THE UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

SOFTWARE DESIGN - 22KTPM3

Single Responsibility (SRP) and Don't Repeat Yourself (DRY) Principle.

Members

ID	Full name
21126028	Vũ Huyền Thiên Lý
22127410	Lưu Thanh Thuý
22127424	Nguyễn Phước Minh Trí
22127435	Võ Lê Việt Tú
22127444	Trần Thị Cát Tường

Instructors

Trần Duy Thảo

Hồ Tuấn Thanh

Nguyễn Lê Hoàng Dũng

Contents

Chapter 1

Introduction

In software development, creating maintainable, scalable, and efficient systems is a fundamental goal. To achieve this, developers rely on well-established principles that guide the design and structure of code. Among these, the Single Responsibility Principle (SRP) and the Don't Repeat Yourself (DRY) Principle are two cornerstones of clean and effective software design.

The Single Responsibility Principle emphasizes that each module or class should have only one reason to change, ensuring focused functionality and reducing complexity. On the other hand, the DRY principle advocates for eliminating redundancy by centralizing reusable logic, leading to cleaner and more concise codebases.

By adhering to these principles, developers can produce systems that are easier to understand, test, and maintain, while minimizing bugs and duplication. This report explores the essence of SRP and DRY, their benefits, and practical ways to implement them in software projects.

Chapter 2

Single Responsibility Principle (SRP)

2.1 Principle content

SOLID is a set of principles applied in object-oriented programming (OOP) design. These principles help developers write code that is easy to read, understand, and maintain. SOLID includes: 1. Single Responsibility Principle 2. Open/Closed Principle 3. Liskov Substitution Principle 4. Interface Segregation Principle 5. Dependency Inversion Principle

Single Responsibility Principle, the first principle in SOLID, states that:

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

This principle can be understood as stating that each class should serve a single, specific purpose rather than multiple purposes.

2.2 Details

To visualize this principle, consider the image on the left part (in red) below.

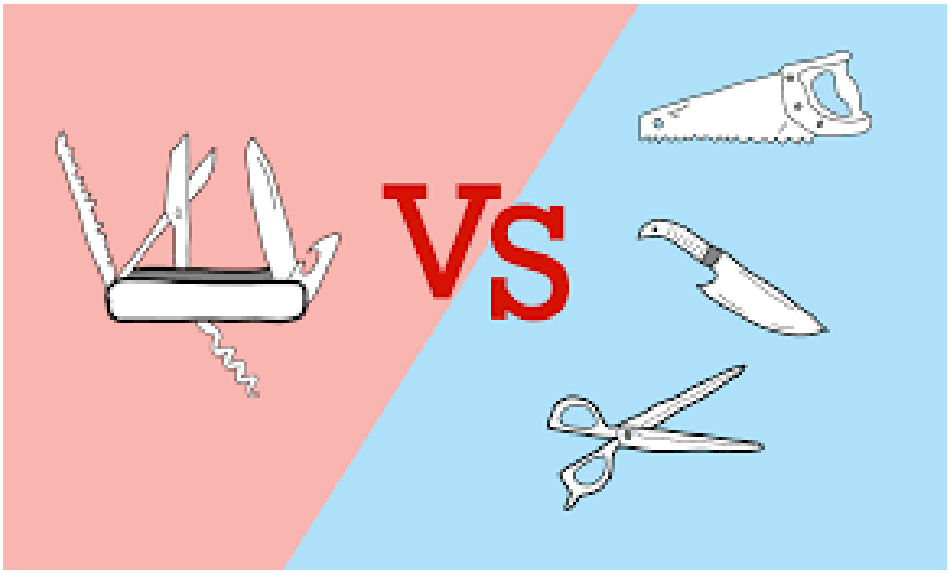

Figure 2.1: Simplifying the Single Responsibility Principle

It shows a multi-functional tool with numerous features: knife, scissors, corkscrew, etc. This tool might seem convenient at first glance, but it is bulky and heavy. More importantly, if one part of the tool breaks, you would need to disassemble the entire tool to repair it. The process of fixing or improving it becomes highly complex and could affect multiple other components.

