SuperClass sc = null;
sc = new SubClassOne();
sc.doAlgorithm();
print( "\n" );
sc = new SubClassTwo();
sc.doAlgorithm();
```



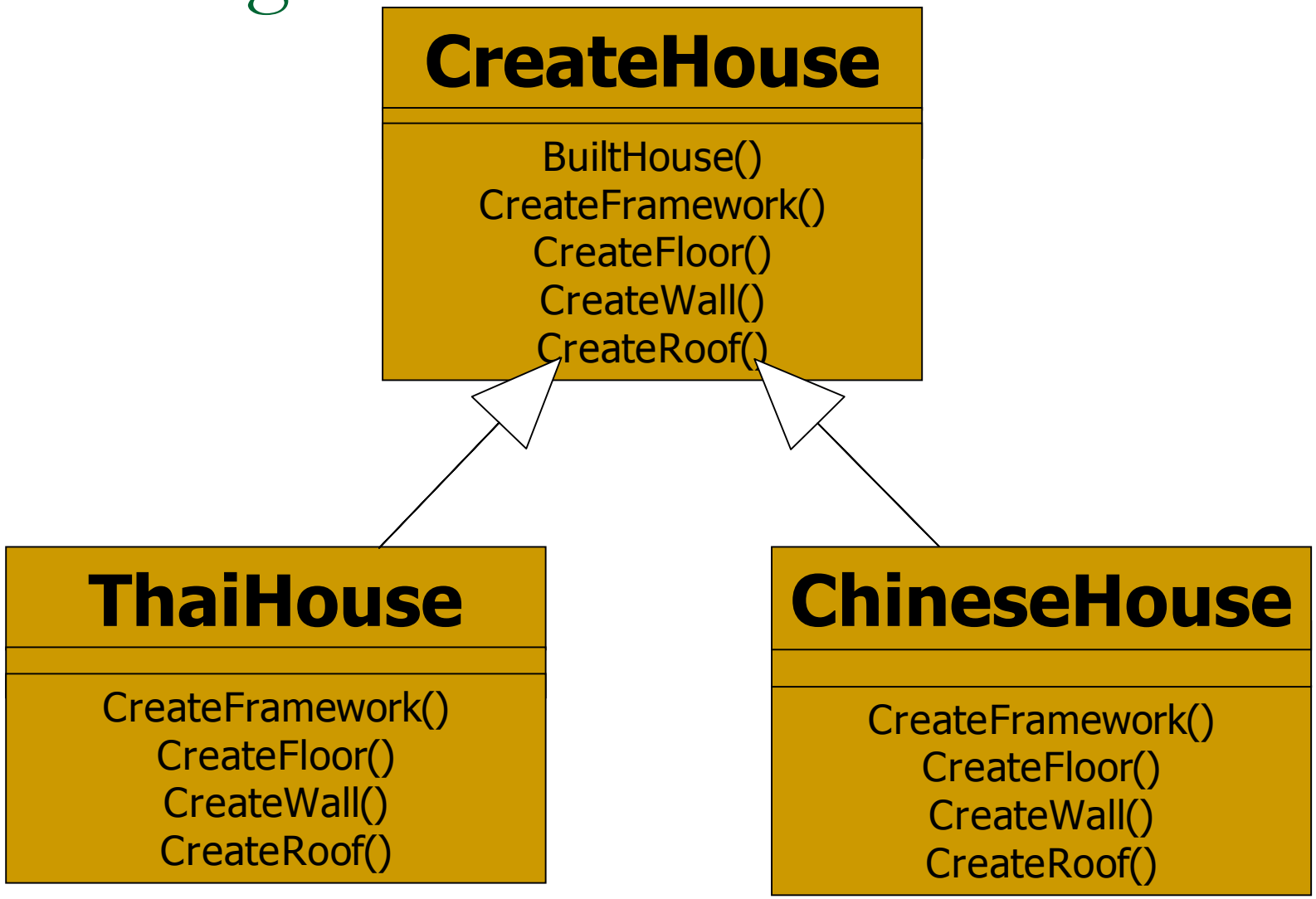
```
Output:
Loop #1 SubClassOne: uniqueMethod1
commonMethod
SubClassOne: uniqueMethod2
Loop #2 SubClassOne: uniqueMethod1
commonMethod
SubClassOne: uniqueMethod2

