Design Principles and Patterns

Overview of Design Principles

The designing and development of the software are performed in the design phase of a software development project. When developing the software of the project, generally there are changes in the requirements document that may occur, such as the client adding additional requirements or changing a requirement. These changes could affect the development of the software where there is a need to apply changes on the source codes.

When the software architecture has a bad design, it will be hard to apply changes on the project. According to Robert Martin (as cited in oodesign.com), there are three (3) important characteristics of a bad design architecture that should be avoided:

- **Rigidity** – The software is hard to change because every applied change will affect many parts of the software.
- **Fragility** – When applying changes, unexpected parts of the software breaks.
- **Immobility** – The modules of the software are hard to reuse ins another software because these cannot be extracted from the current software.

To create a good design architecture, various software design principles can be followed. **Design principles** are a set of guidelines to be followed that helps developers arrange methods and data structures into classes, and how those classes should be interconnected, which can adapt to the changes in requirements without major code rewrites.

The SOLID Principles

Martin assembled a set of five (5) software design principles, and Michael Feathers arranged these principles into an acronym called SOLID. The **SOLID principles** of object-oriented design are applied to software development projects to make software easier to change when requirements changes. These are the design principles of the SOLID principles:

- **S: Single Responsibility Principle (SRP)** – This is one of the basic principles most developers apply to build robust and maintainable software. This suggests that each software module, class, or interface should have only one (1) reason to change.
- **O: Open-Closed Principle (OCP)** – This states that for a software to be easy to change, the software classes must be designed to allow the behavior of those classes to be changed by adding new code rather than changing existing code.
- **L: Liskov Substitution Principle (LSP)** – Barbara Liskov introduced this principle which states that the derived classes should be substitutable for their base classes to build a software from interchangeable modules or classes.
- **I: Interface Segregation Principle (ISP)** – This principle advises software designers to avoid depending on things that they don't use.
- **D: Dependency Inversion Principle (DIP)** – This principle suggests that flexible software are those with classes that depend on abstract classes or interfaces.

Each of these principles can be practiced by any software developer to help them in solving various software development problems. Following these principles help developers to achieve the following:

- Reduce the complexity of source codes
- Increase readability, extensibility, and maintenance
- Reduce accidental errors and implement reusability easily
- Achieve easier and better software testing.

Single Responsibility Principle

The **Single Responsibility Principle (SRP)** instructs developers to design each module, interface, or class of a software system to have only one (1) responsibility. The software systems are changed to satisfy the needs of the users or actors. Those users are the "reason to change" that the principle is referring to.

When requirements change during development, the responsibility of a class also changes. Classes with more than one (1) responsibility should be broken down into smaller classes, each of which should only have a single responsibility and reason to change. SRP does not state that a class should only have one (1) method; rather, a class can have any number of members such as methods and instance variables as long as its members are related to only single responsibility of the class.

Example class that violates SRP

Figure 1 shows an example of a UML class diagram of `Employee` class from a payroll application. This class violates the SRP because its methods consist of three (3) responsibilities to three (3) different actors:

- The accounting personnel is responsible for computing the salary of an employee. It should use the `calculatePay()` method.
- The human resource personnel is responsible for creating a report of hours of an employee. It should use the `reportHours()` method.
- The database administrator is responsible for saving an employee's details. It should use the `saveEmployee()` method.

