COMP2511

Template Pattern

Prepared by

Dr. Ashesh Mahidadia

Template Pattern: Motivation and 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*." [GoF]
- A *template Method* defines the skeleton (structure) of a behavior (by implementing the invariant parts).
- A *template Method* calls *primitive operations*, that could be implemented by sub classes OR has default implementations in an abstract super class.
- Subclasses can redefine only certain parts of a behavior without changing the other parts or the structure of the behavior.

Template Pattern: Motivation and Intent

- Subclasses do not control the behavior of a parent class,

  a parent class calls the operations of a subclass and not the other way around.

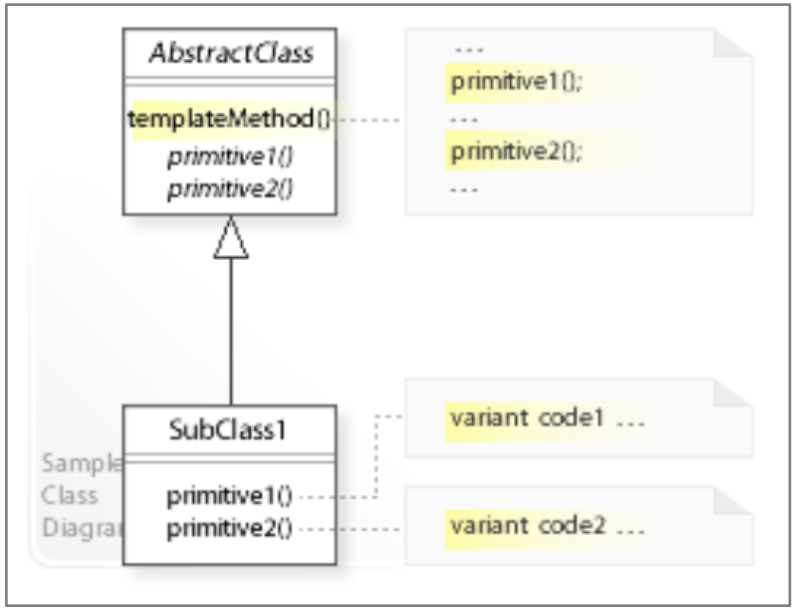
- Inversion of control:
  - when using a library (reusable classes), we call the code we want to reuse.
  - when using a framework (like Template Pattern), we write subclasses and implement the variant code the framework calls.

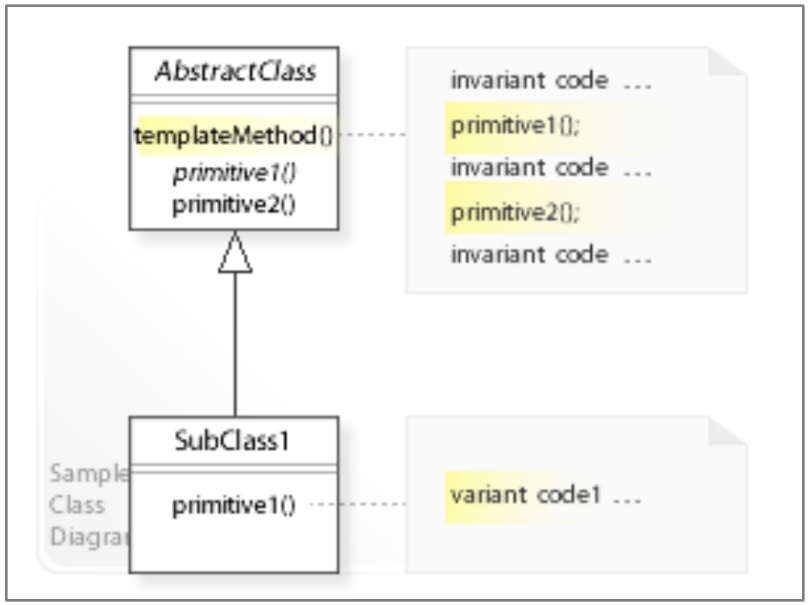
- Template pattern implement the common (invariant) parts of a behavior once "and leave it up to subclasses to implement the behavior that can vary."[GoF, p326]

- Invariant behavior is in one class (localized)

