SOLID چیست؟

SOLID چیست؟

از ۵ اصل تشکیل شده است

1: Single Responsibility Principle (SRP)

2: Open/Closed Principle (OCP)

3: Liskov Substitution Principle (LSP)

4: Interface Segregation Principle (ISP)

5: Dependency Inversion Principle (DIP)

هدف از مطرح شدن SOLID چیست؟

این ۵ اصل، با هدف طراحی و ایجاد سیستم های نرم‌افزاری مطرح شده است و کمک می کند که سیستم مورد نظر به صورت شفاف، خوانا، قابل فهم و انعطاف‌پذیر در مقابل توسعه های آینده و قابل نگهداری در برابر تغییرات در برنامه‌نویسی شیءگرا (OOP) حضور داشته باشد. این اصول به توسعه‌دهندگان کمک می‌کند تا سیستم های نرم افزاری قوی تر و با کیفیت بالاتری را تولید کنند.

با پیروی از این ۵ اصل به کدهایی خواهیم رسید که:

۱. **قابل نگهداری‌تر باشند:**

- با استفاده از طراحی منظم و واضح، تغییرات در یک بخش از کد، تأثیری منفی بر دیگر بخش‌ها نداشته باشد.

۲. **قابل گسترش باشند:**

- قابلیت اضافه کردن ویژگی‌های جدید، بدون نیاز به تغییر کدهای موجود را فراهم کنند.

۳. **قابل استفاده مجدد باشند:**

- با به کارگیری این اصول، کلاس‌ها و ماژول‌ها به گونه‌ای طراحی میشوند که بتوانند در پروژه‌های مختلف یا بخش‌های مختلف یک پروژه مورد استفاده قرار گیرند.

٤. **کاهش وابستگی‌ها :**

- از طریق استفاده از انتزاع‌ها و رابط‌ها(Interface & Abstract)، وابستگی‌های بین اجزای مختلف سیستم کاهش یابد.

در این مقاله اصل اول Single Responsibility Principle (SRP) را مطالعه خواهیم کرد.

اصل ۱: Single Responsibility Principle (SRP)

این اصل داره به این مفهوم اشاره می کنه که، هر کلاس باید فقط و فقط یک وظیفه داشته باشد و فقط به یک دلیل تغییر کند. اگر کلاس چند وظیفه داشته باشد، تغییر در یک بخش ممکن است بر بخش های دیگر هم تاثیر بگذارد. با رعایت SRP، به کلاس هایی خواهیم رسید که نگهداری اسان و درک کد بالاتری را خواهند داشت.

حالا به یک سوال چالشی می رسیم : نمی توانم وظایف را از همدیگه تشخیص و تفکیک کنم. چه جوری مشکلم را حل کنم؟

راه حل: یکسری معیارها هست که می تونی با به کارگیری ان ها وظایف را از هم تشخیص بدی.

۱: دامنه تغییرات را در نظر بگیری

- کلاسی داریم که چند تا کار متفاوت را داره انجام می ده. بنابراین هر بار که تغییری انجام میشه و دلیل تغییر متفاوت هست، داره به ما نشان میده که این کلاس مسئولیت‌های متعدد و مختلفی دارد.

۲: تست پذیری کلاس را بسنجی

- کلاس هایی که مسئولیت‌های متعددی را دارند، تست پذیری کمتری را دارند و نوشتن تست سناریوها برای انها پیچیده و هزینه بر هست زیرا در نوشتن تست همه ی ویژگی ها را باید در نظر گرفت.

۳: کاهش وابستگی های کلاس را لحاظ کنی

- اگر کلاس شما به چند کلاس یا ماژول های دیگر وابسته باشد که در زمینه های متفاوت (غیر از وظیفه ای که این کلاس دارد) کار می کنند، نشان می دهد این کلاس مسئولیت‌های متعددی را دارد.

حالا بیا یک سناریو را در سیستم بانکی بررسیش کنیم:

← کاربران می توانند کارهای مختلفی را انجام بدهند از جمله ی آنها

حساب های بانکی خود را مدیریت کنند – وام دریافت کنند – ثبت نام کنند و پیامک تایید دریافت کنند.

در این مثال، اصل SRP را رعایت نمی کنیم. ببینیم طراحی سیستم به چه شکلی خواهد بود:

```
public class BankService {
    public void registerUser(String email, String password) {
        System.out.println("Registering user: " + email);
        // Save new user to DB
        saveUserToDatabase(email, password);
        // Send a confirmation message to a user
        sendConfirmationEmail(email);
    }
public void createAccount(String email, double initialDeposit) {
        System.out.println("Creating account for: " + email);
        // Save Account to DB
        saveAccountToDatabase(email, initialDeposit);
    }
private void saveUserToDatabase(String email, String password) {
        System.out.println("Saving user to database: " + email);
    }
```

```
    private void sendConfirmationEmail(String email) {
        System.out.println("Sending confirmation email to: " + email);
    }

    private void saveAccountToDatabase(String email, double initialDeposit) {
        System.out.println("Saving account with initial deposit: " + initialDeposit);
    }
}
```

استفاده کلاینت از کد:

```
public class BankApplication {
    public static void main(String[] args) {
        BankService bankService = new BankService();
        bankService.registerUser("customer@example.com", "securePassword");
        bankService.createAccount("customer@example.com", 1000);
    }
}
```

در همین کدی که نوشتیم براحتی می بینیم که کلاس BankService چقدر مستعد تغییر است.