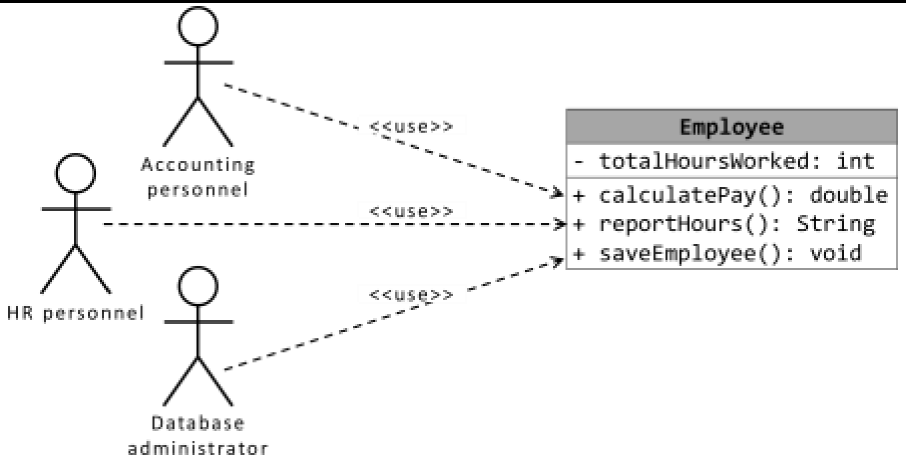

Figure 1. The `Employee` class
Source: Clean Architecture. A Craftsman's Guide to Software Structure and Design, 2017. p. 65

The figure above shows an example of bad design architecture. Putting the three (3) methods into a single `Employee` class coupled several actors that can affect each other's functions, while these actors do not use the other methods.

For example, the accounting team requests to change the way how the hours are calculated. The developer makes changes on the `totalHoursWorked` variable that the `calculatatePay()` method is calling. Unfortunately, the developer does not notice that the modified method is also called by the `reportHours()` method. The accounting team validates the modified method as they desire, but the HR team does not know about these changes and continue to use the `reportHours()` method. As a result, they are getting incorrect values.

This problem may occur when changes are applied to the software classes that has bad design architecture. This is because multiple methods or responsibilities are put into a single class in which different actors are depending on. SRP suggests separating the code that different actors depend on.

<u>Example solution using SRP</u>
To solve this problem, divide the responsibilities should and put these into different classes for different actors. *Figure 2* shows how to separate the responsibilities from the class `Employee` into different classes. Each class holds only the source code necessary for its particular method. These three (3) classes are not allowed to know each other, and the data is separated from their methods although they share access to the `EmployeeData` class. Thus, any accidental merge of data or methods is avoided.

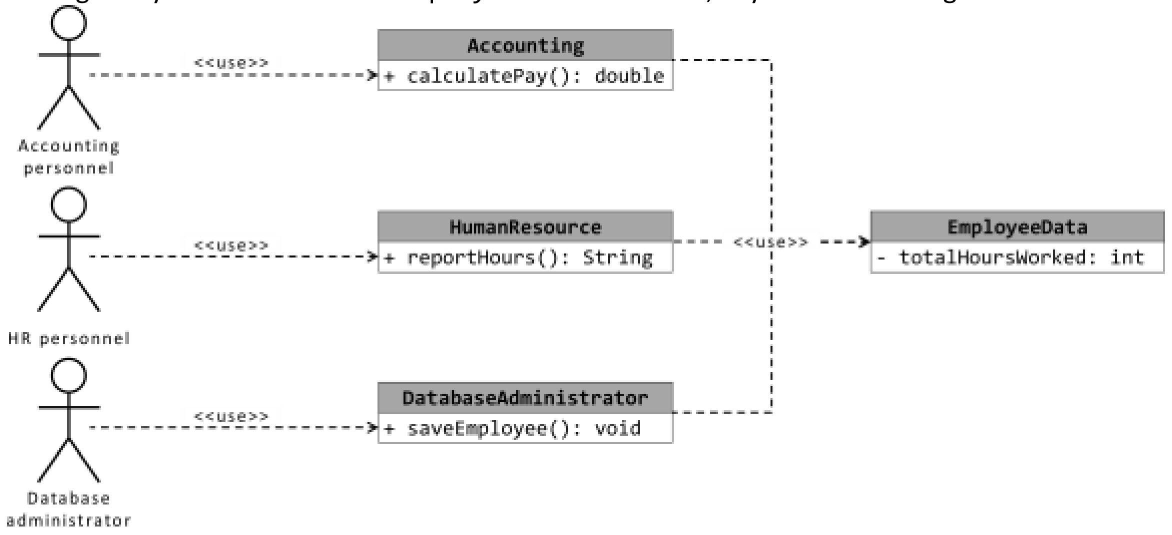

Figure 2. The three (3) classes do not know about each other
Source: Clean Architecture. A Craftsman's Guide to Software Structure and Design, 2017. p. 67

