ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

EXERCISE 03

DEPENDENCY INVERSION PRINCIPLE AND DEPENDENCY INJECTION

SOFTWARE ANALYSIS AND DESIGN

21120280 – Lý Minh Khuê

Thành phố Hồ Chí Minh – 2024

TABLE OF CONTENTS

CONCEPT OF DIP

1. Dependency Inversion Principle concept:

Dependency Inversion Principle (DIP) is one part of the **SOLID** – a set of 5 principles in design(**not exclusively** in software), each letter stands for a specific principle.

DIP suggests that heigh-level modules should not depend on the lower ones. Instead, both should be only dependent upon the abstraction of the class rather than the details(concrete implementations). Hard to understand? Let's breaking it down to simpler terms:

- High-level modules should not import anything from low-level modules. Both should depend on abstractions (e.g., interfaces).
- Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.

DIP was introduced to solve the issue around tight couplings between modules, which makes the system brittle and less flexible to adapt any changes. **DIP** tells us to not directly impact other modules when applying changes in one. That can be achieved by implementing abstraction between layers, which allows the modules to be independent and interchangeable. Thus, the system is more robust and less prone to fail when swapping out components or extend functionality.

However, there is some misunderstandings roaming around when implementing **DIP**. We should not be confused of what makes **SRP** and **DIP**(both are parts of **SOLID** principles). While **SRP** focuses on ensuring that a module or class has only one reason to change, **DIP** complements this by guiding how modules should depend on each other. **SRP** deals with the internal responsibilities of a module, while **DIP** addresses the external dependencies of a module.

2. Examples:

- **Without DIP:**

```java
// Gas engine class
2 usages  new *
class GasEngine {
    1 usage  new *
    public void start() {
        System.out.println("Starting the car with a gas engine.");
    }
}

// Electric engine class
2 usages  new *
class ElectricEngine {
    1 usage  new *
    public void start() {
        System.out.println("Starting the car with an electric engine.");
    }
}
```

Figure 1. GasEngine and ElectricEngine class

```java
// Car class directly depends on concrete implementations of Engine
4 usages  new *
class Car {
    2 usages
    private final GasEngine gasEngine;
    2 usages
    private final ElectricEngine electricEngine;

    2 usages  new *
    public Car() {
        this.gasEngine = new GasEngine();
        this.electricEngine = new ElectricEngine();
    }

    1 usage  new *
    public void startWithGasEngine() {
        gasEngine.start();
    }

    1 usage  new *
    public void startWithElectricEngine() {
        electricEngine.start();
    }
}
```

Figure 2. Implementation of Car

The **Car** class directly depends on concrete implementations of the **Engine** (specifically, **GasEngine** and **ElectricEngine**). This violates the DIP because high-level modules (Car) depend on low-level modules (Engine implementations) rather than using any abstractions.
This design makes the **Car** class coupled to specific engine implementations tightly. Further change in the engine implementation would require modifications to the **Car** class. It reduces flexibility, as it's not easy to swap out engine implementations without modifying the **Car** class.

```
// Main class
new *
class Main {
    new *
    public static void main(String[] args) {
        Car gasCar = new Car();
        gasCar.startWithGasEngine(); // Output: Starting the car with a gas engine.

        Car electricCar = new Car();
        electricCar.startWithElectricEngine(); // Output: Starting the car with an electric engine.
    }
}
```

Figure 3. Without DIP Main class

The implementation above is an example of bad design when Car class heavily depends on the low-level classes like GasEngine and ElectricEngine, which means when we apply any changes to either of these classes, the Car class will do the same. This usage violates Open/Closed Principle, making the whole system hard to maintain or evolve over time.

