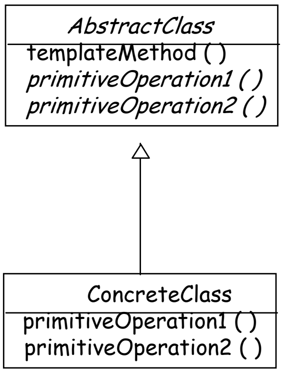
The Template Method Pattern

The **Template Method Pattern** defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Toni Sellarès
Universitat de Girona

The Template Method Pattern

You have an abstract class that is the base class of a hierarchy, and the behavior that is common to all objects in the hierarchy is implemented in the abstract class and other details are left to the individual subclasses.



Template Method allows you to define a skeleton of an algorithm in an operation and defer some of the steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Time for some caffeine….

Starbuzz Coffee Barista Training Manual

Baristas! Please follow these recipes precisely when preparing Starbuzz beverages.

Starbuzz Coffee Recipe

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

Starbuzz Tea Recipe

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

The recipe for coffee and tea are very similar!

Whipping up some Coffee in Java

```
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 thru filter”);
      }
      public void pourInCup  ( ) {
            System.out.println(“Pouring into cup”);
      }
      public void addSugarAndMilk ( ) {
            System.out.println (“Adding Sugar and Milk” );
      }
}
```

Recipe for coffee - each step is implemented as a separate method.

Each of these methods implements one step of the algorithm. There’s a method to boil the water, brew the coffee, pour the coffee in the cup, and add sugar and milk.

And now for the Tea….

```
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 pourInCup  ( ) {
            System.out.println(“Pouring into cup”);
      }
      public void addLemon ( ) {
            System.out.println (“Adding Lemon” );
      }
}
```

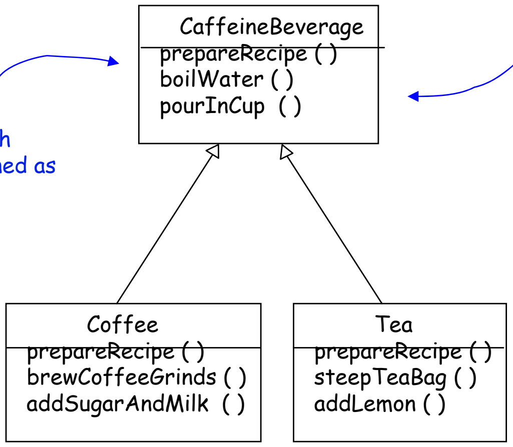
Very similar to the coffee - 2nd and 4th steps are different.

These methods are exactly the same

These methods are specialized to Tea

We have code duplication - that’s a good sign that we need to clean up the design. We should abstract the commonality into a base class since coffee and tea are so similar, right??
-- Give a class diagram to show how you would redesign the classes.

Sir, may I abstract your Coffee, Tea?



CaffeineBeverage
prepareRecipe ( )
boilWater ( )
pourInCup ( )

Coffee
prepareRecipe ( )
brewCoffeeGrinds ( )
addSugarAndMilk ( )

Tea
prepareRecipe ( )
steepTeaBag ( )
addLemon ( )

The prepareRecipe ( ) method differs in each subclass, so it is defined as abstract.

The boilWater( ) and pourInCup ( ) are shared by both subclasses, so defined in the superclass.

Each subclass implements its own recipe.

Each subclass( ) overrides the prepareRecipe ( ) method and implements its own recipe.

The methods specific to Coffee and Teac stay in the subclasses.

Is this a good redesign? Are we overlooking some other commonality? What are other ways that Coffee and Tea are similar?

What else do they have in common?

Both the recipes follow the same algorithm:

(1) Boil some water

(2) Use hot water to extract the tea or coffee

(3) Pour the resulting beverage into a cup

(4) Add the appropriate condiments to the beverage.

These two are already abstracted into the base class.

These aren't abstracted but are the same, they just apply to different beverages.

Can we abstract prepareRecipe ( ) too? Yes…

Abstracting PrepareRecipe ( )

Provide a common interface for the different methods

- Problem : Coffee uses brewCoffeeGrinds ( ) and addSugarAndMilk () methods while Tea uses steepTeaBag ( ) and addLemon ( ) methods

- Steeping and brewing are pretty analogous -- so a common interface may be the ticket: brew ( ) and addCondiments ()

The New Java Classes….