In this solution, each class suggests that developers make several classes that contain fewer methods and instance variables. But this solution using SRP will make the project easier to maintain where developers can easily add changes or new requirements without major code rewrites.

Open-Closed Principle

The **Open-Closed Principle (OCP)** states that software modules, interfaces, or classes should be open for extension but closed for modification. This means that when there are new requirements for the software, the new behavior must be added by deriving a new class without modifying the existing class. Because if developers change the behavior of the class, the changes may cause some parts of the software to break. OCP suggests not to add functionality on the class that has been already tested and is working properly, but rather to extend that class for adding new functionality.

To apply this principle, abstract the class using inheritance or interfaces. Then, whenever there is a new requirement, that functionality should be added to a new derived class without modifying the existing class.

Example class that violates OCP

Figure 3 shows an example class diagram which violates the OCP. The `BankAccount` class handles the methods for creating different types of employee. This class violates the OCP since the `BankAccount` class has to be modified every time a new type of bank account has to be added.

The disadvantages of this architecture are as follows:

- For every newly added method, the unit testing of the software should be done again.
- When a new requirement is added, the maintenance and adding function may take time.
- Adding a new requirement might affect the other functionality of software even if the new requirement works.

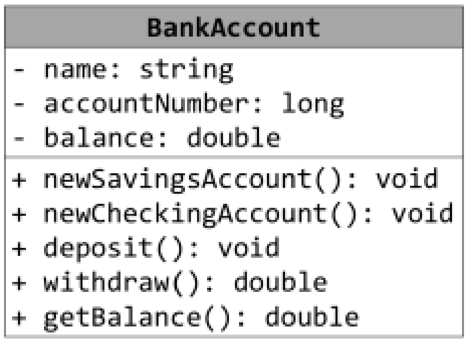

Figure 3. Example diagram that violates OCP
Source: Software Development, Design and Coding: With Patterns, Debugging, Unit Testing, and Refactoring (2nd ed.), 2017. p. 126

Implementing the OCP

To implement the OCP, use an interface, an abstract class, or abstract methods to extend the functionality of the class.

Figure 4 shows an abstraction of the `BankAccount` class. This abstract class is now closed for modification but opened for an extension since it can be extended to create new classes for a new type of bank accounts without modifying the `BankAccount` abstract class.

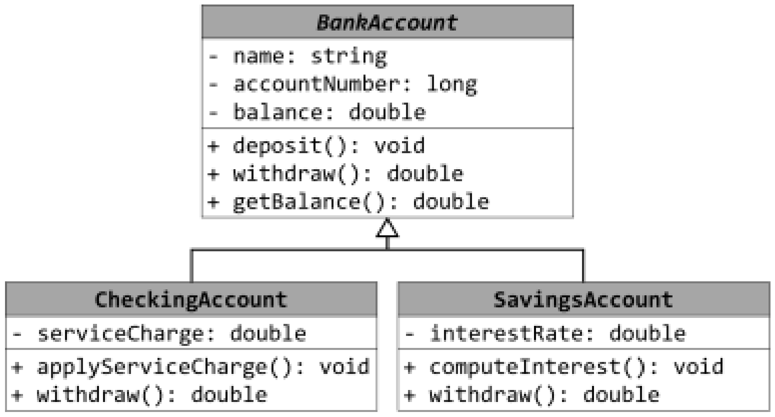

Figure 4. Example diagram that adheres to OCP
Source: Software Development, Design and Coding: With Patterns, Debugging, Unit Testing, and Refactoring (2nd ed.), 2017. p. 126

Liskov Substitution Principle

The **Liskov Substitution Principle (LSP)** suggests that when creating a new derived class of an existing class, make sure that the derived class can be a substitute for its base class. This means that all the derived classes of a base class should be able to perform the same behavior of their base class without negative side effects.

