Clean Code:

A Handbook of Agile Software Craftsmanship

Chapter 11: 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 main method constructs the required objects for the system and hands them over to the application layer, which utilizes them. The workings of the main method and the object creation process remains unknown to application layer.

Dependency Injection and Inversion of Control (IoC)

A potent technique for segregating building from application logic is dependency injection. This involves employing inversion of control (IoC) in managing dependencies. With dependency inversion, we can delegate secondary responsibilities from one object to others that are specifically designed for the related purpose, thereby upholding the single responsibility principle. 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)

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

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

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

Dependency Injection and Inversion of Control (IoC)

```
public class User {
    private String contactInfo;

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

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

```
public class UserController {
    private MessageSender messageSender;
    private EmailSender emailSender = new EmailSender();
    private SMSSender smsSender = new SMSSender();

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

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

Dependency Injection and Inversion of Control (IoC)

Dependency Injection

```
public class Main {
    public static void main(String[] args) {
        UserController userController = new UserController(new EmailSender());
        User user = new User("user@email.com");
        userController.registerUser(user);
    }
}
```

Scaling Up

It's highly improbable that a system gets initialed supporting all the requirements of the customer. We start by implementing the immediate requirements and then refactor the code to expand the system and add new features. The prerequisite to do that is to design a simple loosely coupled system. This approach is not only economically efficient for debugging and code modification but also facilitates system growth by seamlessly integrating new sub-systems. Some of huge software systems in the world involve different layers like caching, security, virtualization etc. It was possible, only because each sub-system in these systems are minimally coupled and easy to differentiate from the others.

Optimized Decision Making

In software design, decision optimization refers to the practice of making well-informed choices by delaying decisions until the most relevant information becomes available. Premature decisions, made without adequate knowledge, can result in suboptimal outcomes for a project. On the other hand, deferring decisions allows for optimal choices based on evolving project needs and feedback from stakeholders. This adaptability is achieved through modular systems with clear boundaries, enabling on-the-fly decisions that align with changing circumstances. This flexibility, stemming from separated concerns, facilitates agile decision-making, reducing complexity in software development. Ultimately, the key takeaway is that decision optimization enhances software's adaptability and agility by ensuring choices are made with accurate and up-to-date information.

Standards

Standards in software development refer to established guidelines and conventions that shape code design. They ensure consistency, streamline collaboration, and reduce errors. Adhering to standards eases onboarding for new developers, automates enforcement, and improves documentation practices. Ultimately, standards lead to clearer, more maintainable, and higher-quality codebases.

Domain-Specific Languages (DSL)

DSLs are specialized 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 a way that is easily understandable by non-technical experts. By creating a DSL that mirrors the language of the domain, software development becomes 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. The chapter stresses that DSLs should be expressive, focused on the domain's core concepts, and designed to provide clarity and maintainability. Ultimately, 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 logic of the domain becomes vague and blurred, 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 demands. 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.