Le Pattern Template Method

I. Introduction

La technique du **patron de méthode** (*template method pattern*) est un patron de conception *(design pattern)* comportemental utilisé en génie logiciel.

Un patron de méthode définit le squelette d'un algorithme à l'aide d'opérations abstraites dont le comportement concret se trouvera dans les sous-classes, qui implémenteront ces opérations.

Cette technique, répandue dans les classes abstraites, permet de :

- Fixer clairement des comportements standards qui devraient être partagés par toutes les sous-classes, même lorsque le détail des sous-opérations diffère ;
- Factoriser le mode du code qui serait redondant s'il se trouvait répété dans chaque sous-classe.

II. Le Pattern Template Method

Le pattern template method est utilisé :

- Pour implémenter les parties invariantes d'un algorithme seule fois et laisser aux sous-classes de soin d'implémenter les parties qui varient.

- Lorsqu'un comportement commun entre des sous-classes devrait être factorisé et placé dans une classe commune afin d'éviter la duplication de code.

« Refactoring to generalize »

- Identifier les différences dans le code existant
- Extraire ces différences dans de nouvelles méthodes
- Utiliser le template method pour appeler ces méthodes

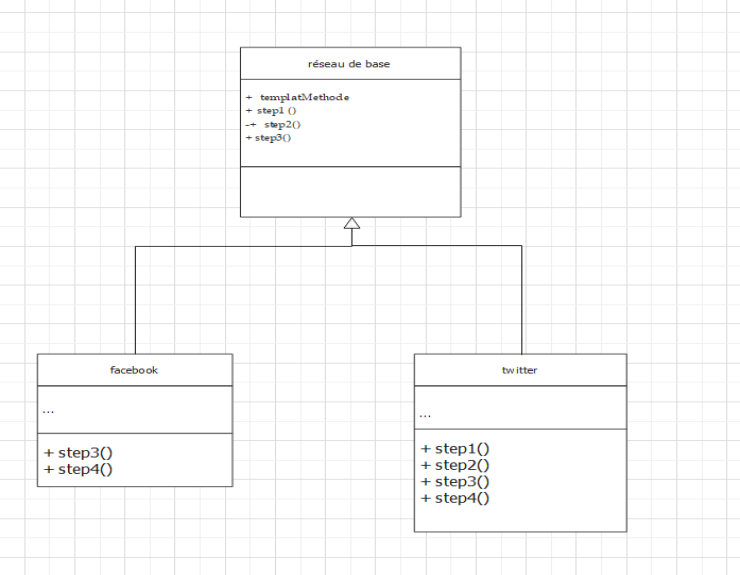
- Pour contrôler les extensions des sous-classes

- Définir une méthode template
  Qui appelle des méthodes « hook » en des points spécifiques de manières a ne permettre les extensions seulement qu'en ces points

Cas illustratif :

Dans cet exemple, le modèle De méthode de modèle définit un algorithme d'utilisation d'un réseau social. Les sous-classes qui correspondent à un réseau social particulier, implémentent ces étapes en fonction de l'API fournie par le réseau social.

Structure :



Démonstration :

Classe de réseau social de base :

_index.php | package refactoring_guru.template_method.example.n

```java
package refactoring_guru.template_method.example.networks;

/**
 * Base class of social network.
 */
public abstract class Network {
    String userName;
    String password;

    Network() {}

    /**
     * Publish the data to whatever network.
     */
    public boolean post(String message) {
        // Authenticate before posting. Every network uses a different
        // authentication method.
        if (logIn(this.userName, this.password)) {
            // Send the post data.
            boolean result =  sendData(message.getBytes());
            logOut();
            return result;
        }
        return false;
    }

    abstract boolean logIn(String userName, String password);
    abstract boolean sendData(byte[] data);
    abstract void logOut();
}
```

Réseau social concret :

```java
package refactoring_guru.template_method.example.networks;

/**
 * Class of social network
 */
public class Facebook extends Network {
    public Facebook(String userName, String password) {
        this.userName = userName;
        this.password = password;
    }

    public boolean logIn(String userName, String password) {
        System.out.println("\nChecking user's parameters");
        System.out.println("Name: " + this.userName);
        System.out.print("Password: ");
        for (int i = 0; i < this.password.length(); i++) {
            System.out.print("*");
        }
        simulateNetworkLatency();
        System.out.println("\n\nLogIn success on Facebook");
        return true;
    }

    public boolean sendData(byte[] data) {
        boolean messagePosted = true;
        if (messagePosted) {
            System.out.println("Message: '" + new String(data) + "' was posted on Facebook");
            return true;
        } else {
            return false;
        }
    }

    public void logOut() {
        System.out.println("User: '" + userName + "' was logged out from Facebook");
    }

    private void simulateNetworkLatency() {
        try {
            int i = 0;
            System.out.println();
            while (i < 10) {
                System.out.print(".");
                Thread.sleep(500);
                i++;
            }
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }
}
```

