BBM 486
Design Patterns Project

Subject: Template Design Pattern

Group Members:

21685417 - Nilay Doğan
21627059 - Mustafa Candan
21526695 - Denizcan Bağdatlıoğlu
21627538 - Batuhan Mete
21627466 - Ali Osman Kocaman

What does “template” mean?

template

/ˈtɛmpleɪt, ˈtɛmplət/

Tanımları şurada görün:

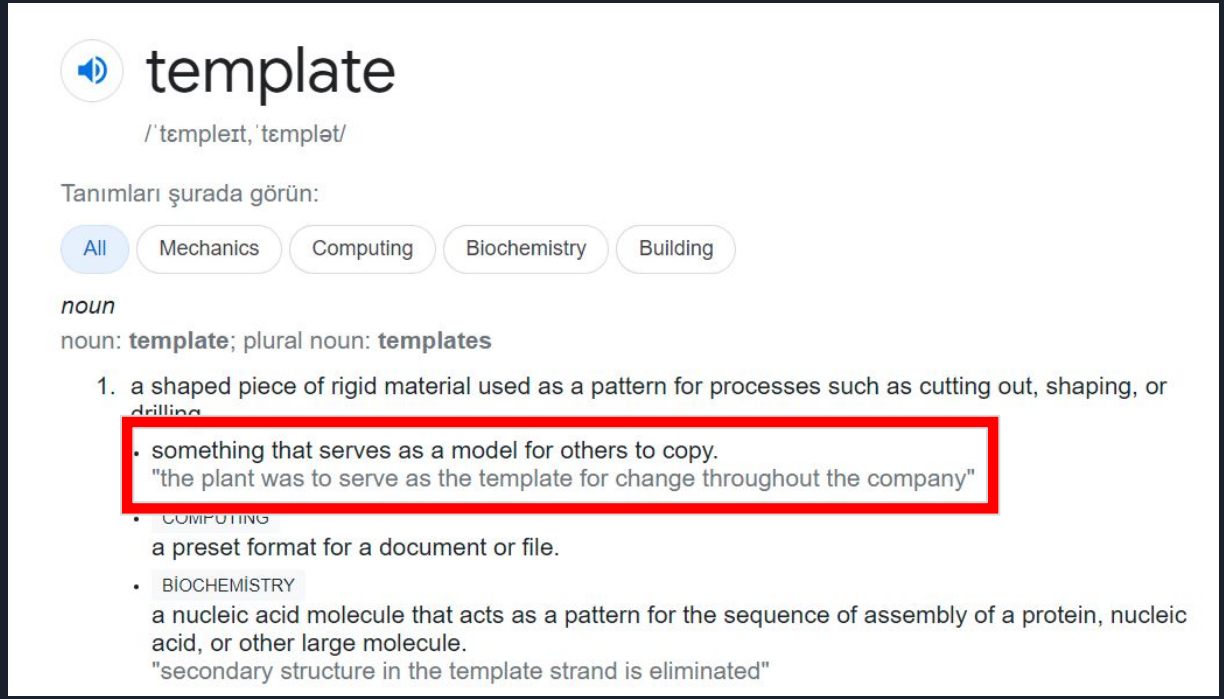
All | Mechanics | Computing | Biochemistry | Building

noun

noun: **template**; plural noun: **templates**

1. a shaped piece of rigid material used as a pattern for processes such as cutting out, shaping, or drilling.
   - something that serves as a model for others to copy.
     "the plant was to serve as the template for change throughout the company"
   - COMPUTİNG
     a preset format for a document or file.
   - BİOCHEMİSTRY
     a nucleic acid molecule that acts as a pattern for the sequence of assembly of a protein, nucleic acid, or other large molecule.
     "secondary structure in the template strand is eliminated"

What does “template” mean?



template

/ˈtɛmpleɪt, ˈtɛmplət/

Tanımları şurada görün:

All | Mechanics | Computing | Biochemistry | Building

noun

noun: **template**; plural noun: **templates**

1. a shaped piece of rigid material used as a pattern for processes such as cutting out, shaping, or drilling

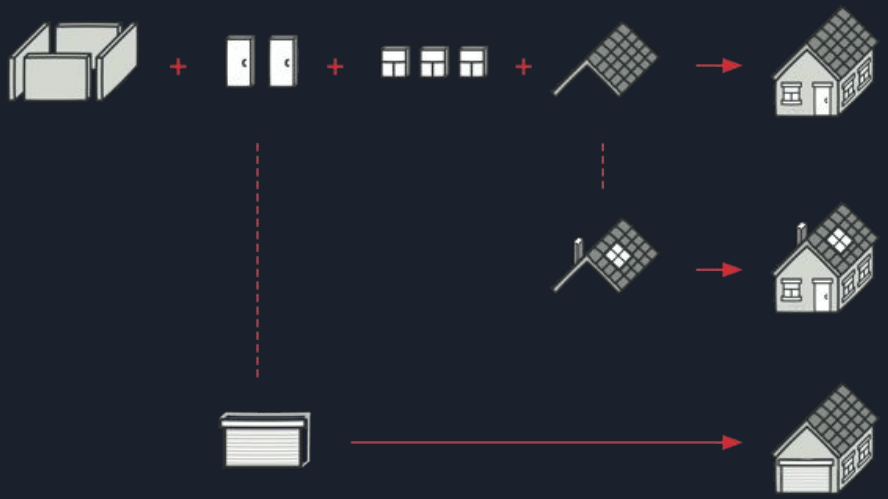
   - something that serves as a model for others to copy.
     "the plant was to serve as the template for change throughout the company"
   - COMPUTING
     a preset format for a document or file.
   - BİOCHEMİSTRY
     a nucleic acid molecule that acts as a pattern for the sequence of assembly of a protein, nucleic acid, or other large molecule.
     "secondary structure in the template strand is eliminated"

Template Design Pattern

- is a behavioral design pattern
- defines the skeleton of an algorithm in the super-class
- lets subclasses override specific steps of the algorithm
- is mostly used in framework development
- defines the sequential steps to execute a multi-step algorithm