هم داره کار ثبت نام را انجام می دهد و هم حساب بانکی را ایجاد می کند و هم پیام تایید می فرستد و ... . بنابراین هر کاری نیاز به تغییری داشته باشد منجر به تغییر این کلاس خواهد شد و دلیل تغییرات به بیش از یک دلیل می رسد و دامنه تغییرات مان بیشتر می شود. و این طراحی بدی است.

تست این کلاس را ببینیم.

```
import static org.junit.Assert.*;
import org.junit.Test;

public class BankServiceTest {

    @Test
    public void testRegisterUser() {
        BankService bankService = new BankService();
        bankService.registerUser("test@example.com", "password");
        // تست واقعی ندارد زیرا منطق به هم پیوسته است
        assertTrue(true); // فقط برای نشان دادن ساختار تست
    }
}
```

قدم بعدی، اصلاح کد نوشته شده است و اصل SRP را رعایت می کنیم.

حالا بیایید طراحی قبلی را اصلاح کنیم و طراحی جدید کلاس ها را ببینیم

```
public class UserService {
    private final UserRepository userRepository;
    private final EmailService emailService;

    public UserService(UserRepository userRepository, EmailService emailService) {
        this.userRepository = userRepository;
        this.emailService = emailService;
    }
public void registerUser(String email, String password) {
        System.out.println("Registering user: " + email);
        userRepository.saveUserToDatabase(email, password);
        emailService.sendConfirmationEmail(email);
    }
}
public class AccountService {
    private final AccountRepository accountRepository;

    public AccountService(AccountRepository accountRepository) {
        this.accountRepository = accountRepository;
    }
```

```
public void createAccount(String email, double initialDeposit) {
        System.out.println("Creating account for: " + email);
        accountRepository.saveAccountToDatabase(email, initialDeposit);
    }
}
public class UserRepository {
    public void saveUserToDatabase(String email, String password) {
        System.out.println("Saving user to database: " + email);
    }
}
public class AccountRepository {
    public void saveAccountToDatabase(String email, double initialDepositAmount) {
        System.out.println("Saving account with initial deposit: " + initialDepositAmount);
    }
}
public class EmailService {
    public void sendConfirmationEmail(String email) {
        System.out.println("Sending confirmation email to: " + email);
    }
}
```

استفاده کلاینت از کد:

```
public class BankApplication {
    public static void main(String[] args) {
        UserRepository userRepository = new UserRepository();
        EmailService emailService = new EmailService();
        UserService userService = new UserService(userRepository, emailService);

        userService.registerUser("customer@example.com", "securePassword");

        AccountRepository accountRepository = new AccountRepository();
        AccountService accountService = new AccountService(accountRepository);
        accountService.createAccount("customer@example.com", 1000);
    }
}
```

چه تغییری ایجاد شد؟

کد تمیز و زیبا را می بینیم. کلاس UserService فقط داره ثبت نام کاربر را انجام میده و اگر شامل تغییری شود، دلیل تغییر فقط می تونه به ثبت داده های کاربر ربط داشته باشه و نه چیزی دیگر. و بقیه ی کلاس ها هم با این منطق و به صورت خیلی ظریف طراحی شده اند.

حالا کلاس تست ها را ببینیم که چه قدر ساده و با تست پذیری بالا نوشته می شوند.

```
import static org.junit.Assert.*;
import org.junit.Test;

public class UserServiceTest {

    @Test
    public void testRegisterUser() {
        UserRepository mockRepo = new UserRepository() {
            @Override
            public void saveUserToDatabase(String email, String password) {
                // منطق تست
                assertEquals("test@example.com", email);
            }
        };

        EmailService mockEmailService = new EmailService() {
            @Override
            public void sendConfirmationEmail(String email) {
                assertEquals("test@example.com", email);
            }
        };
```

```
        UserService userService = new UserService(mockRepo, mockEmailService);
        userService.registerUser("test@example.com", "password");
    }
}

public class AccountServiceTest {

    @Test
    public void testCreateAccount() {
        AccountRepository mockRepo = new AccountRepository() {
            @Override
            public void saveAccountToDatabase(String email, double initialDeposit) {
                // منطق تست
                assertEquals("test@example.com", email);
                assertEquals(1000, initialDeposit, 0);
            }
        };

        AccountService accountService = new AccountService(mockRepo);
        accountService.createAccount("test@example.com", 1000);
    }
```

}

در انتها چی نتیجه می گیریم؟

با پیاده‌سازی اصل SRP و تفکیک مسئولیت‌ها، می‌تونیم کدی قابل فهم‌تر، تست‌پذیرتر و قابل نگهداری‌تری بنویسیم و در اینده تغییرات آسان تر انجام می شود و خطا کاهش می یابد.

این اصل را نقض کنیم، به پیچیدگی در طراحی و نگهداری می رسیم، در نوشتن تست دچار مشکل می شویم و تغییرات هزینه بر در اینده را تضمین دادیم.