Template Pattern: Structure

- Abstract class defines a `templateMethod()` to implement an invariant structure (behaviour)
- `templateMethod()` calls methods defined in the abstract class (abstract or concrete) - like primitive1, primitive2, etc.
- Default behaviour can be implemented in the abstract class by offering concrete methods
- Importantly, sub classes can implement primitive methods for variant behaviour

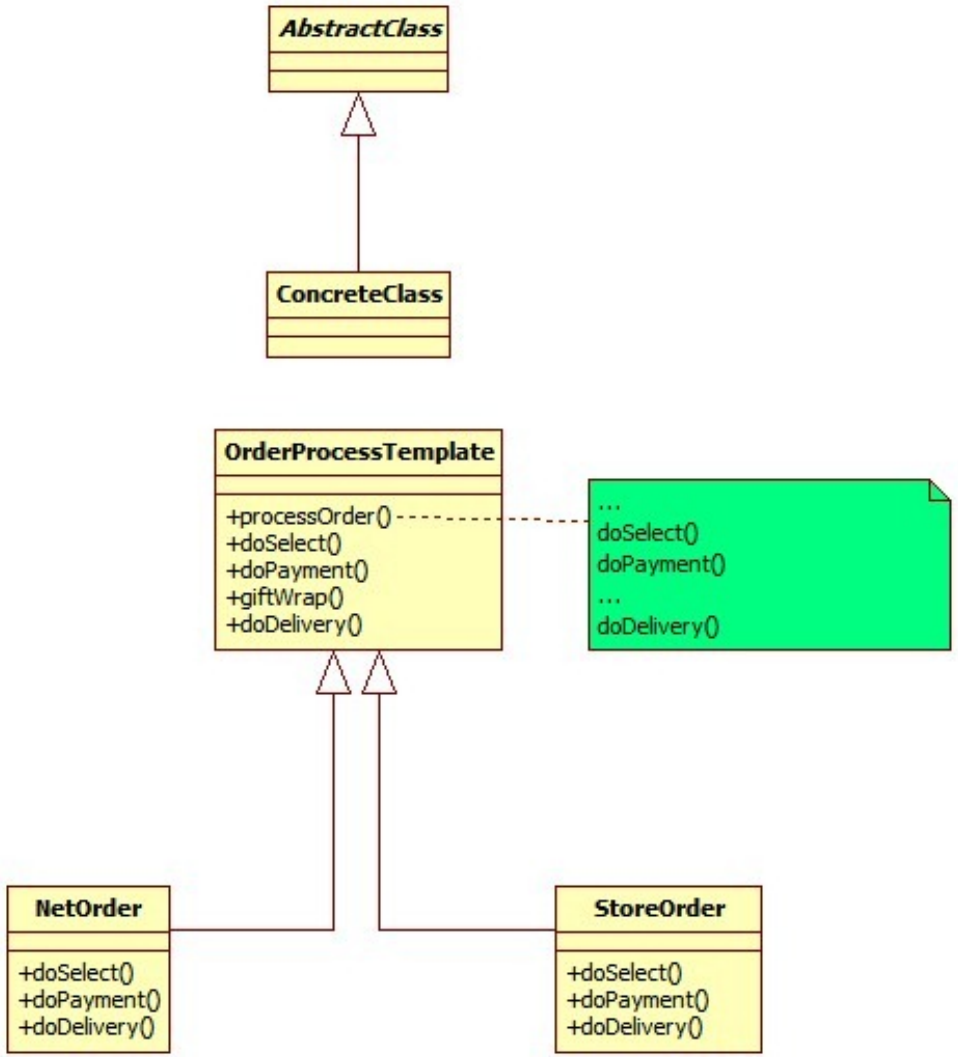




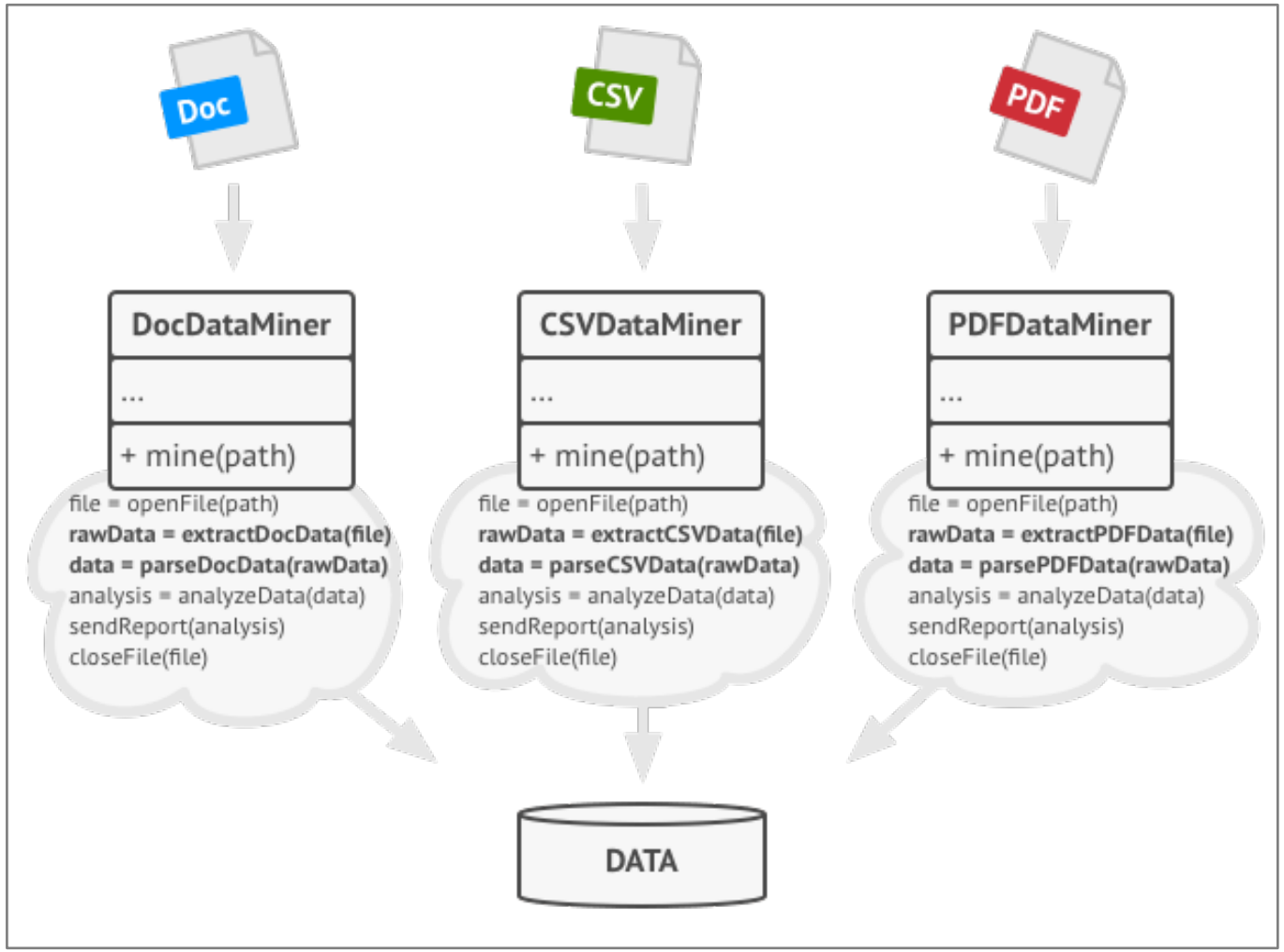
Template Pattern: Structure

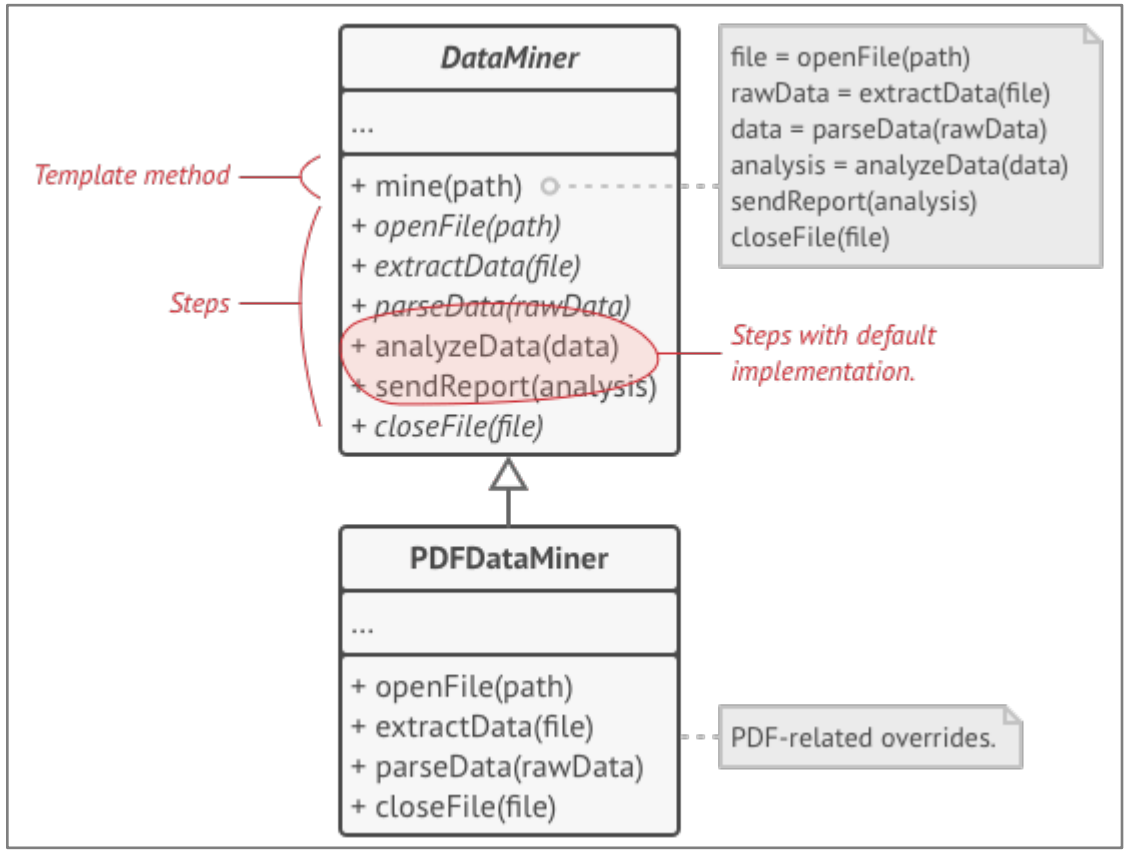
- "To reuse an abstract class effectively, subclass writers must understand which operations are designed for overriding." [GoF, p328]
- **Primitive operations** : operations that have default implementations or must be implemented by sub classes.
- Final operations: concrete operations that cannot be overridden by sub classes.
- **Hook operations**: concrete operations that do nothing by default and can be redefined by subclasses if necessary. This gives subclasses the ability to “hook into” the algorithm at various points, if they wish; a subclass is also free to ignore the hook. (see the example)

Template Pattern: Example



Template Pattern: Example