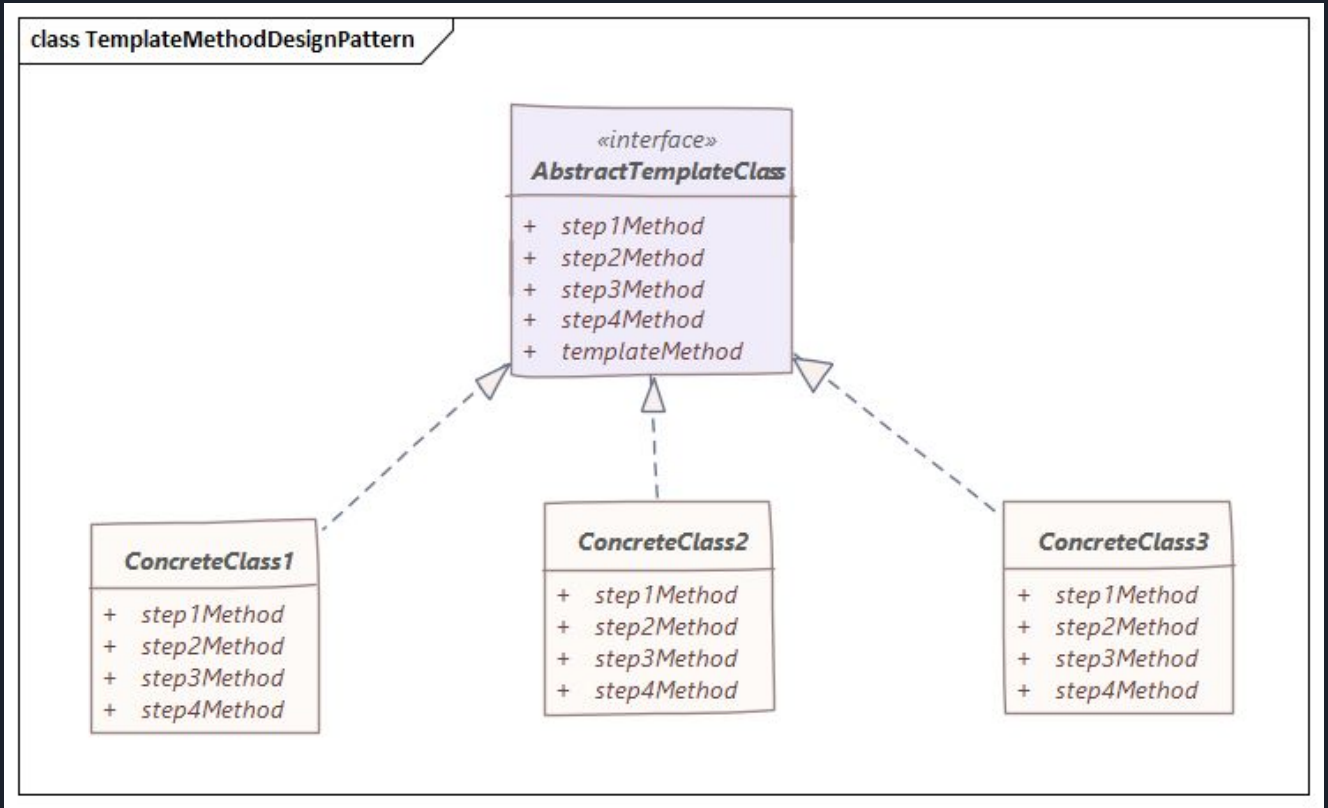
- these steps can be created as an abstract method