- **With DIP:**

We can implement the same functionality of the above code with a more flexible and robust approach using interface and abstraction.

```
6 usages  2 implementations  new *
interface Engine {
    1 usage  2 implementations  new *
    void start();
}
```

Figure 4. Interface Engine

By defining '**Engine**' as an interface, it allows for multiple types of engines to be created (e.g., gas-powered engine, electric engine) while ensuring that they all provide the required functionality (in this case, the ability to start the engine).
The use of this interface in this scenario encourage loose couplings between classes and concrete implementations

```
1 usage  new *
class GasEngine implements Engine {
    1 usage  new *
    @Override
    public void start() {
        System.out.println("Starting the car with a gas engine.");
    }
}

1 usage  new *
class ElectricEngine implements Engine {
    1 usage  new *
    @Override
    public void start() {
        System.out.println("Starting the car with an electric engine.");
    }
}
```

Figure 5. GasEngine and ElectricEngine classes with DIP

```java
4 usages  new *
class Car {
    2 usages
    private final Engine engine;

    2 usages  new *
    public Car(Engine engine) {
        this.engine = engine;
    }

    2 usages  new *
    public void start() {
        engine.start();
    }
}
```

Figure 6. Car class with DIP

The high-level ‘Car’ class only depends on the ‘Engine’ interface, rather than the details like ‘GasEngine’nor ‘ElectricEngine’. All of those classes depend on the same interface ‘Engine’, so future changes of different types of engines would not place any impacts on the Car class.

```java
new *
class Main {
    new *
    public static void main(String[] args) {
        // Creating a gas-powered car
        Engine gasEngine = new GasEngine();
        Car gasCar = new Car(gasEngine);
        gasCar.start(); // Output: Starting the car with a gas engine.

        // Creating an electric car
        Engine electricEngine = new ElectricEngine();
        Car electricCar = new Car(electricEngine);
        electricCar.start(); // Output: Starting the car with an electric engine.
    }
}
```

Figure 7. Main class with DIP

IMPORTANCE OF DIP

While we are writing code, we are likely to use multiple principles and pattern designs which are best practices. We might split our codebase into multiple modules or classes. However, this is where the implementation turns into a whole mess with lots of bugs and vulnerabilities after violating other principles when trying to focusing on one. For this instance, The modules will highly depend on dependencies, which makes the system stagger and the future change cost would be huge. **DIP** motivation is to achieve loosely coupling with the aim of prevent us from depending too much upon modules that are likely to change.

Understanding the Dependency Inversion Principle (DIP) provides a clearer idea how to inject interfaces into other components. This involves injecting interfaces through a class constructor, which proves useful in scenarios like testing, where we can use fake implementations of dependencies as interface mocks.

However, isn't this just Dependency Injection (DI)?

Imagine we're dealing with tightly coupled classes. How can we make it more manageable and loosely coupled classes?

FEATURE	With DIP	Without DIP
Developing different components	Easy	Hard to test due to class dependencies
Testing	Easy to test in isolation	Hard to test due to tight coupling between classes
Extending components	Easy to extend	Hard to extend as classes are tightly coupled
Deploying parts of the system	Easy to independently deploy parts of the system	Need to recompile all the software for a small fix
Merging branches of work	Easy to merge branches as changes are isolated	Hard to merge branches as code has dependencies

DEPENDENCY INJECTION AND IOC

Yes, **Dependency Injection(DI)** is also included in what so-called **DIP**, but **DIP** is not just about **DI**. Better explanation is that DI is the tool to achieve the inversion.
Yet, We can't instantiate interfaces, we can only rely on concrete classes. But creating a concrete object must be processed, which should prevent unwanted dependency when instantiating that object.

With **Inversion of Control (IoC)** and **Factory Design Pattern** will help us in this matter. In **IoC**, the creation of dependencies is shitfted to a container, that manage the whole process of creating and preserving the lifecycle of the objects, while traditional programming uses a component to do all the work, binding a tight coupling between classes.

Using Java, a OOP language would explain this well, specifically using Spring boot DI frameworks.

- Container: The IoC container in Spring manages the instantiation, configuration, and assembly of objects (beans). It creates objects, wires them together, configures them, and manages their complete lifecycle.

- Beans: In Spring, an object that is managed by the IoC container is called a bean. Beans are defined in the Spring configuration file (XML or JavaConfig) or through annotations.

