Template Method

Liad Vaksman

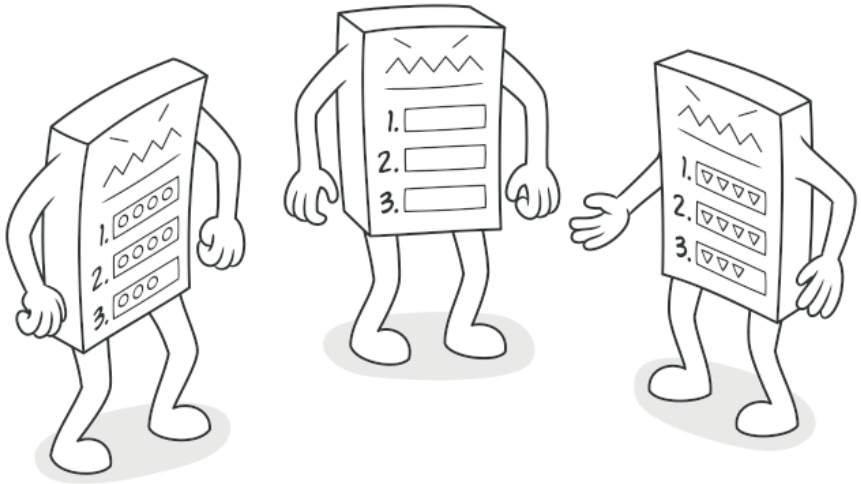
Lior Wunsch

Instructor: Tamir Kucherov, Design Patterns

Introduction

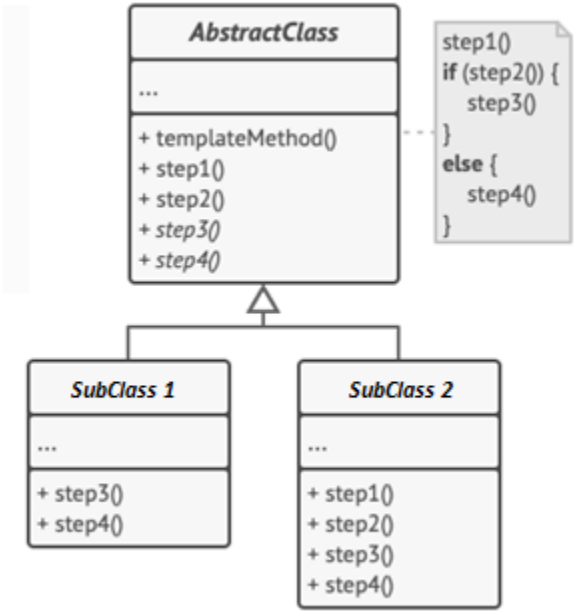
Template Method is a behavioral design pattern that allows you to defines a skeleton of an algorithm in a base class and let subclasses override the steps without changing the overall algorithm's structure.

Solving us code duplication for different classes who uses the same algorithms with some variations. Without changing the algorithms itself.



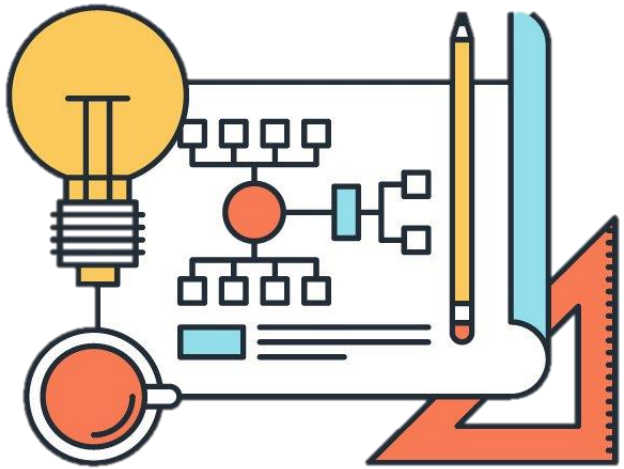
Structure

1. **Abstract Class** declares methods that act as steps of an algorithm, as well as the actual template method which calls these methods in a specific order.
2. **Sub Classes** can override all of the steps, but not the template method itself.