A class with too many functions can also become bulky and overly complex. In the IT industry, requirements often change frequently, and adding or modifying features, as well as fixing bugs, tends to take more time than implementing the initial functionality. If a class has too many responsibilities and becomes too cumbersome, making changes to the code will be challenging, time-consuming, and may inadvertently impact other functioning modules. Moreover, over time, the project's codebase may balloon in size, making maintenance difficult and leaving it too late to refactor.

Applying the SRP to the multifunctional tool mentioned earlier, we could separate it into individual components like scissors, a knife, and a corkscrew (right part of image). This way, repairing or upgrading each part would be isolated and wouldn't affect other parts. The same principle applies to code: each module or class should be responsible for a single, specific function, making maintenance and modifications much simpler.

2.3 Code example

The code snippet below is an example of violating the SRP. The Student class has too many responsibilities: storing student information, formatting the display of the information, and saving the information.

```
public class Student {
  public string Name { get; set;}
  public int Age { get; set;}

  // Formatting
  public string GetStudentInfoText() {
    return "Name: " + Name + ". Age: " + Age;
  }
  public string GetStudentInfoHTML() {
    return "<span>" + Name + " " + Age + "</span>";
  }
  public string GetStudentInfoJson() {
    return Json.Serialize(this);
  }

  // Database-saving
  public void SaveToDatabase() {
    dbContext.Save(this);
  }
  public void SaveToFile() {
    Files.Save(this, "fileName.txt");
  }
}
```

Initially, this code may not seem problematic, but as more features are added later, the Student class will grow larger, making it harder to read and maintain. Additionally, if other classes like Teacher or Manager are introduced, and each has to implement formatting and database-saving functionalities, maintaining the code will become even more challenging.

The solution to this problem is to divide the responsibilities into smaller classes, with each class having a single responsibility. This way, modifications and changes can be confined to individual classes without affecting others. Moreover, functionalities like formatting or database-saving can be reused if new entities such as Teacher or Manager are introduced.

```
// Student
public class Student {
  public string Name { get; set;}
  public int Age { get; set;}
}
```

```
// Formatting
public class Formatter {
  public string FormatText(Student std) {
    return "Name: " + std.Name + ". Age: " + std.Age;
  }
  public string FormatHtml(Student std) {
    return "<span>" + std.Name + " " + std.Age + "</span>";
  }
  public string FormatJson(Student std) {
    return Json.Serialize(std);
  }
}

// Database-saving
public class Store {
  public void SaveToDatabase(Student std) {
    dbContext.Save(std);
  }
  public void SaveToFile(Student std) {
    Files.Save(std, "fileName.txt");
  }
}
```

Another typical example is the Utils class in many projects. This class often serves as a collection of functions used to handle small tasks. Initially, when the project is small, the Utils class might contain only a few functions.

```
// The Utils class violates the SRP.
// Since it is small, it can be overlooked.
public class Utils {
  public string GetUser();
  public DateTime GetTime();
  public string GetCurrentLocation();
  public DbConnection GetDatabaseConnection();
}
```

However, as the codebase grows, the number of functions in Utils can reach dozens or even hundreds. At that point, breaking it down into smaller, more focused classes becomes crucial.

```
// Utils is big, need to break down
public class Utils {
  public string GetUser();
```

```
    // ...
    public DateTime GetTime();
    // ...
    public string GetCurrentLocation();
    // ...
    public DbConnection GetDatabaseConnection();
}

// After breaking down into smaller
public class UserUtils {
    // ...
}
public class TimeLocationUtils {
    // ...
}
public class DatabaseUtils {
    // ...
}
```

Chapter 3

Don't Repeat Yourself Principle (DRY)

3.1 Principle content

"Don't Repeat Yourself" (DRY) is a software development principle that encourages developers to avoid duplicating code in a system.

This principle aims to reduce repetition of information that is likely to change, replacing it with abstractions that are less likely to change, or using data normalization to avoid redundancy in the first place.

3.2 Details

The main reason behind the emergence of this principle is **Duplicated Code**. Duplicated Code refers to code that is repeated in multiple places within a project. The critical issue here is that no one can guarantee that this duplicated code will never change. In the future, it might cause bugs or require feature updates. At that point, if the code has been duplicated 10 times in the source code, you will have to make changes in 10 different places, with the risk of missing some occurrences.