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

```
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 Tea extends CaffeineBeverage {
      public void brew ( ){
            System.out.println (“Steeping the Tes”);
      }
      public void addCondiments ( ) {
            System.out.println(“Adding Lemon”);
      }
}
```

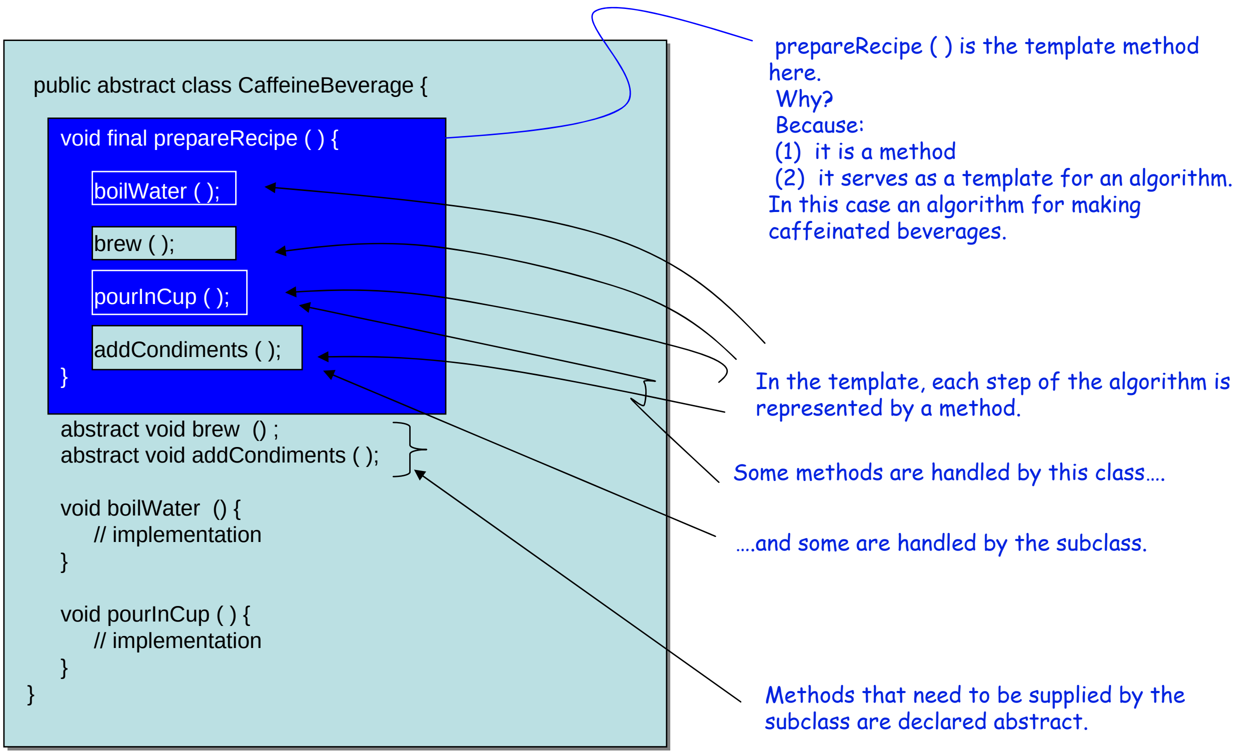
```
public class Coffee extends CaffeineBeverage {
      public void brew ( ) {
            System.out.println (“Dripping Coffee Thru the Filters”);
      }
      public void addCondiments ( ){
            System.out.println(“Adding Sugar and Milk”);
      }
}
```

What have we done?

- We have recognized that the two recipes are essentially the same, although some of the steps require different implementations.
  - So we’ve generalized the recipe and placed it in the base class.
  - We’ve made it so that some of the steps in the recipe rely on the subclass implementations.

Essentially - we have implemented the Template Method Pattern!