Example class that conforms to LSP

Figure 5 shows an example of class design architecture that conforms to the LSP. The class named `License` has a method named `calculateFee()`, which is called by the `Billing` class. There are also two (2) derived classes of the `License` class named `PersonalLicense` and `BusinessLicense`. These two (2) derived classes use different algorithms to calculate the license fee. This design conforms to the LSP because the behavior of the `Billing` class does not depend on any of the two (2) derived classes. But both of the derived classes can substitute the `License` class.

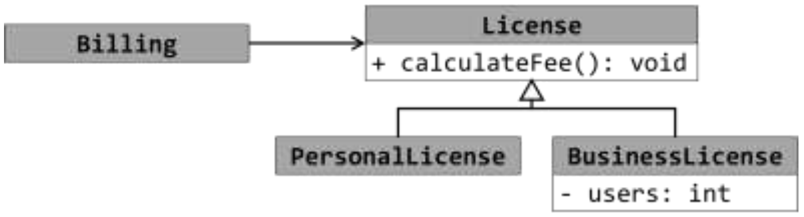

Figure 5. License class and its derived classes that conforms to LSP
Source: Clean Architecture. A Craftsman's Guide to Software Structure and Design, 2017. p. 76

The LSP should be extended to the level of architecture. A simple violation of substitution can cause a system's architecture to break.

Interface Segregation Principle

The **Interface Segregation Principle (ISP)** suggests avoiding depending on things that are not used. This means that clients should not be forced to implement interfaces they don't use. Instead, on one (1) big interface, several small interfaces are preferred based on groups of methods where each interface serves one (1) submodule.

Example class that violates ISP

Figure 6 shows an example class diagram for an e-commerce website that needs to have a shopping cart and associated order-processing mechanism. The developer designed the interface `IOrderProcessor`, assuming that it will accept only online payments through credit cards. The design of the interface `IOrderProcessor`, therefore, has three (3) methods: `validateCardInfo()` that validates the credit card information, `validateShippingAddress()` that validates the shipping destination, and `processOrder()` that initiates the order processing by placing the order in the system.

The `OnlineOrderProcessor` class implements `IOrderProcessor` and implements all of its functionalities. When the requirements changed, where the e-commerce website considered to accept cash-on-delivery payments, the developer needs to create a new class named `CashOnDeliveryOrderProcessor` that also implements `IOrderProcessor` and all of its functionalities. But the `CashOnDeliveryOrderProcessor` class will not involve credit cards, so the developer designed the `validateCardInfo()` method that implemented inside the `CashOnDeliveryOrderProcessor` class to throw an exception to avoid any error.

With this design, the e-commerce website may work as expected, but a potential problem may occur in the future. For example, the process of online credit card-based payment has changed. The developer then will have to modify the `IOrderPrrocessor` interface to have additional methods to process credit card payments. However, even the `CashOnDeliveryOrderProcessor` class doesn't need any additional functionality. The `CashOnDeliveryOrderProcessor` will be forced to implement these additional methods and will have codes that throw an exception. This will take several changes on the program even if the `CashOnDeliveryOrderProcessor` class does not use those changes. This design architecture, therefore, violates the ISP.