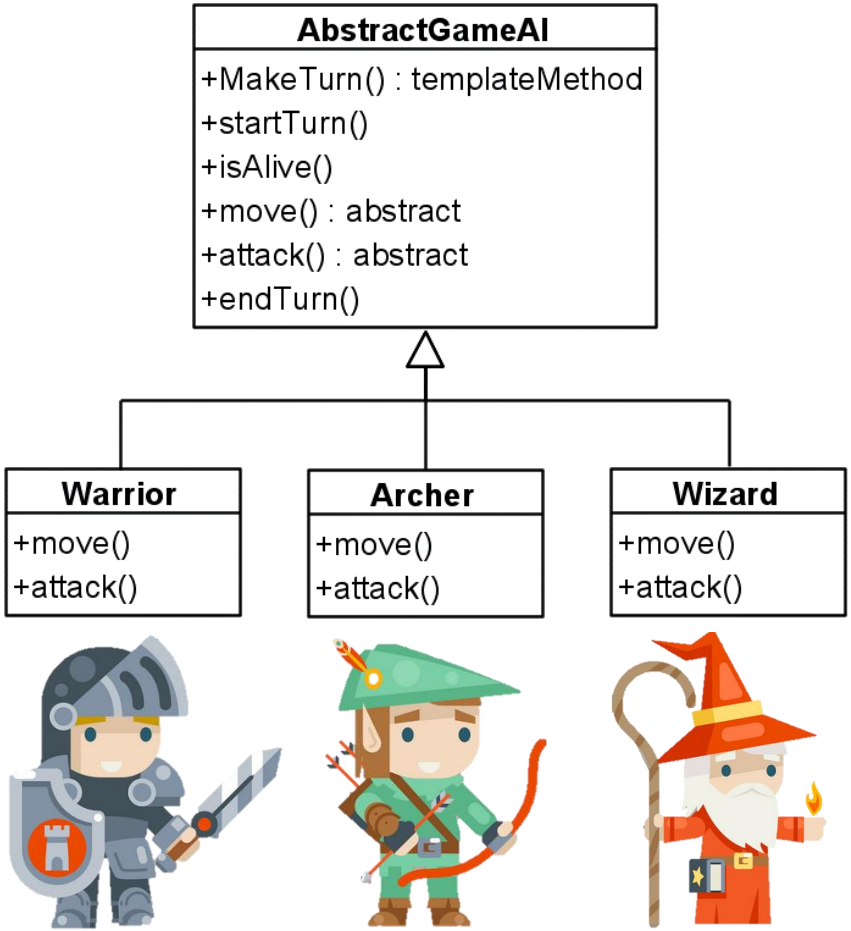


Description

- Break down an algorithm into a series of methods.
- Put a series of calls to these methods inside a single template method.
- The steps may either be **abstract** or have some **default** implementation.
- Developers provide their own subclasses, implement all abstract steps, and override some of the optional ones if needed (but not the template method itself).



UML


AbstractGameAI
+MakeTurn() : templateMethod
+startTurn()
+isAlive()
+move() : abstract
+attack() : abstract
+endTurn()
Warrior
+move()
+attack()
Archer
+move()
+attack()
Wizard
+move()
+attack()

Code

```java
public abstract class AbstractGameAI {
    protected int health = 5;
    protected String name;
    protected AbstractGameAI enemyAi;

    public AbstractGameAI(String characterName) {
        this.name = characterName;
    }

    public void setEnemyAi(AbstractGameAI enemyAi) {
        this.enemyAi = enemyAi;
    }

    public boolean isAlive() {
        return health > 0;
    }
```

```java
//  Template method
    public final void MakeTurn() {
        startTurn();
        if (isAlive()) {
            move();
            attack();
        }
        endTurn();
    }

//  abstract methods implemented by client
    public abstract void move();
    public abstract void attack();
```

```java
//  implementations of some steps are defined in base class
    private final void startTurn() {
        System.out.print("== " + name + " | ");
        System.out.print("HP:" + health + " ==\n");
    }

    private final void endTurn() {
        if (isAlive())
            System.out.println("==== End Turn ====\n");
        else
            System.out.println(name + " is Dead, RIP (x_x )");
    }
```

Code

```java
public class Warrior extends AbstractGameAI {

    public Warrior(String characterName) {
        super(characterName);
    }

    @Override
    public void move() {
        System.out.println("Move 1 Step Forward!");
    }

    @Override
    public void attack() {
        Random r = new Random();
        int damage = r.nextInt(5);
        enemyAi.health -= damage;
        System.out.println("Warrior - Inflicted " + damage + " Damage!");
    }
```

```java
public static void main(String[] args) {
    Warrior bob = new Warrior("Bob");
    Warrior alice = new Warrior("Alice");
    bob.setEnemyAi(alice);
    alice.setEnemyAi(bob);

    while (bob.isAlive() && alice.isAlive())
    {
        bob.MakeTurn();
        alice.MakeTurn();
    }

}
```