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
    }
}
```

prepareRecipe ( ) is the template method here.
Why?
Because:
(1) it is a method
(2) it serves as a template for an algorithm.
In this case an algorithm for making caffeinated beverages.

In the template, each step of the algorithm is represented by a method.

Some methods are handled by this class….

….and some are handled by the subclass.

Methods that need to be supplied by the subclass are declared abstract.

The Template Method defines the steps of an algorithm and allows subclasses to provide the implementation of one or more steps.

Behind the scenes…..

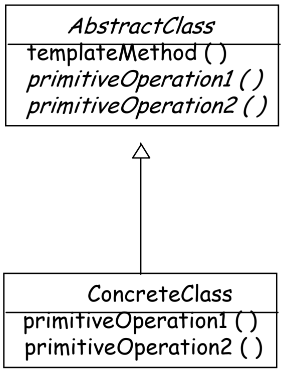
```
 Tea myTea = new Tea ( );

myTea.prepareRecipe ( );
            boilWater ( );
            brew ( );
            pourInCup ( );
            addCondiments ( );
```

Polymorphism ensures that while the template controls everything, it still calls the right methods.

The Template Method

The **Template Method Pattern** defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure.



The template method makes use of the primitiveOperations to implement an algorithm. It is decoupled from the actual implementations of these operations.

The Abstract Class contains the template method.

….and abstract versions of the operations used in the template method.

There may be many ConcreteClasses each implementing the full set of operations required by the template method.

The ConcreteClass implements the abstract operations, which are called when the templateMethod ( ) needs them.

Hooked on the Template Method

A ***hook*** is a method that is declared in the abstract class, but only given an empty or default implementation.

– Gives the subclasses the ability to “*hook into*” the algorithm at various points, if they wish; they can ignore the hook as well.

```
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 condiments, only then do we call addCondiments ( )

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

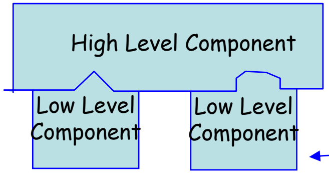
If subclasses want to use the hook they simply override it!

The Hollywood Principle

Don’t call us, we’ll call you!

With the Hollywood principle

- We allow low level components to hook themselves into a system
- But high level components determine when they are needed and how.
- High level components give the low-level components a “don’t call us, we’ll call you” treatment.

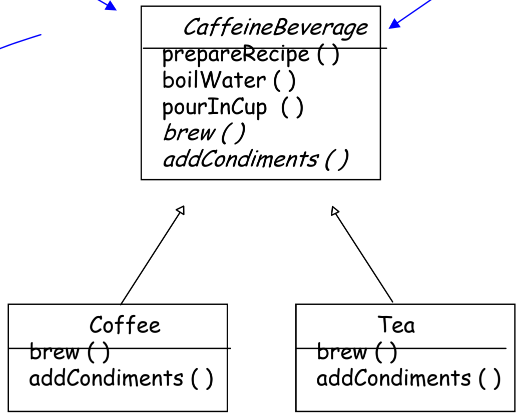


But high-level components control when and how.

Low-level components can participate in the computation.

A low-level component never calls a high-level component directly.

The Hollywood Principle and the Template Method



CaffeineBeverage is our high-level component. It has control over the algorithm for the recipe, and calls on the subclasses only when they are needed for an implementation of a method.

Clients of beverages will depend on the CaffeineBeverage abstraction rather than a concrete Tea or Coffee, which reduces dependencies in the overall system.

The subclasses are used simply to provide implementation details.

Tea and Coffee never call the abstract class directly without being “called” first.

Template Method implementation in Java

- Give primitive and hook methods protected access
  - These methods are intended to be called by a template method, and not directly by clients
- Declare primitive methods as abstract in the superclass
  - Primitive methods **must** be implemented by subclasses
- Declare hook methods as non-abstract
  - Hook methods **may** optionally be overridden by subclasses
- Declare template methods as final
  - This prevents a subclass from overriding the method and interfering with it’s algorithm structure