Loop #1 SubClassTwo: uniqueMethod1
commonMethod
SubClassTwo: uniqueMethod2
Loop #2 SubClassTwo: uniqueMethod1
commonMethod
 SubClassTwo: uniqueMethod2
```

Come back to our solution

Variations added to Template Floor Plan

Class Diagram



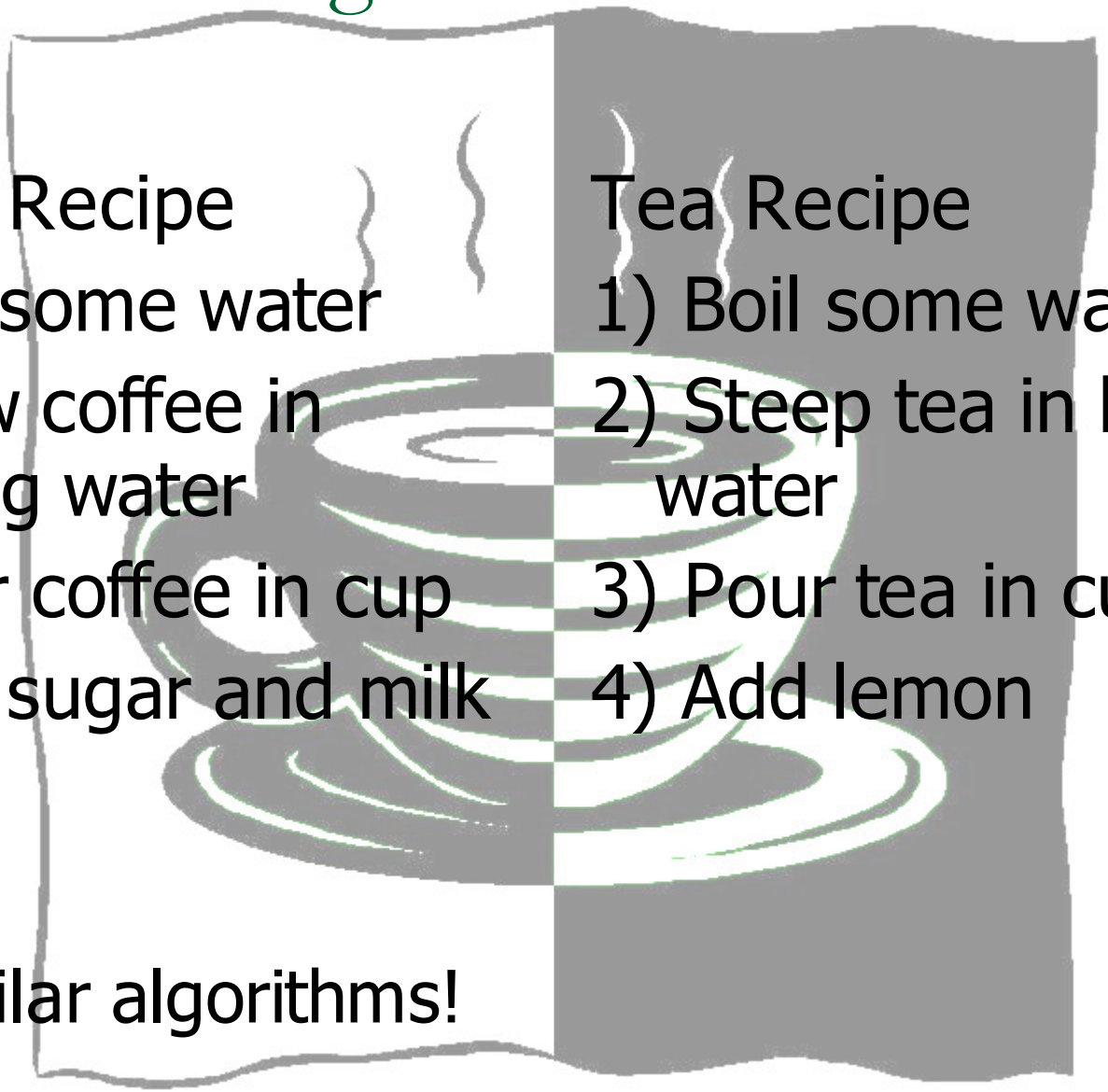
Coffee Break

It's time for some caffeine

Some people can't live without their coffee; some people can't live without their tea. The common ingredient?

Caffeine of course!

Barista Training Manual



Coffee Recipe

1) Boil some water
2) Brew coffee in boiling water
3) Pour coffee in cup
4) Add sugar and milk

Tea Recipe

1) Boil some water
2) Steep tea in boiling water
3) Pour tea in cup
4) Add lemon

=> Similar algorithms!

Whipping up some coffee and tea