Output

```
== Bob | HP:5 ==
Move 1 Step Forward!
Warrior - Inflicted 4 Damage!
==== End Turn ====

== Alice | HP:1 ==
Move 1 Step Forward!
Warrior - Inflicted 1 Damage!
==== End Turn ====

== Bob | HP:4 ==
Move 1 Step Forward!
Warrior - Inflicted 3 Damage!
==== End Turn ====

== Alice | HP:-2 ==
Alice is Dead, RIP (x_x )
```

Pros and Cons

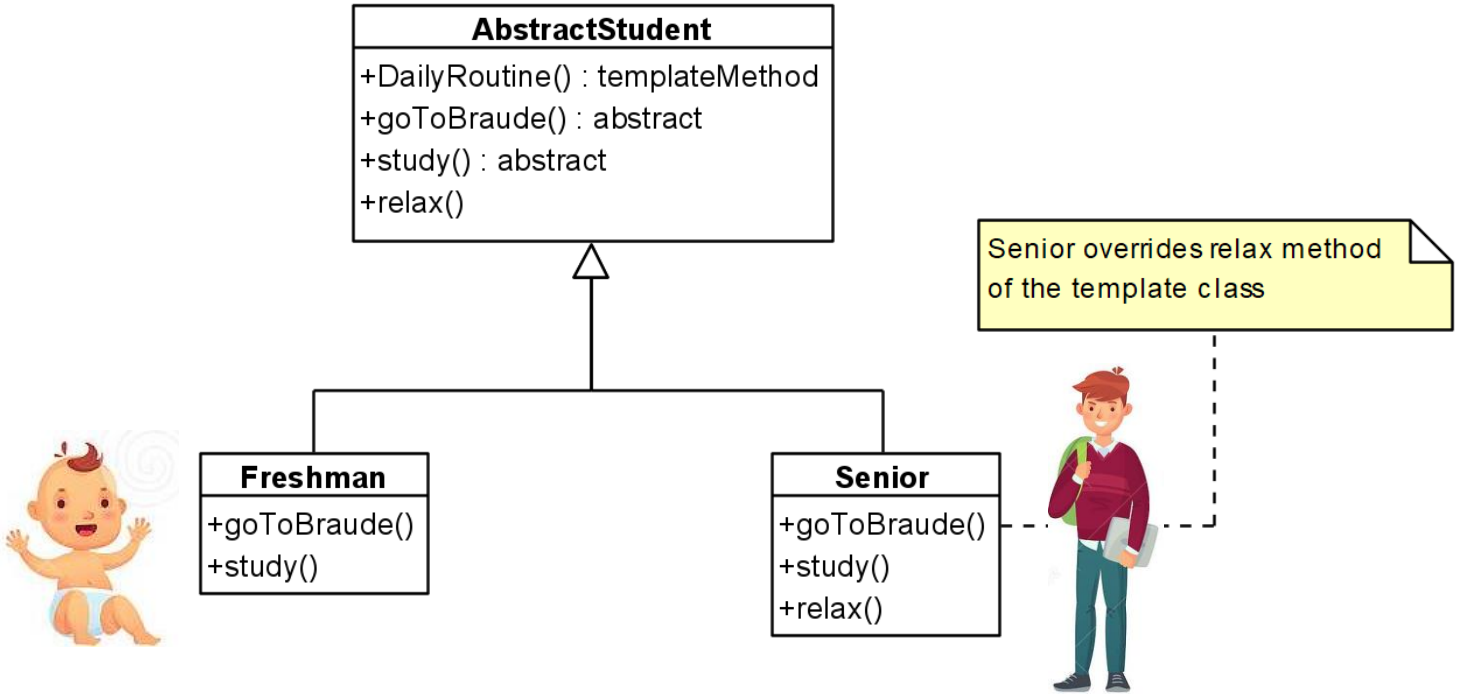
- ✓ You can let clients override only certain parts of a large algorithm, making them less affected by changes that happen to other parts of the algorithm.
- ✗ Some clients may be limited by the provided skeleton of an algorithm.
- ✗ Template methods tend to be harder to maintain the more steps they have.

Class Exercise 1

Add/Modify the code to support 2 additional character types:

Archer & Wizard, as seen for Warrior in the code example.

Class Exercise 2



Thank You