Real-World Analogy

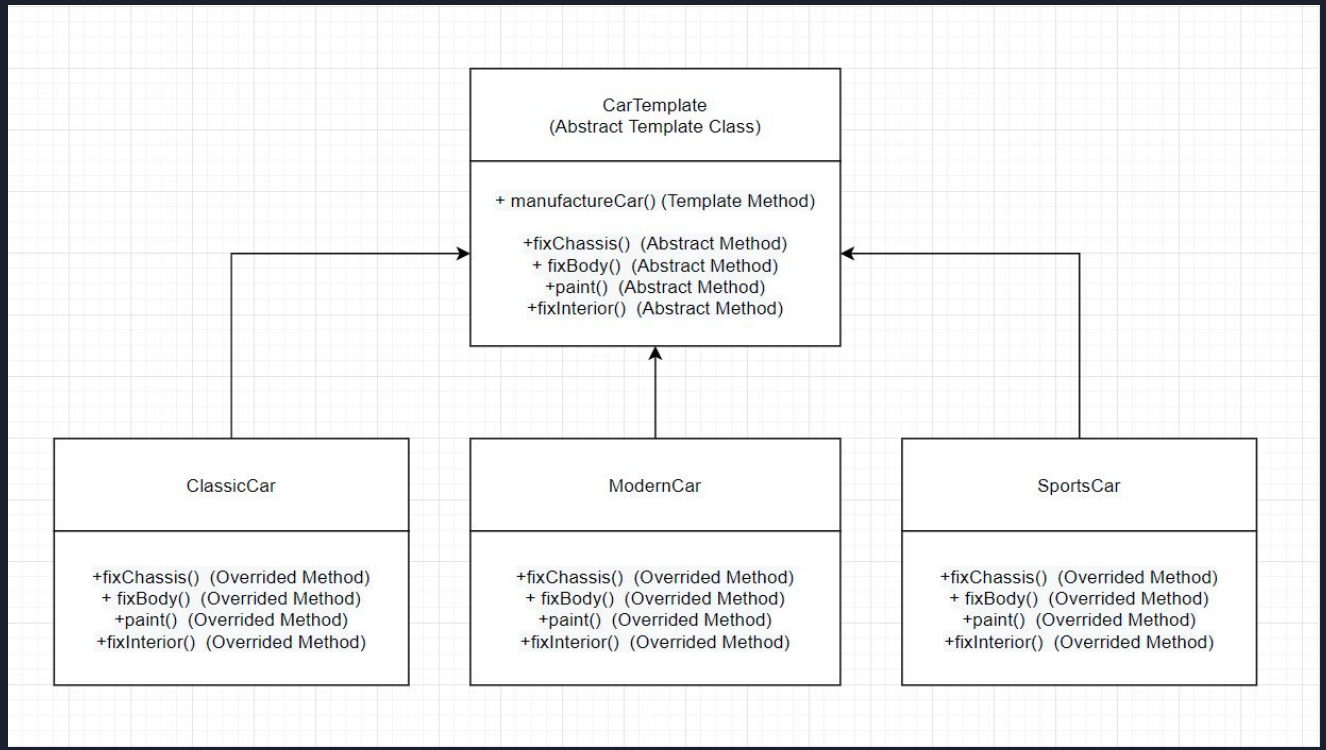


- this method approach can be used in mass housing construction.
- house may contain extension points to let owner adjust some details.
- each building step can have slight changes.
- this changes can lead the differences from other houses.

The UML Diagram



CarManufacturing Example



CarTemplate Class

Java

```java
package org.trishinfotech.template;

public abstract class CarTemplate {

    protected String chassis;
    protected String body;
    protected String paint;
    protected String interior;

    public CarTemplate() {
        super();
    }

    // steps
    public abstract void fixChassis();

    public abstract void fixBody();

    public abstract void paint();

    public abstract void fixInterior();

    // template method
    public void manufactureCar() {
        fixChassis();
        fixBody();
        paint();
        fixInterior();
    }

```

ClassicCar class

```java
package org.trishinfotech.template;

public class ClassicCar extends CarTemplate {

    public ClassicCar() {
        super();
    }

    @Override
    public void fixChassis() {
        System.out.println("Assembling chassis of the classical model");
        this.chassis = "Classic Chassis";
    }

    @Override
    public void fixBody() {
        System.out.println("Assembling body of the classical model");
        this.body = "Classic Body";
    }

    @Override
    public void paint() {
        System.out.println("Painting body of the classical model");
        this.paint = "Classic White Paint";
    }

    @Override
    public void fixInterior() {
        System.out.println("Setting up interior of the classical model");
        this.interior = "Classic interior";
    }

}

```

ModernCar class

```java
package org.trishinfotech.template;

public class ModernCar extends CarTemplate {

    public ModernCar() {
        super();
    }

    @Override
    public void fixChassis() {
        System.out.println("Assembling chassis of the modern model");
        this.chassis = "Modern Chassis";
    }

    @Override
    public void fixBody() {
        System.out.println("Assembling body of the modern model");
        this.body = "Modern Body";
    }

    @Override
    public void paint() {
        System.out.println("Painting body of the modern model");
        this.paint = "Modern Black Paint";
    }

    @Override
    public void fixInterior() {
        System.out.println("Setting up interior of the modern model");
        this.interior = "Modern interior";
    }

}

```

SportsCar class

```java
package org.trishinfotech.template;

public class SportsCar extends CarTemplate {

    public SportsCar() {
        super();
    }

    @Override
    public void fixChassis() {
        System.out.println("Assembling chassis of the sports model");
        this.chassis = "Sporty Chassis";
    }

    @Override
    public void fixBody() {
        System.out.println("Assembling body of the sports model");
        this.body = "Sporty Body";
    }

    @Override
    public void paint() {
        System.out.println("Painting body of the sports model");
        this.paint = "Sporty Torch Red Paint";
    }

    @Override
    public void fixInterior() {
        System.out.println("Setting up interior of the sports model");
        this.interior = "Sporty interior";
    }

}