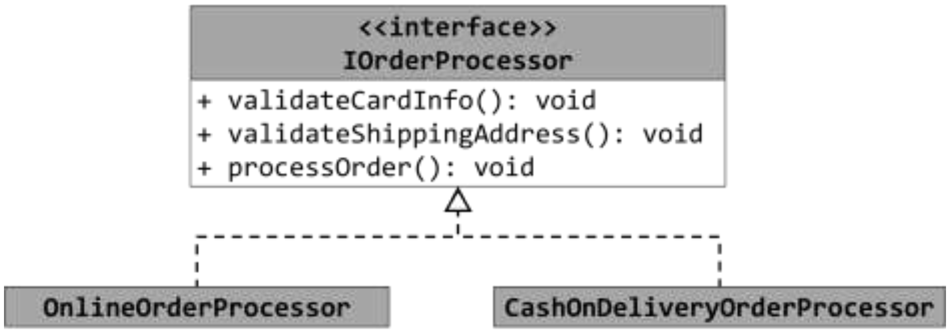
<<interface>>
IOrderProcessor
+ validateCardInfo(): void
+ validateShippingAddress(): void
+ processOrder(): void
OnlineOrderProcessor
CashOnDeliveryOrderProcessor

Figure 6. IOrderprocessor violates ISP
Source: Beginning SOLID principles and design patterns for asp.net developers, 2016. p. 74

Example solution using ISP

To solve this problem, ISP suggests splitting the required methods and instance variables into several interfaces. In the example class diagram in *Figure 6*, the methods on the interface `IOrderProcessor` are split into two (2) interfaces: `IOrderProcessor` and `IOnlineOrderProcessor` (see *Figure 7*). The `IOrderProcessor` interface includes two (2) methods: `validateShippingAddress()` and `processOrder()`. These two (2) methods are implemented by `OnlineOrderProcessor` and

CashOnDeliveryOrderProcessor classes. The other interface named IOnlineOrderProcessor consists of the validateCardInfo() method, which is implemented only by the OnlineOrderProcessor class. With this software design architecture, any new requirements on the online credit card-based payments are confined to IOnlineOrderProcessor and are implemented only in the OnlineOrderProcessor class. The CashOnDeliveryOrderProcessor class is not forced to implement these changes.

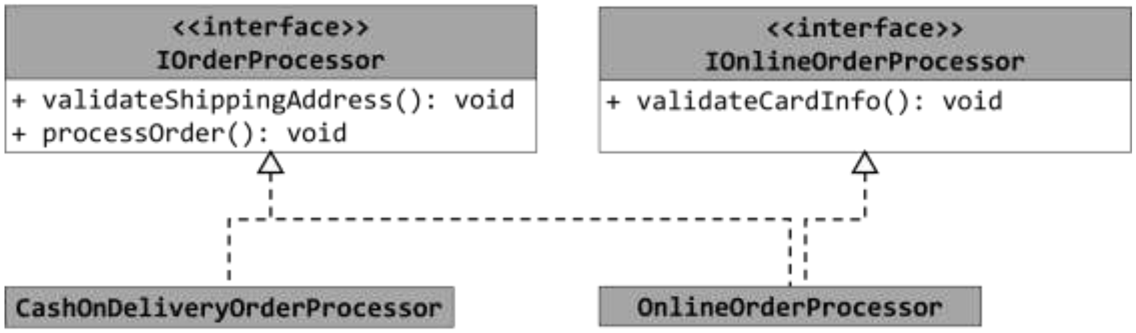

Figure 7. IOrderprocessor and IOnlineOrderProcessor conforms to ISP
Source: Beginning SOLID Principles and Design Patterns for asp.net Developers, 2016. p. 75

Dependency Inversion Principle

The **Dependency Inversion Principle (DIP)** suggests that the most flexible software systems are those in which source code dependencies refer only to abstractions, not to concretions. The DIP states the following parts:

A. **High-level classes should not be dependent on low-level classes. Both of them should depend on abstractions.** This means a high-level class is a class that does something significant in the application, while a low-level class is a class that does secondary work.
B. **Abstractions should not depend upon details. Details should depend upon abstractions.** This means that developers must design the abstract class by looking at the needed details of the class.