```
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");
        }
}
```

Whipping up some coffee and tea

```
public class Tea {

        void prepareRecipe() {
                boilWater();
                brewTeaBag();
                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");
        }
}
```

Redesign the classes

We need to redesign the Coffee and Tea classes in order to remove redundancy.

Redesign the classes (Cont'd)

Tea
1) Boil some water.
2) Steep the teabag in the water
3) Pour tea in a cup
4) Add lemon

Coffee
1) Boil some water.
2) Brew the coffee grinds
3) Pour coffee in a cup
4) Add sugar and milk

Caffeine Beverage
1) Boil some water
2) Brew
3) Pour beverage in a cup
4) Condiments

2) Steep the teabag in the water
4) Add lemon

2) Brew the coffee grinds
4) Add sugar and milk

Meet the template Method

```
public abstract class CaffeineBeverage()
    void final prepareRecipe () {
        boilWater();
        brew();
        pourInCup();
        addCondiments();

}
        abstract void brew();

        abstract void addCondiments();

        void boilWater() {
                //implementation
        }
        void pourInCup() {
                //implementation
        }
}
```

Let's make some tea..

Okay, first we need a Tea object…

```
Tea myTea = new Tea();
```

Then we call the template method

```
myTea.prepareRecipe();
```

Which follows the algorithm for making caffeine beverages.

Let's make some tea..

- First we boil water:

  boilWater();

- Next we need to brew the tea, which only the subclass knows how to do:

  brew();

- Now we pour the tea in the cup; this is the same for all beverages so it happens in CaffeineBeverage:

  pourInCup();

- Finally, we add the condiments, which are specific to each beverage, so the subclass implements this:

  addCondiments();

What did Template Method get us?

Without TM

Coffee and tea are running the show;
they control the algorithm

Code is duplicated across coffee and tea

Classes are organized in a structure that requires a lot of work to add a new Caffeine beverage.

With TM

The CaffeineBeverage class runs the show; it has the algorithm, and protects it.

The CaffeineBeverage class maximizes reuse among the subclasses.

The Template Method version provides a framework that other caffeine beverage can be plugged into. New caffeine beverages only need to implement a couple a methods.

The 'hook'-method

```
abstract class AbstractClass {

        final void templateMethod () {
                primitiveOperation1();
                primitiveOperation2();
                concreteOperation();
                hook();
        }
        abstract void primitiveOperation1();
        abstract void primitiveOperation2();

        final void concreteOperation(){
                // implementation here
        }

        void hook();
}
```

The 'hook'-method (Cont'd)

```
public abstract class
CaffeineBeverageWithHook {

final voidprepareRecipe () {
        boilWater();
        brew();
        pourInCup();
        if (customerWantsCondiments()) {
                addCondiments();
        }
}

abstract void brew();

abstract void addCondiments();

voidboilWater() {
        System.out.println("Boiling water");
}
```

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

boolean
        customerWantsCondiments() {
        return true;
}
}
```

The subclass can ovveride this method, but doesn't have to.

The 'hook'-method (Cont'd)

```
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.println("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");
        }
        if (answer == null) {
                return "no);
        }
        return answer;
}
```

Here's where we override the hook and provide our own functionality.

Disadvantage

Disadvantage of Template methods

- Communicates intent poorly
- Difficult to compose functionality
- Difficult to comprehend program flow
- Difficult to maintain

Conclusion

Conclusion

- A "template method" defines the steps of an algorithm, deferring to subclasses for the implementation of those steps.
- The Template Method Pattern gives us an important technique for code reuse.
- The Template Method's abstract class may define concrete methods, abstract methods and hooks.
- Abstract methods are implemented by subclasses.

Conclusion

- Hooks are methods that do nothing or default behavior in the abstract class, but may be overridden in the subclass.
- To prevent subclasses from changing the algorithm in the template method, declare the template method as final.
- The Hollywood Principle guides us to put decision-making in high-level modules that can decide how and when to call low level modules.

Conclusion

- You'll see lots of uses of the Template Method Pattern in real world code, but don't expect it all (like any pattern) to be designed "by the book".
- The Strategy and Template Method Patterns both encapsulate algorithms, one by inheritance and one by composition.
- The Factory Method is specialization of Template Method.


Thank you