ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

EXERCISE 03

SINGLE RESPONSIBILTY PRINCIPLE

SOFTWARE ANALYSIS AND DESIGN

21120280 – Lý Minh Khuê

Thành phố Hồ Chí Minh – 2024

TABLE OF CONTENTS

TABLE OF CONTENTS **2**
CONCEPT OF SRP **3**
IMPORTANCE OF SRP **6**
1. Enhanced Readability and Maintainability: 6
2. Improved Scalability: 6
3. Facilitates Reusability: 6
4. Encourages Modular Design: 6
5. Better Collaboration: 6

CHALLENGES AND DISADVANTAGES **7**
1. Increased complexity: 7
2. Over-Abstraction: 7
3. Complexity of Interactions: 7
4. Trade-offs with Other Principles: 7
5. Project Size and Maintenance: 7

IMPLEMENTATION STRATEGIES **8**
1. Identify Responsibilities: 8
2. Use Design Patterns: 8
3. Separation of Concerns: 8
4. Encapsulation and Abstraction: 8
5. Code Reviews and Pair Programming: 8

REFERENCES **9**

CONCEPT OF SRP

Single Responsibility Principle(SRP) is one part of the **SOLID** – a set of 5 principles in design(**not exclusively** in software), each letter stands for a specific principles.

SRP is a fundamental principle that infers "every module should have one reason to change". Sounds concise as it is, do not misunderstand it with "a function/module should do one job". While this holds true, but slicing the functions down to smaller ones is not about **SRP** nor **SOLID.**

So what "one reason to change" is actually referring to? Now software system are always entitled to adapt through time to satisfy the requirements of users or stakeholders. Therefore, the phrase generally aims at those who affect the module/system, here are **users** and **stakeholders**. However, terms like "users" and "stakeholders" are not quite accurate. We use a familiar term in sofware analysis and design **actor** which encompass enough.

A module should be responsible to one, and only one, actor.

For example:

```java
// Class handling both user authentication and profile management
no usages
class UserHandler {
    no usages
    public boolean authenticateUser(String username, String password) {
        return true;
    }

    no usages
    public UserProfile getUserProfile(String username) {
        return new UserProfile(username, email: "abcd1234@email.com");
    }
}

// Class representing a user profile
2 usages
class UserProfile {
    1 usage
    private String username;
    1 usage
    private String email;

    1 usage
    public UserProfile(String username, String email) {
        this.username = username;
        this.email = email;
    }
}
```

Figure 1. User module without SRP

- **`UserHandler`** class is responsible for both authenticating users (authenticateUser method) and managing user profiles (getUserProfile method).
- Violation: The **`UserHandler`** class has multiple responsibilities: authentication and profile management. If there are changes needed in one aspect (authentication logic), it may affect the other aspect (profile management), leading to potential code complexity and maintenance issues.

This violates the **Single Responsibility Principle** because a single class should have only one reason to change.

```java
// Class handling user authentication
no usages
class UserAuthenticator {
    no usages
    public boolean authenticateUser(String username, String password) {
        return true;
    }
}

// Class handling user profile management
no usages
class UserProfileManager {
    no usages
    public UserProfile getUserProfile(String username) {
        return new UserProfile(username, email: "abcd1234@email.com");
    }
}

// Class representing a user profile
2 usages
class UserProfile {
    1 usage
    private String username;
    1 usage
    private String email;

    1 usage
    public UserProfile(String username, String email) {
        this.username = username;
        this.email = email;
    }
}
```

Figure 2. User module with SRP

- **`UserAuthenticator`**: Responsible for authenticating users based on provided credentials.
- **`UserProfileManager`**: Responsible for managing user profiles, such as fetching user profiles from a database.
- **`UserProfile`**: Represents a user profile, with fields for username and email. This class has a single responsibility of storing user profile information.

Each class encapsulates a single responsibility, adhering to the Single Responsibility Principle.

IMPORTANCE OF SRP

1. Enhanced Readability and Maintainability:

Classes and modules are focused on a single responsibility. This makes the codebase easier to read, understand, and maintain. When each class has a clear and distinct purpose, it becomes simpler to identify and fix bugs or make changes without impacting unrelated parts of the system.

2. Improved Scalability:

By breaking down complex functionalities into smaller, well-defined components, **SRP** enables better scalability. Each responsibility can be independently managed and extended, allowing for easier integration of new features or modifications without affecting the entire system.

3. Facilitates Reusability:

When a class has a single responsibility, it can be easily extracted and reused in other parts of the system or even in different projects. This promotes code reuse and reduces duplication.

4. Encourages Modular Design:

SRP encourages a modular design approach where each module or component encapsulates a specific set of related functionalities. This modularity promotes better organization and separation of concerns within the codebase, making it easier to manage and evolve over time.

5. Better Collaboration:

When multiple developers work on a project, adhering to **SRP** can foster collaboration. Each developer can focus on implementing or modifying components related to specific responsibilities without interfering with others' work. This reduces conflicts and promotes smoother teamwork.

CHALLENGES AND DISADVANTAGES

1. Increased complexity:

As the system requires more classes to implement the same functionality. This can potentially make the codebase more complex and harder to navigate.

2. Over-Abstraction:

Dividing responsibilities too finely can result in excessive abstraction, where the code becomes difficult to understand. Balancing the granularity of responsibilities and abstraction levels is crucial to maintaining code clarity and simplicity.

3. Complexity of Interactions:

When responsibilities are divided into many small components, managing interactions between these components can become more complex. Coordination between classes or modules may require additional design patterns or architectural considerations to ensure cohesion and maintainability.

4. Trade-offs with Other Principles:

SRP must be balanced with other design principles. Going too deep for **SRP** may sometimes conflict with these principles, requiring developers to make trade-offs based on the specific requirements and constraints of the project.

5. Project Size and Maintenance:

SRP may be more challenging to apply effectively in larger, more complex projects with numerous interrelated components. Maintaining a clear separation of responsibilities becomes increasingly difficult as the size and scope of the project grow.

IMPLEMENTATION STRATEGIES

1. Identify Responsibilities:

Analyze existing functions and classes to identify their core responsibilities. Aim for high cohesion within classes, where each class focuses on a single, well-defined responsibility.

2. Use Design Patterns:

Utilize design patterns to help enforce **SRP** and promote modular, maintainable code. Patterns such as the Factory Method, Strategy, and Observer can help separate concerns and encapsulate responsibilities in distinct classes.

3. Separation of Concerns:

Identify different aspects of functionality, such as presentation, business logic, and data access, and encapsulate them in separate components. This separation makes it easier to understand, modify, and maintain each part of the system.

4. Encapsulation and Abstraction:

Encapsulate implementation details within classes. Use abstraction to hide complex internal workings and provide a simplified, high-level interface for interacting with objects.

5. Code Reviews and Pair Programming:

Conduct regular code reviews and engage in pair programming to promote adherence to **SRP**. Review code with a focus on identifying violations of **SRP** and providing constructive feedback on how to refactor and improve design. Collaborative practices help spread knowledge and encourage adherence.

REFERENCES

https://www.freecodecamp.org/news/solid-principles-single-responsibility-principle-explained/
https://techmaster.vn/posts/33702/nguyen-tac-phat-trien-phan-mem-tot
https://www.ggorantala.dev/srp-advantages-and-disadvantages/