Un réseau social de plus :

```java
package refactoring_guru.template_method.example.networks;

/**
 * Class of social network
 */
public class Twitter extends Network {

    public Twitter(String userName, String password) {
        this.userName = userName;
        this.password = password;
    }

    public boolean logIn(String userName, String password) {
        System.out.println("\nChecking user's parameters");
        System.out.println("Name: " + this.userName);
        System.out.print("Password: ");
        for (int i = 0; i < this.password.length(); i++) {
            System.out.print("*");
        }
        simulateNetworkLatency();
        System.out.println("\n\nLogIn success on Twitter");
        return true;
    }

    public boolean sendData(byte[] data) {
        boolean messagePosted = true;
        if (messagePosted) {
            System.out.println("Message: '" + new String(data) + "' was posted on Twitter");
            return true;
        } else {
            return false;
        }
    }

    public void logOut() {
        System.out.println("User: '" + userName + "' was logged out from Twitter");
    }

    private void simulateNetworkLatency() {
        try {
            int i = 0;
            System.out.println();
            while (i < 10) {
                System.out.print(".");
                Thread.sleep(500);
                i++;
            }
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }
}
```

Code client :

```java
package refactoring_guru.template_method.example;

import refactoring_guru.template_method.example.networks.Facebook;
import refactoring_guru.template_method.example.networks.Network;
import refactoring_guru.template_method.example.networks.Twitter;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

/**
 * Demo class. Everything comes together here.
 */
public class Demo {
    public static void main(String[] args) throws IOException {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        Network network = null;
        System.out.print("Input user name: ");
        String userName = reader.readLine();
        System.out.print("Input password: ");
        String password = reader.readLine();

        // Enter the message.
        System.out.print("Input message: ");
        String message = reader.readLine();

        System.out.println("\nChoose social network for posting message.\n" +
                "1 - Facebook\n" +
                "2 - Twitter");
        int choice = Integer.parseInt(reader.readLine());

        // Create proper network object and send the message.
        if (choice == 1) {
            network = new Facebook(userName, password);
        } else if (choice == 2) {
            network = new Twitter(userName, password);
        }
        network.post(message);
    }
}
```

Résultat de l'exécution :

```
_index.php  ×  package refactoring_gu

Input user name: dieynaba
Input password: qswe
Input message: Hello, World!

Choose social network for posting message.
1 - Facebook
2 - Twitter
2

Checking user's parameters
Name: dieynaba
Password: ****
..........

LogIn success on Twitter
Message: 'Hello, World!' was posted on Twitter
User: 'dieynaba' was logged out from Twitter
```

Discussion :

- Analysez l'algorithme ciblé pour voir si vous pouvez le décomposer en étapes. Déterminez les étapes communes à toutes les sous-classes et celles qui sont uniques.
- Créez une classe de base abstraite et déclarez le patron de méthode et un ensemble de méthodes abstraites pour représenter les opérations de l'algorithme. Faites une ébauche de la structure de l'algorithme dans ce patron de méthode en appelant les opérations correspondantes. Rendez ce patron final pour empêcher les sous-classes de la redéfinir.
- Cela ne pose aucun problème si toutes les opérations sont abstraites, mais une implémentation par défaut bénéficierait à certaines opérations. Les sous-classes n'ont pas besoin d'implémenter ces méthodes.

- Pensez à ajouter des crochets entre les étapes cruciales de votre algorithme.
- Pour chaque variante de l'algorithme, créez une nouvelle sous-classe. Elle *doit* implémenter toutes les opérations abstraites, mais *peut* également redéfinir les opérations facultatives.

III. Avantages et inconvénients :

- Avantages :
    - Vous permettez aux clients de redéfinir certaines parties d'un grand algorithme. Elles sont ainsi moins affectées par les modifications apportées aux autres parties de l'algorithme.
    - Vous pouvez remonter le code dupliqué dans la classe mère.
- Inconvénients :
    - Certains clients peuvent être limités à cause du squelette de l'algorithme.
    - Vous ne respectez pas le *Principe de substitution de Liskov*, si vous supprimez l'implémentation d'une étape par défaut dans une sous-classe.
    - Plus vous avez d'étapes, plus le patron de méthode devient difficile à maintenir.

IV. Conclusion :

Le Patron de méthode est basé sur l'héritage : il vous laisse modifier certaines parties d'un algorithme en les étendant dans les sous-classes.