COMP 3550

6.3 — DEPENDENCY INJECTION: WHAT, WHY, AND HOW

Week 6: Alternatives to Inheritance & Dependency Injection

WHAT IS DEPENDENCY INJECTION (DI)?

We have actually already seen it (when we did SOLID)

- **What:** A class says what it needs, not how to get it
- **Why:** Keeps classes loosely coupled and easier to test
- **How:** Most common = constructor injection for clarity and immutability

```
// Old way – tightly coupled
class PaymentProcessor {
    private PaymentMethod method = new CreditCardPayment();
}

// DI way – loosely coupled
class PaymentProcessor {
    private PaymentMethod method;
    PaymentProcessor(PaymentMethod method) {
        this.method = method;
    }
}
```

RECAP OF THE “NEW” PROBLEM

Every new is a hidden chain.

- *new* inside a class hardcodes the dependency
- Makes it impossible to swap implementations without editing the class
- Breaks testability (can’t inject mocks/fakes)
- Creates rigid code that’s harder to maintain

```
class PaymentProcessor {
    private PaymentMethod method;

    PaymentProcessor() {
        this.method = new CreditCardPayment(); // locked in
    }
}
```

REFACTOR EXAMPLE

```java
class UserManager {
    private EmailService emailService = new EmailService();

    void registerUser(String email) {
        // ... user creation logic ...
        emailService.sendWelcomeEmail(email);
    }
}
```

REFACTOR EXAMPLE

```
class UserManager {
    private EmailService emailService;

    UserManager(EmailService emailService) {
        this.emailService = emailService;
    }

    void registerUser(String email) {
        // ... user creation logic ...
        emailService.sendWelcomeEmail(email);
    }
}
```

UserManager is still tied to the concrete EmailService. What if we made it even more flexible, so it could send messages in any way, without caring if it’s email, SMS, or something else?

EMAILSERVICE TO SENDABLE INTERFACE

```java
interface Sendable {
    void send(String message,
              String recipient);
}

class EmailService implements Sendable {
    public void send(String message,
                     String recipient) {
        // send email
    }
}

class SmsService implements Sendable {
    public void send(String message,
                     String recipient) {
        // send SMS
    }
}
```

```java
class UserManager {
    private Sendable sender;

    UserManager(Sendable sender) {
        this.sender = sender;
    }

    void registerUser(String email) {
        sender.send("Welcome!", email);
    }
}
```

BENEFITS OF DEPENDENCY INJECTION (RECAP)

- **Better Testing**
  - Swap real dependencies for mocks/fakes in unit tests
  - No external systems or side effects
  - Example: new UserManager(new MockSender())
- **Cleaner Wiring**
  - Classes don’t create their own dependencies, external code wires them together
  - Can be done manually (factory methods) or via DI frameworks (Spring, Guice)
  - Central place to change configurations

BEST PRACTICES & PITFALLS IN DEPENDENCY INJECTION

Best Practices

- Prefer constructor injection
  - makes dependencies explicit & ensures immutability.
- Inject abstractions, not concretions
  - keeps code flexible.
- Group related dependencies into a single service
  - avoids “too many parameters” smell.
- Use DI frameworks wisely
  - let them wire, keep logic in code.

BEST PRACTICES & PITFALLS IN DEPENDENCY INJECTION

Best Practices

- Prefer constructor injection
  - makes dependencies explicit & ensures immutability.
- Inject abstractions, not concretions
  - keeps code flexible.
- Group related dependencies into a single service
  - avoids “too many parameters” smell.
- Use DI frameworks wisely
  - let them wire, keep logic in code.

Pitfalls

- Over-injection
  - dozens of dependencies in one class = bad design smell.
- Setter injection as default
  - allows objects to exist in invalid states.
- Injecting config data everywhere
  - pass through a single configuration object.
- Treating DI as magic
  - you still need to understand object lifetimes.

PROJECT PAUSE & REFLECT

Pick one class in your project and refactor it to receive its dependency via constructor injection.

- Haven’t applied this to your Database Manager / Connection yet?
  - Now’s the perfect time.
- Imagine swapping your real database for:
  - A mock/fake database for testing
  - A completely different database model (e.g., HSQLDB to MongoDB, with a little extra rework but still cool to imagine!).

- **Goal**: Make your code swappable, testable, and flexible, no new inside!