- From https://refactoring.guru/design-patterns/template-method

Template Pattern: Example

Read the example code discussed/developed in the lectures, and also provided for this week

```java
public abstract class MyReportTemplate {

    public void genReport() {

        InputStream f1 = openFile();

        SortedMap<String, ArrayList<String>> data = parseFile( f1 );

        generateReport ( data );

        if( isRequestedSummary()){
            generateSummary(data);
        }

    }

    public void generateSummary(SortedMap<String, ArrayList<String>> data) {
        System.out.println("generating Summary (default from MyReportTemplat ...");

    }

    public boolean isRequestedSummary() {
        return false;
    }

    public void generateReport(SortedMap<String, ArrayList<String>> data) {
        System.out.println("generating report (default from MyReportTemplat ...");

    }

    protected abstract SortedMap<String, ArrayList<String>> parseFile(InputStream f1);
    public abstract String getFilename();

    public InputStream openFile() {

        String filename = getFilename();
        InputStream f1 = null;

        try {
            f1 = new FileInputStream(filename);
        } catch (FileNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        return f1;
    }
```

Template method

Step 1

Step 2

Step 3

Hook

Step 4

Default methods

Abstract methods

Default method

Template Pattern: Example

Read the example code discussed/developed in the lectures, and also provided for this week

```java
public class CSVReport extends MyReportTemplate{
    private String fname = "";
    private boolean reqSummary = false;

    public CSVReport() {
        super();
        fname = "src/example/data.csv";
        reqSummary = false;
    }
    public CSVReport(String filename, boolean requestSummary) {
        this.fname = filename;
        this.reqSummary = requestSummary;
    }

    @Override
    protected SortedMap<String, ArrayList<String>> parseFile(InputStream f1) {
        // CSV parsing code here
        System.out.println("parsing CSV data file: " + getFilename());
        TreeMap<String, ArrayList<String>> data =
                new TreeMap<String, ArrayList<String>>();
        // populate data object in this method  ..
        return data;
    }

    @Override
    public String getFilename() {
        // ask user for a file name.. or get from a constructor
        return fname;
    }

    @Override
    public boolean isRequestedSummary() {

        return this.reqSummary;
    }
```