```

Main class

```
1  package org.trishinfotech.template;
2
3  public class Main {
4
5      public static void main(String[] args) {
6          CarTemplate car = new SportsCar();
7          car.manufactureCar();
8          if (car != null) {
9              System.out.println("Below car delievered: ");
10             System.out.println("================================================================================");
11             System.out.println(car);
12             System.out.println("================================================================================");
13         }
14     }
15
16 }
17
```

The Output

```
Java
1 Assembling chassis of the sports model
2 Assembling body of the sports model
3 Painting body of the sports model
4 Setting up interior of the sports model
5 Below car delievered:
6 ================================================================================
7 Car [chassis=Sporty Chassis, body=Sporty Body, paint=Sporty Torch Red Paint, interior=Sporty interior]
8 ================================================================================
9
```

When to use Template Method:

- to let subclasses implement varying behavior
- to avoid duplication in the code the general workflow structure is implemented once in the abstract class's algorithm, and necessary variations are implemented in the subclasses
- to control at what points subclassing is allowed as opposed to a simple polymorphic override, where the base method would be entirely rewritten allowing radical change to the workflow, only the specific details of the workflow are allowed to change

➢ The template pattern is also used to implement the Hollywood Principle which says " Don't call us, we'll call you" with the use of Template Method Pattern, we are telling subclasses, "Don't call us, we'll call you".

Relations with Other Patterns

Factory method is specialization of Template Method. At the same time, a Factory Method can serve as a step in a large Template Method.

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

Pros and Cons

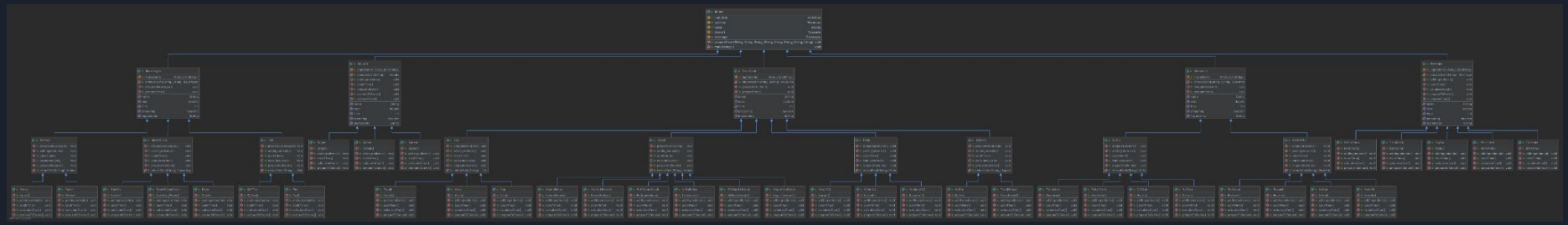
Pros

- You can let clients can override only certain parts of a large algorithm, making them less affected by changes happened to the other parts of the algorithm.
- You can pull the duplicate code into a superclass.

