Mohammadmahdi Farrokhy

Clean Code:

A Handbook of Agile Software Craftsmanship

Chapter 09: Systems

Sub-Systems Isolation

A software system is composed of distinct sub-systems and layers, each handling specific tasks. Crucially, these sub-systems must be isolated from one another. The client method constructs the required objects for the system and hands them over to the application layer, which utilizes them.

Dependency Injection and Inversion of Control (IoC)

A potent technique to segregate creation from application logic is dependency injection. This involves employing inversion of control (IoC) in dependencies management. With dependency inversion, we can delegate secondary responsibilities from one object to others that are specifically designed for it, leading to compliance of SRP. In the dependency handling concept, an object should not bear the responsibility of instantiating its own dependencies. Instead, this task must be entrusted to a different, influential mechanism that inverts the control. Typically, this powerful mechanism is either the main routine or a single purpose container, considering the setup as a global concern.

Dependency Injection and Inversion of Control (IoC)

```java
public interface MessageSender {
    public void sendMessage(String recipient, String message);
}
```

```java
public class EmailSender implements MessageSender {
    @Override
    public void sendMessage(String recipient, String message) {
        System.out.println("Sending email to " + recipient + ": " + message);
    }
}
```

```java
public class SMSSender implements MessageSender{
    @Override
    public void sendMessage(String recipient, String message) {
        System.out.println("Sending SMS to " + recipient + ": " + message);
    }
}
```

```java
public class User {
    private String contactInfo;

    public User(String contactInfo) {
        this.contactInfo = contactInfo;
    }

    public String getContactInfo() {
        return contactInfo;
    }
}
```

Dependency Injection and Inversion of Control (IoC)
BAD CODE

```
public class UserController {
    private String communicationMethod;
    private EmailSender emailSender = new EmailSender();
    private SMSSender smsSender = new SMSSender();

    public UserController(String communicationMethod) {
        this.communicationMethod = communicationMethod;
        emailSender = new EmailSender();
        smsSender = new SMSSender();
    }

    public void registerUser(User user) {
        if (communicationMethod.equals("Email")) {
            emailSender.sendMessage(user.getContactInfo(),
                    "Welcome to our platform!");
        }
        else if (communicationMethod.equals("SMS")) {
            smsSender.sendMessage(user.getContactInfo(),
                    "Welcome to our platform!");
        }
    }
}
```

Dependency Injection and Inversion of Control (IoC)
GOOD CODE

```
public class UserController {
    private MessageSender messageSender;

    public UserController(MessageSender messageSender) {
        this.messageSender = messageSender;
    }

    public void registerUser(User user) {
        messageSender.sendMessage(user.getContactInfo(),
                "Welcome to our platform!");
    }
}
```

Dependency Injection

```
public class Client {
    public void operation() {
        UserController userController = new UserController(new EmailSender());
        User user = new User("user@email.com");
        userController.registerUser(user);
    }
}
```

Domain-Specific Languages (DSL)

DSLs are customized languages tailored to a particular domain, enabling concise and precise communication between developers and stakeholders. DSLs allow for the expression of complex business rules and concepts in such way that is easily understood by non-technical experts. Creating a DSL, creates a software development environment more aligned with the problem space. Incorporating DSLs in the development process enhances collaboration and reduces the gap between technical and non-technical team members. It enables domain experts to actively participate in shaping the software's behavior, resulting in more accurate implementations. DSLs should be expressive, focused on the domain's core concepts, and designed to provide clarity and maintainability. Utilizing DSLs empowers teams to build systems that are closely aligned with the domain's intricacies and requirements.

Conclusion

Systems must also be clean. An aggressive architecture obscures domain logic and affects agility. When the domain's logic becomes ambiguous, the quality will suffer, because it will be easier for the bugs to hide and it will be more difficult to implement the customer's requirements.

If agility is compromised, productivity will suffer and the benefits of TDD will be lost. At all levels of abstraction, the goal must be clear. Regardless of whether you are designing a system or individual modules, never forget to use the simplest things that work for your needs.