Step 2

Part of Step 1

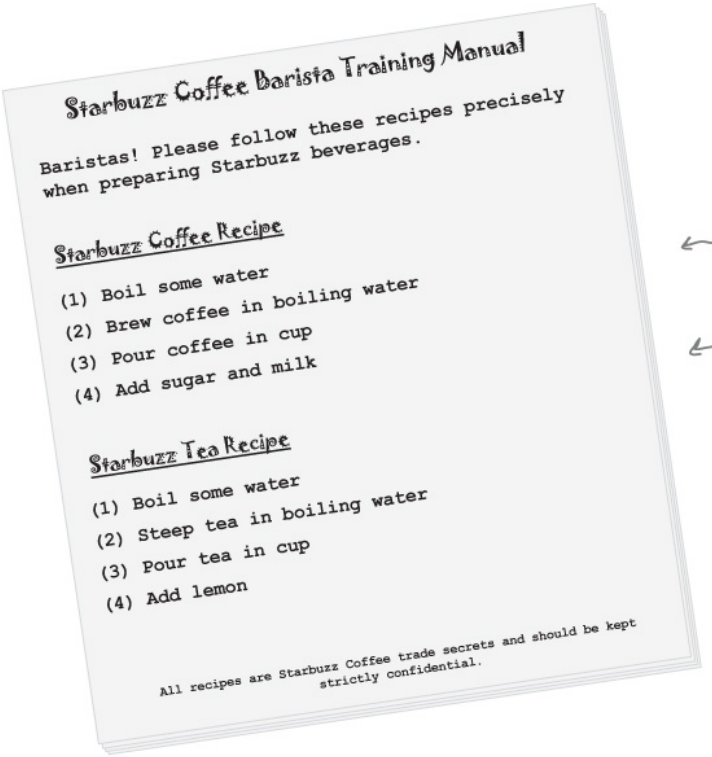
Hook

Template Pattern: Example

Read the example code discussed/developed in the lectures, and also provided for this week

```java
public class Test1 {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        System.out.println("\n*** Generate CSV report ... ... ");

        CSVReport rep1 = new CSVReport("src/example/data.csv", true);
        rep1.genReport();

        System.out.println("\n*** Generate XML report ... ... ");

        XMLReport rep2 = new XMLReport("src/example/data.xml");
        rep2.genReport();

        System.out.println("\n*** Generate CSV with Summarys report ... ... ");

        CSVReportWithSummary rep3 = new CSVReportWithSummary();
        rep3.genReport();


    }

}
```

Template Pattern: Example



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

All recipes are Starbuzz Coffee trade secrets and should be kept strictly confidential.

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

Here's our Coffee class for making coffee.

Here's our recipe for coffee, straight out of the training manual.

Each of the steps is implemented as a separate method.

```
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");
    }
}
```

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

From the Head First Design Book

Template Pattern: Example

```
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");
    }
}
```

This looks very similar to the one we just implemented in Coffee; the second and fourth steps are different, but it's basically the same recipe.

These two methods are specialized to Tea.

Notice that these two methods are exactly the same as they are in Coffee! So we definitely have some code duplication going on here.

From the Head First Design Book

Template Pattern: Example

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

```
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 our **template method**. Why?

Because:

(1) It is a method, after all.

(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.

The methods that need to be supplied by a subclass are declared abstract.

From the Head First Design Book

Template Pattern: Example (hook)

```
public abstract class CaffeineBeverageWithHook {

    final 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().

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.

From the Head First Design Book

Template Pattern: Example (hook)

```
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.

From the Head First Design Book

Template Vs Strategy Patterns

- Template Method works at the class level, so it's **static**.
- Strategy works on the object level, letting you switch behaviors at **runtime**.
- Template Method is based on inheritance: it lets you alter parts of an algorithm by extending those parts in subclasses.
- Strategy is based on composition: you can alter parts of the object's behavior by supplying it with different strategies that correspond to that behavior at runtime.