Writing DRY (Don't Repeat Yourself) code offers numerous benefits that enhance the quality and efficiency of software development. DRY code is easier to maintain because changes or updates need to be made in only one place, reducing the risk of inconsistencies. It also promotes cleaner and more readable code by eliminating unnecessary repetition, making it easier for developers to understand the logic. By reusing existing code instead of duplicating it, development time and effort are saved, resulting in more efficient and productive workflows. Additionally, DRY ensures consistent behavior across the codebase, as specific logic is centralized rather than scattered. This

consistency reduces the likelihood of introducing bugs or errors caused by inconsistent changes in duplicated code.

Applying the DRY mindset is essential. Developers or project managers are free to choose how to implement this principle. The implementations may vary depending on the specific use cases. Below are some ways to apply this principle:

1. Abstractions

   This is commonly the case for designing classes. When multiple classes share some common business logic, we simply abstract the logic into a superclass, which each class then inherits. Standard object-oriented programming (OOP) has many practices that encourage the DRY principle for writing readable, reusable, and maintainable code. Furthermore, there are alternatives like composition that solve the limitations of using inheritance and give programmers more flexibility.

2. Automation

   This is the case for project awareness of developers. Teams should be cross-functional to ensure active communication between developers. Managers should encourage such communication so developers never end up repeating completed work and can discuss their mutual problems.

   Removing duplication as it arises isn't enough. Instead, it must be an active process. Without proper systems and oversight, this problem will persist and continue to plague the organization. Some tools like **Sonarqube** can be applied to check project's quality.

3. Normalization

   This is the case for designing databases. Duplications are common in many data representations. This creates redundant data, which is harder to maintain.

   The goal of normalization is to ensure the data is consistent and properly distributed. It also helps maintain data integrity and creates a single source of truth. Furthermore, data normalization removes redundancies in data, which makes the database more flexible and scalable.

3.3 Code example

Below is an example of code that violates the DRY principle. In both the Student and Teacher classes, there is duplicate code for checking whether the registration email is valid. However, this duplication can cause difficulties when upgrading or modifying the email validation logic (e.g., requiring emails to end with .edu.vn). In such cases, there are two separate pieces of code that need to be updated.

```
public class Student {
    public boolean validateEmail(String email) {
        if (!email.contains("@") || !email.contains(".")) {
            System.out.println("Invalid email");
            return false;
        }
        return true;
    }
}

public class Teacher {
    public boolean validateEmail(String email) {
        if (!email.contains("@") || !email.contains(".")) {
            System.out.println("Invalid email");
            return false;
        }
        return true;
    }
}
```

To address this issue, we can create a dedicated class for email validation. With this approach, regardless of whether there are 2 classes or 100 classes that need email validation, any modifications or upgrades only need to be made in this single class.

```
public class EmailValidator {
    public static boolean validate(String email) {
        if (!email.contains("@") || !email.contains(".")) {
            System.out.println("Invalid email");
            return false;
        }
        return true;
    }
}

public class Student {
    public bool register(String email) {
        if (!EmailValidator.validate(email)) {
            return false;
        }
        // ...
    }
}
```

```
public class Teacher {
    public bool register(String email) {
        if (!EmailValidator.validate(email)) {
            return false;
        }
        // ...
    }
}
```

However, the code above still violates the DRY principle since both the Student and Teacher classes share the same register method. This can be resolved by creating an abstract class User, which the two classes will inherit from.

```
class EmailValidator {
    public static boolean validate(String email) {
        if (!email.contains("@") || !email.contains(".")) {
            System.out.println("Invalid email.");
            return false;
        }
        return true;
    }
}

abstract class User {
    public bool register(String email) {
        if (EmailValidator.validate(email)) {
            return false;
        }
        // ...
    }
}

class Student extends User {

}

class Teacher extends User {

}
```

Chapter 4

Conclusion

The **Single Responsibility Principle (SRP)** and **Don't Repeat Yourself Principle (DRY)** are two of the most important principles in software design. They provide a strong foundation for writing clean, maintainable, and scalable code. By adopting these principles, developers can create systems that are easier to understand, modify, and extend while minimizing redundancy and inconsistencies. Developers should embrace these mindsets to cultivate better coding practices, improve collaboration within teams, and deliver high-quality software that can adapt to changing requirements efficiently.