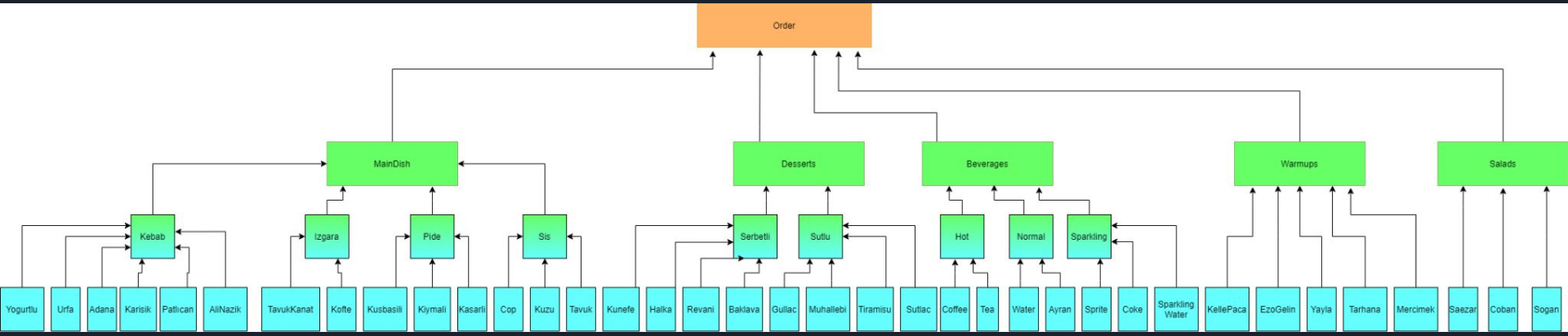
Cons

- Some clients may be limited by the provided skeleton of an algorithm.
- You might violate the Liskov Substitution Principle by suppressing a default step implementation via a subclass.
- Template methods tend to be harder to maintain the more steps they have.

Our UML Diagram



Our UML Diagram (Simplified Version)

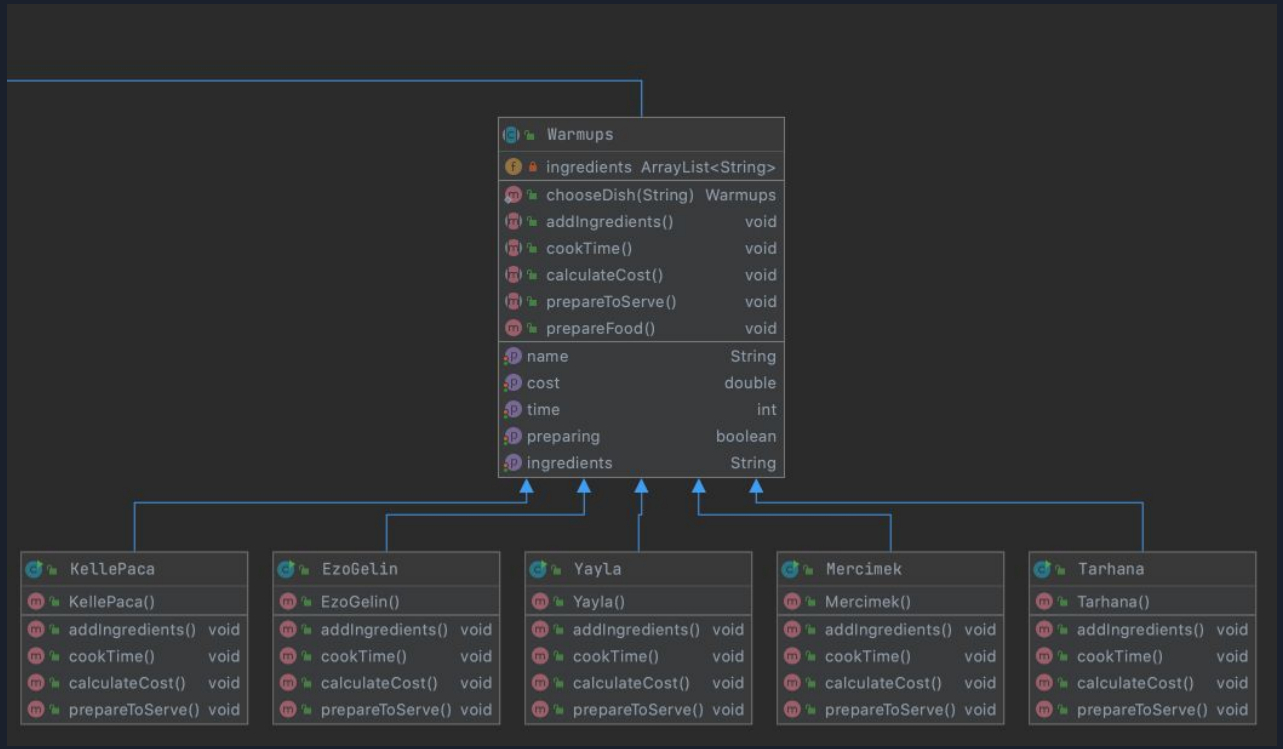


Abstract Class

Abstract + Concrete Class

Concrete Class

UML Diagram(Warmups)