- Autowired: Marks a constructor, field, setter method, or config method as to be autowired by Spring's dependency injection facilities.

```
9 usages  kuwelym
@Repository
public interface ProductRepository extends JpaRepository<Product, Long> {
    1 usage  kuwelym
    Boolean existsByNameAndCategoryId(String name, Long categoryId);
}
```

Figure 8. ProductRepository class

```java
public interface ProductService {
    List<ProductDTO> findAllProducts();
    ProductDTO findProductById(Long id);
    ProductDTO createProduct(String name, Double price, Long categoryId, String description);
    ProductDTO updateProduct(Long id, String name, Double price, Long categoryId, String description);
    ResponseEntity<?> deleteProduct(Long id);
}
```

Figure 9. ProductService interface

```java
@Service
public class ProductServiceImpl implements ProductService {
    private final ProductRepository productRepository;
    private final CategoryRepository categoryRepository;

    @Autowired
    public ProductServiceImpl(ProductRepository productRepository, CategoryRepository categoryRepository) {
        this.productRepository = productRepository;
        this.categoryRepository = categoryRepository;
    }

    @Override
    public List<ProductDTO> findAllProducts() {
        List<Product> products = productRepository.findAll();
        return products.stream().map((ProductServiceMapper::toProductDTO)).collect(Collectors.toList());
    }

    @Override
    public ProductDTO findProductById(Long id) {
        return productRepository.findById(id).map(ProductServiceMapper::toProductDTO).orElse(null);
    }
```

Figure 10. ProductServiceImpl class

In this example, **ProductServiceImpl** depends on **ProductRepository** and **CategoryRepository**, and Spring will inject an instance of **ProductRepository** and **CategoryRepository** into **ProductServiceImpl** during application startup. This allows **ProductServiceImpl** to use the functionality provided by **UserRepository** without directly instantiating it.

```java
kuwelym *
@RequestMapping("/api/v1")
@RestController
public class ProductController {
    6 usages
    private final ProductService productService;
    4 usages
    private final AuthorizationUtil authorizationUtil;

    kuwelym *
    @Autowired
    public ProductController(ProductService productService, AuthorizationUtil authorizationUtil) {
        this.productService = productService;
        this.authorizationUtil = authorizationUtil;
    }

    new *
    @GetMapping("/products")
    public ResponseEntity<?> getProducts(@RequestHeader(value = "If-None-Match", required = false) String ifNoneMatch) {
        List<ProductDTO> products = productService.findAllProducts();
        if (products.isEmpty()) {
            return ResponseEntity.status(HttpStatus.NOT_FOUND)
                    .body("No products found");
        }

        String eTag = calculateETagForProducts(products);
        if (ifNoneMatch ≠ null && ifNoneMatch.equals(eTag)) {
            return ResponseEntity.status(HttpStatus.NOT_MODIFIED)
                    .eTag(eTag)
                    .body(null);
        }
        return ResponseEntity.ok()
                .eTag(eTag)
                .body(products);
    }
```

Figure 11. ProductController class

Similarly, in **ProductController**, both **ProductService** and **AuthorizationUtil** are injected via constructor injection.

In the codebase, Spring's IoC container manages the instantiation of **ProductServiceImpl**, **ProductController**, and other Spring-managed beans by injecting dependencies into classes rather than creating them within the class, DI promotes loose coupling between components.

Furthermore, this implementation seperates the abstraction level and concrete level of the system. For instance, **ProductServiceImpl** is a concrete implementation of the **ProductService** interface. It contains the actual business logic for handling product-related operations.

The separation of interfaces and concrete implementations adheres to the Dependency Inversion Principle. In this case, high-level modules such as **ProductServiceImpl** and **ProductController** depend on abstractions (interfaces), while the actual implementations are injected at runtime.

REFERENCES

https://stackify.com/dependency-inversion-principle/
https://viblo.asia/p/nguyen-tac-thu-nam-trong-solid-the-dependency-inversion-principle-ORNZqXwMK0n#_factory-1
https://www.baeldung.com/cs/dip
https://www.geeksforgeeks.org/dependecy-inversion-principle-solid/