Template Method: Structural Example 1

```
AbstractTemplate.java

public abstract class AbstractTemplate {
           final void templateMethod() {
                      operation1();
                      operation2();
                      operation3();
           }
           abstract void operation1();
           abstract void operation2();
           final void operation3() {
                   System.out.println("I am defined in the super
   class.");
           }
}
```

```
ConcreteTemplate.java
public class ConcreteTemplate extends AbstractTemplate {
       @Override
       void operation1() {
            System.out.println("I am defined in the sub class.");
            }
       @Override
       void operation2() {
            System.out.println("I am defined in the sub class.");
            }
 }
```

```
TemplateTester.java

public class TemplateTester {
    public static void main(String[] args) {
        AbstractTemplate temp = new ConcreteTemplate();
        temp.templateMethod();
        }
 }
```

Template Method: Structural Example 2

```
public abstract class AbstractTemplateWithHook {
   final void templateMethod() {
        operation1();
        operation2();
        operation3();
        hook();
   }

   abstract void operation1();
   abstract void operation2();
   final  void operation3() {
        System.out.println("I am defined in the super class.");
   }

// A subclass can override the hook if it wishes
   boolean hook() {
        System.out.println("Returning true");
        return true;
   }
}
```

```
public class ConcreteTemplateWithHook extends AbstractTemplateWithHook
{
   @Override   void operation1() {
        System.out.println("I am defined in the sub class.");
   }
   @Override   void operation2() {
        System.out.println("I am defined in the sub class.");
   }
   boolean hook() {
        System.out.println("returning false");
        return false;
   }
}

public class TemplateTesterWithHook {
/**
 * @param args
 */
    public static void main(String[] args) {
        AbstractTemplateWithHook temp = new
       ConcreteTemplateWithHook();
       temp.templateMethod();
    }
}
```

Swingin’ with Frames

- JFrame - most basic Swing container and inherits a paint ( ) method.
- Default behavior of paint ( ) method - does nothing because it is a *hook*.
- By overriding the paint () method, you can insert yourself into JFrame’s algorithm for displaying its area of screen and have your own graphic output incorporated into the JFrame.
- Next up -- a simple example using JFrame to override the paint ( ) method.

Simple JFrame Example

```
public class MyFrame extends JFrame {
      public MyFrame (String title) {
             super (title);
             this.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);

             this.setSize ( 300, 300);
             this.setVisible (true);
      }
      public void paint (Graphics graphics){
             super.paint (graphics);
             String msg = “I rule!!”;
             graphics.drawString(msg, 100, 100);
      }
      public static void main (String[] args){
             MyFrame myFrame = new MyFrame (“Head First Design Patterns”);
      }
}
```

We’re extending JFrame, which contains a method update () that controls the algorithm for updating the screen. We can hook into that algorithm by overriding the paint () method.

Don’t look behind the curtain, just some initialization here….

JFrame’s update algorithm calls paint ( ). By default paint() does nothing…it’s a hook. We are overriding paint(), and telling the JFrame to draw a message in the window.

Applets

- Applets provide numerous hooks!

```
public class MyApplet extends Applet {
     String message;

     public void init ( ){
          message = “Hello World, I’m alive!”;
          repaint ( );
     }
     public void start ( ){
          message = “Now I’m starting up…”;
          repaint ( );
     }
     public void stop ( ) {
          message = “Now I’m being stopped…”;
          repaint ();
     }
     public void destroy ( ){
          // applet is going away
     }
     public void paint  (Graphics g){
          g.drawString(message, 5, 15);
     }
}
```

The init hook allows the applet to do whatever it wants to initialize the applet the first time.

repaint ( ) is a concrete method in the Applet class that lets upper-level components know that the applet needs to be redrawn.

The start hook allows the applet to do something when the applet is just about to be displayed on the web page.

If the user goes to another page, the stop hook is used and the applet can do whatever it needs to do to stop its actions.

And the destroy hook is used when the applet is going to be destroyed, say, when the browser pane is closed. We could try and display something here…but what’s the point?

Well looky here! Applet also makes use of the paint() method as a hook.

Concrete applets make extensive use of hooks to supply their own behaviors. Because these methods are implemented as hooks, the applet isn’t required to implement them.

Summary (1/2)

- 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.
- *Hooks* are methods that do nothing or default behavior in the abstract class, but may be overridden in the subclasses.

Summary (2/2)

- The Hollywood Principle guides us to put decision making in high level modules that can decide how and when to call the low-level modules.
- 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 Factory Method is a specialization of the Template Method!