Example class that violates DIP

Figure 8 shows two (2) classes violating the DIP. The UserManager class depends on the EmailNotifier class. The UserManager class contains a method that changes a password, while the EmailNotifier class contains the method that notifies the user through e-mail. In this example, the UserManager class does the main task, while the EmailNotifier class does the secondary task. Therefore, the UserManager class is the high-level class, while the EmailNotifier is the low-level class. This design violates DIP because the high-level class depends on the low-level class.

The problem in this design is that the high-level class depends too much on the low-level class. When the requirements for the EmailNotifier class changes, the UserManager class might need some modifications. For example, when there are changes in the requirements of the software system that the notification must be SMS or pop-up, the UserManager class needs to be modified to change the new notification class. Another problem is that the low-level class must be available at the time of writing and testing the high-level class. The developer is forced to finish low-level classes before writing the high-level classes.

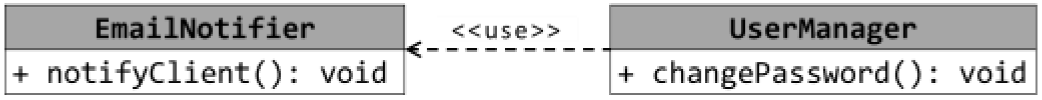

Figure 8. High-level class depends on a low-level class
Source: Beginning SOLID Principles and Design Patterns for asp.net Developers, 2016. p. 80

Example solution using DIP

To solve this problem, DIP suggests that both high-level and low-level classes should depend on abstraction such as abstract class or interface. In *Figure 9*, the UserManager class no longer uses the EmailNotifier class directly. Instead, an interface INotifier has been added.

The EmailNotifier class implements the INotifier interface. The constructor of UserManager class receives an instance of a class that implements INotifier from the external world. The changePassword() method now uses this instance to call the notifyClient() method. If the requirement for the notification feature changes, the changes will not affect the codes in the UserManager class. This design makes the high-level class UserManager and the low-level classes, such as the EmailNotifier, to depend on the same abstraction, which the INotifier interface. This design, therefore, conforms to the DIP.

The second part of DIP tells that details should depend upon abstractions. In the figure below, the *INotifier* interface is an abstraction that looks at the needs of the `UserManager` class. The `EmailNotifier`, `SMSNotifier`, and `PopupNotifier` classes contain the details that depend on the `INotifier` interface.

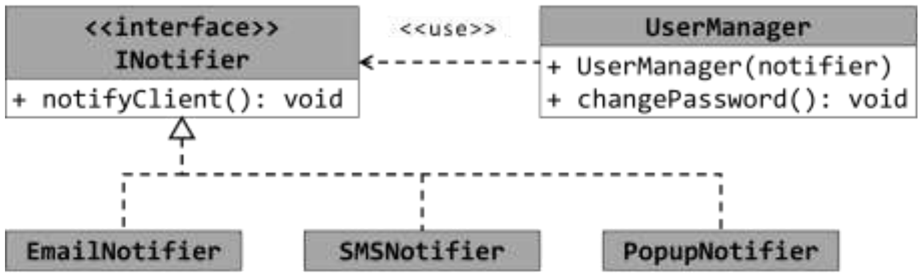

Figure 9. Design conforming to DIP
Source: Beginning SOLID Principles and Design Patterns for asp.net Developers, 2016. p. 81

REFERENCES:

Design Principles (n.d.). In *OODesign.com Object Oriented Design*. Retrieved from https://www.oodesign.com/design-principles.html

Dooley J. (2017). *Software development, design and coding: With patterns, debugging, unit testing, and refactoring* (2nd ed.). Retrieved from https://books.google.com.ph/books?id=LGRADwAAQBAJ&dq=Software+Development,+Design+and+Coding:+With+Patterns,+Debugging,+Unit+Testing,+and+Refactoring

Joshi, B. (2016). *Beginning SOLID principles and design patterns for asp.net developers.* California: Apress Media, LLC.

Martin, R. (2017). *Clean architecture. A craftsman's guide to software structure and design.* Retrieved from https://archive.org/details/CleanArchitecture