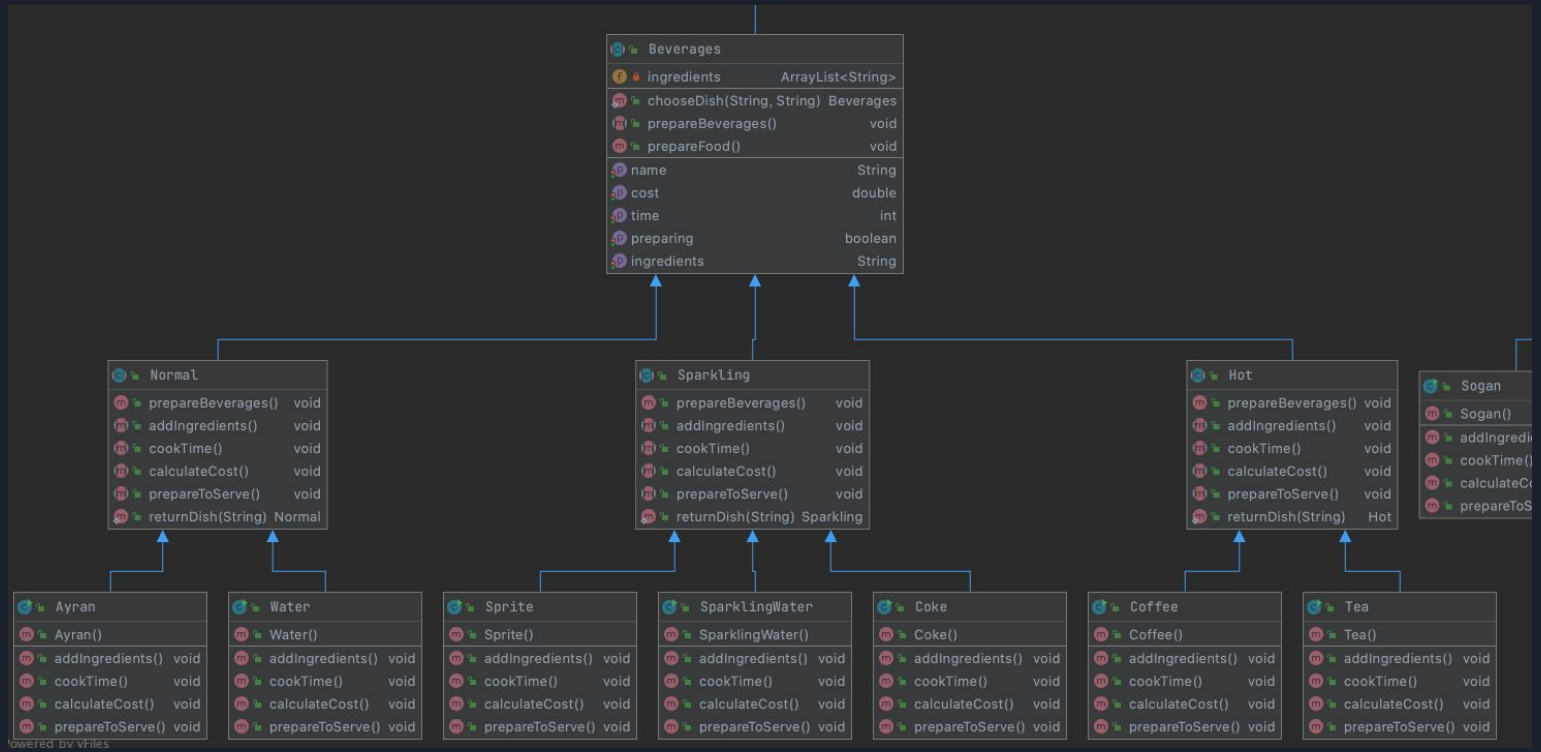
Warmups (Abstract Class)

```
abstract public class Warmups extends Order{
    private String name;
    private int time;
    private double cost;
    private boolean preparing;
    private ArrayList<String> ingredients = new ArrayList<>();

    public static Warmups chooseDish(String dish){
        if(dish.equalsIgnoreCase("ezo gelin")){
            return new EzoGelin();
        }else if(dish.equalsIgnoreCase("kelle paca")){
            return new KellePaca();
        } else if(dish.equalsIgnoreCase("mercimek")){
            return new Mercimek();
        }else if(dish.equalsIgnoreCase("tarhana")){
            return new Tarhana();
        }else if(dish.equalsIgnoreCase("yayla")){
            return new Yayla();
        }
        return null;
    }

    //steps
    public abstract void addIngredients();
    public abstract void cookTime();
    public abstract void calculateCost();
    public abstract void prepareToServe();

    //template method
    public void prepareFood(){
        addIngredients();
        cookTime();
        calculateCost();
        prepareToServe();
    }
```

