Reducing Coupling in Software Design

Affan Rauf

September 20, 2024

What is Coupling?

- Coupling refers to the degree of direct knowledge one class has of another.
- Tight coupling occurs when classes are highly dependent on each other.
- Loose coupling reduces dependencies between components.

What is Coupling?

- Coupling refers to the degree of direct knowledge one class has of another.
- Tight coupling occurs when classes are highly dependent on each other.
- Loose coupling reduces dependencies between components.

Goal: Achieve loose coupling to create flexible, maintainable, and testable software.

Abstract Classes to Reduce Coupling

- An abstract class defines a template for its subclasses without providing full implementations.
- Subclasses provide the actual implementation details.
- Abstract classes allow code reuse while reducing dependencies on concrete classes.

Abstract Classes to Reduce Coupling

- An abstract class defines a template for its subclasses without providing full implementations.
- Subclasses provide the actual implementation details.
- Abstract classes allow code reuse while reducing dependencies on concrete classes.

Example:

```
abstract class Shape {
    abstract void draw();
}
class Circle extends Shape {
    void draw() { System.out.println("Drawing Circle"); }
}
```

Interfaces for Loose Coupling

- Interfaces define a contract without any implementation.
- Classes can implement multiple interfaces, promoting flexibility.
- Depend on abstractions (interfaces) rather than concrete implementations.

Interfaces for Loose Coupling

- Interfaces define a contract without any implementation.
- Classes can implement multiple interfaces, promoting flexibility.
- Depend on abstractions (interfaces) rather than concrete implementations.

Example:

```
interface Drawable {
    void draw();
}
class Rectangle implements Drawable {
    public void draw() { System.out.println("Drawing Rectangle")
}
```

Dependency Injection (DI)

- DI is a design pattern where dependencies are "injected" into a class rather than the class instantiating them.
- Promotes loose coupling by making the code independent of the creation process of dependencies.

Dependency Injection (DI)

- DI is a design pattern where dependencies are "injected" into a class rather than the class instantiating them.
- Promotes loose coupling by making the code independent of the creation process of dependencies.

Example: Constructor Injection

```
class Service {
    private final Repository repo;
    public Service(Repository repo) {
        this.repo = repo;
    }
}
```

Dependency Injection (DI)

- DI is a design pattern where dependencies are "injected" into a class rather than the class instantiating them.
- Promotes loose coupling by making the code independent of the creation process of dependencies.

Example: Constructor Injection

```
class Service {
    private final Repository repo;
    public Service(Repository repo) {
        this.repo = repo;
    }
}
```

Benefit: Service is not tightly coupled to a specific Repository implementation.

Dependency Inversion Principle (DIP)

- Part of SOLID principles: Depend on abstractions, not on concrete classes.
- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

Dependency Inversion Principle (DIP)

- Part of SOLID principles: Depend on abstractions, not on concrete classes.
- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

Example:

```
interface Repository {
    void saveData(String data);
}
class DatabaseRepo implements Repository {
    public void saveData(String data) {
        // Save to DB
    }
}
```

Service uses the ‘Repository‘ interface, not the ‘DatabaseRepo‘ directly, reducing coupling.

How DI and DIP Reduce Coupling

- **Dependency Injection:** Moves the responsibility of dependency creation out of the class, making it more flexible and testable.
- **Dependency Inversion:** High-level code depends on abstractions, not concrete implementations, making changes easier.

How DI and DIP Reduce Coupling

- **Dependency Injection:** Moves the responsibility of dependency creation out of the class, making it more flexible and testable.
- **Dependency Inversion:** High-level code depends on abstractions, not concrete implementations, making changes easier.

Key Result: Systems become more modular, maintainable, and adaptable to change.

Summary

- Reducing coupling makes the system more flexible and maintainable.
- Abstract classes and interfaces define contracts and reduce dependencies on concrete implementations.
- Dependency Injection separates creation from usage, reducing tight coupling.
- Dependency Inversion Principle promotes reliance on abstractions, decoupling high-level and low-level modules.