Dependency Inversion Principle

AMT

8 - Dependency Inversion Principle

Résumé du document

Cette note présente le principe d'inversion de dépendance, qui vise à réduire le couplage entre modules en les reliant à des abstractions. Elle met en avant l'importance de l'injection de dépendances pour améliorer la flexibilité et la testabilité des objets. Enfin, elle explique l'inversion de contrôle, où un framework gère la création et le flux des objets.

Table des matières

1. Dependency Inversion Principle

Les principes dit SOLID sont des principes de conception logicielle qui permettent de créer des logiciels plus maintenables, flexibles et évolutifs. Le principe de l'inversion de dépendance est le dernier principe de cette liste.

1.1. Définition

Le principe de l'inversion de dépendance est un principe de conception logicielle qui stipule que:

- Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau. Les deux doivent dépendre d'abstractions.
- Les abstractions ne doivent pas dépendre des détails. Les détails doivent dépendre des abstractions.

En utilisant le principe d'inversion de dépendance nous pouvons réduire le **couplage** entre les modules et ainsi augmenter leur **réutilisabilité**.

1.2. Exemple de dépendance

Dans cet exemple, la classe `Application` dépend directement de la classe `ConsoleLogger`. Cela va à l'encontre du principe de l'inversion de dépendance.

1.2.1. Sans Application de DIP

```
class ConsoleLogger {
    public void logMessage(String message) {
        System.out.println("Log message to
console: " + message);
    }
}

class Application {

    // The high-level Application  depends
directly on the low-level ConsoleLogger.
    private ConsoleLogger logger = new
ConsoleLogger();

    public void performOperation() {
        logger.logMessage("Performing an
operation");
    }
}

public class Main {

    public static void main(String[] args) {
        Application app = new Application();
        app.performOperation();
    }
}
```

1.2.2. Application de DIP

```
// The Logger illustrates the dependency
inversion principle (DIP)
interface Logger {
    void logMessage(String message);
}

// The low-level ConsoleLogger implements the
abstraction.
class ConsoleLogger implements Logger {
    public void logMessage(String message) {
        System.out.println("Log message to
console: " + message);
    }
}

class Application {

    // The high-level Application now depends
on the abstraction.
    private Logger logger;

    // The dependency is injected via the
constructor.
    public Application(Logger logger) {
        this.logger = logger;
    }

    public void performOperation() {
        logger.logMessage("Performing an
operation");
    }
}

public class Main {
    public static void main(String[] args) {
        Application app = new Application(new
ConsoleLogger());
        app.performOperation();
    }
}
```

2. Dependency Injection

Le principe de l'injection de dépendance est une technique qui permet de respecter le principe de l'inversion de dépendance. L'injection de dépendance consiste à fournir les dépendances d'un objet à l'extérieur de cet objet. Cela permet de rendre les objets plus flexibles et plus facilement testables.

2.1. Roles

L'injection de dépendance implique trois rôles principaux:

- **Service**: est un module de **bas** niveau fournissant une fonctionnalitées
- **Client**: est un module de **haut** niveau qui dépend du service
- **Injector**: est un module qui crée et fournit les dépendances aux clients

2.1.1. Types d'injection de dépendance

Il existe trois types d'injection de dépendance:

2.1.2. Constructeur

les dépendances sont fournies via le constructeur

```
// The Application correspond to the client.
class Application {

    // The Logger correspond to the service.
    private Logger logger;

    // The dependency to the service is
injected via a constructor.
    public Application(Logger logger) {
        this.logger = logger;
    }
}
```

2.1.3. Setter

les dépendances sont fournies via des setters

```
// The Application correspond to the client.
class Application {

    // The Logger correspond to the service.
    private Logger logger;

    // The dependency to the service is
injected via a setter.
    public void setLogger(Logger logger) {
        this.logger = logger;
    }
}
```

2.1.4. Interface

les dépendances sont fournies via une interface

```
// The Service provides an interface with an
injector method.
interface LoggerSetter {
    void setLogger(Logger logger);
}

// The Client implements the interface.
class Application implements LoggerSetter {

    private Logger logger;

    // The dependency is injected via the
setter.
    public void setLogger(Logger logger) {
        this.logger = logger;
    }
}

// An injector injects the dependency to the
client.
class Injector {
    public void injectLogger(LoggerSetter
client) {
        client.setLogger(new ConsoleLogger());
    }
}
```

2.2. Realité de l'injection de dépendance

Aujourd'hui, grâce aux différents frameworks de développement, l'injection de dépendance est devenue une pratique courante. Les frameworks permettent de gérer automatiquement les dépendances des objets.

```
// The Application correspond to the client.
class Application {

    // The annotation acts as an interface and makes the injector aware of the dependency.
    @Inject
    private Logger logger;

}
```

3. Inversion of Control

L'inversion de contrôle (IoC) est un principe de conception qui stipule que le contrôle du flux d'un programme est inversé. Dans un programme traditionnel, le flux du programme est contrôlé par le programme lui-même. Dans un programme IoC, le flux du programme est contrôlé par un framework.

3.1. Exemple

Dans cet exemple, le framework Spring contrôle le flux du programme. Le framework est responsable de la création et de la gestion des objets.

```
// The Application corresponds to the client.
@ApplicationScoped
class Application {

    // The Logger corresponds to the service.
    @Inject
    private Logger logger; // Dependency injection via @Inject

    public void performOperation() {
        logger.log("Performing operation...");
    }
}

// Interface for Logger
interface Logger {
    void log(String message);
}

// Implementation of Logger
@ApplicationScoped
class ConsoleLogger implements Logger {
    @Override
    public void log(String message) {
        System.out.println("Console Logger: " + message);
    }
}

// Main class
public class Main {

    public static void main(String[] args) {
        // Use the Quarkus CDI container to retrieve the Application instance
        // ...
    }
}
```