Mercimek (Concrete Class)

```
public class Mercimek extends Warmups{
    public Mercimek() { super.setName("Mercimek Corbasi"); }

    @Override
    public void addIngredients() {
        super.setIngredients("Kırmızı mercimek");
        super.setIngredients("Soğan");
        super.setIngredients("Patates");
    }

    @Override
    public void cookTime() { super.setTime(15); }

    @Override
    public void calculateCost() { super.setCost(11); }

    @Override
    public void prepareToServe() { super.setPreparing(true); }
}
```

UML Diagram (Beverages)



Beverages Class (Abstract Class)

```
abstract public class Beverages extends Order{
    private String name;
    private int time;
    private double cost;
    private boolean preparing;
    private ArrayList<String> ingredients = new ArrayList<>();

    public static Beverages chooseDish(String beverageType, String drink){
        if(beverageType.equalsIgnoreCase("hot")){
            return Hot.returnDish(drink);
        }else if(beverageType.equalsIgnoreCase("normal")){
            return Normal.returnDish(drink);
        }else if(beverageType.equalsIgnoreCase("sparkling")){
            return Sparkling.returnDish(drink);
        }
        return null;
    }

    public abstract void prepareBeverages();
    //template method
    public void prepareFood(){

        prepareBeverages()
    }
```

Hot Class (Abstract + Concrete Class)

```
abstract public class Hot extends Beverages {

    @Override
    public void prepareBeverages(){
        addIngredients();
        cookTime();
        calculateCost();
        prepareToServe();
    }

    public abstract void addIngredients();
    public abstract void cookTime();
    public abstract void calculateCost();
    public abstract void prepareToServe();



    public static Hot returnDish(String type){
        if(type.equalsIgnoreCase("coffee")){
            return new Coffee();
        }else if(type.equalsIgnoreCase("tea")){
            return new Tea();
        }
        return null;
    }
}
```

Tea Class (Concrete Class)

```
public class Tea extends Hot{
    public Tea() { super.setName("Tea"); }

    @Override
    public void addIngredients() {
        super.setIngredients("Tea");
        super.setIngredients("Water");
    }

    @Override
    public void cookTime() { super.setTime(7); }

    @Override
    public void calculateCost() { super.setCost(3); }

    @Override
    public void prepareToServe() { super.setPreparing(true); }
}
```

References

1. https://www.geeksforgeeks.org/template-method-design-pattern/
2. https://refactoring.guru/design-patterns/template-method
3. https://dzone.com/articles/using-template-method-design-pattern-in-java

Thank You For Listening!