SOLID principles

## What are SOLID principles

* They are a set of rules and best practices to follow while designing classes structure
* Help developers write readable, reusable, maintainable and scalable code
* They are:
  + **S**ingle Responsibility Principle
  + **O**pen-Closed Principle
  + **L**iskov Substitution Principle
  + **I**nterface Segregation Principle
  + **D**ependency Inversion Principle

## Single Responsibility Principle

**"a class should do one thing and therefore it should have only a single reason to change"**

* Only one potential change (database logic, logging logic...) in the software’s specification should be able to affect the specification of the class
* If software's specifications change, we only need to modify a specific class, and less likely to break the whole application
* *Why it important*:
  + If we edit the same class for different reason -> incompatible modules
  + Makes version control easier: fewer conflicts will appear and easier to resolve
* Examples: this class need refactor because the SendEmail and ValidateEmail methods have nothing to do with the UserService class

public class UserService  
{  
 public void Register(string email, string password)  
 {  
 if (!ValidateEmail(email))  
 throw new ValidationException("Email is not an email");  
 var user = new User(email, password);  
  
 SendEmail(new MailMessage("mysite@nowhere.com", email) { Subject="HEllo foo" });  
 }  
  
 public virtual bool ValidateEmail(string email)  
 {  
 return email.Contains("@");  
 }  
  
 public bool SendEmail(MailMessage message)  
 {  
 \_smtpClient.Send(message);  
 }  
}  
Copy

## Open-Closed Principle

**"software entities (classes, modules, functions...) should be open for extension and closed to modification"**

* Modification means changing the code of an existing class, and extension means adding new functionality
* We should be able to add new functionality without touching the existing code for the class
* It is usually done with the help of interfaces and abstract classes
* *Why it important*: avoid touching the tested and reliable code if possible
* *Note*: It's not possible to write code that suit for all future changes or perfectly closed -> do not overly complex the code
* Example:

public class InvoicePersistence {  
 Invoice \_invoice;  
  
 public InvoicePersistence(Invoice invoice) {  
 \_invoice = invoice;  
 }  
  
 public void SaveToFile(string fileName, string fileType) {  
 if (fileType == "json")   
 {  
 Console.WriteLine("Saving invoice to json");  
 }  
 else if (fileType == "txt")  
 {  
 Console.WriteLine("Saving invoice to txt");  
 }  
 }  
Copy

* Follow OCP:

public interface IInvoicePersistence   
{  
 public void Save(Invoice invoice);  
}  
  
public class JsonPersistence : IInvoicePersistence   
{  
 public override void Save(Invoice invoice) {  
 // Save to json  
 }  
}  
  
public class TxtPersistence : IInvoicePersistence   
{  
 public override void Save(Invoice invoice) {  
 // Save to txt  
 }  
}  
  
public class DatabasePersistence : IInvoicePersistence   
{  
 public override void Save(Invoice invoice) {  
 // Save to DB  
 }  
}  
Copy

## Liskov Substitution Principle

**"subclasses should be substitutable for their base classes and have it behave in the same manner without modification"**

* This principle can be consider as an extension of the OCP. We must ensure that newly derived classes extend the base classes without changing their behavior
* Some checklist items to determine anti-Liskov:
  + **No new exceptions should be thrown in derived class**: on reverse, if base class throw NullException but subclass allow null, it also violates
  + **Pre-conditions cannot be strengthened**: ex. subclass requires a int params to be positive but the base class doesn't
  + **Post-conditions cannot be weakened**: ex. base class requires db connection should be closed before return but the subclass doesn't
  + **Invariants must be preserved**: refer to the conditions that must hold true for the base class and its subclasses. Subclass must ensure that the invariants of the base class are preserved.
    - For example, a List<T> class has a Length property and an array of type T[]. Invariant of this class is that Length is always less than or equal to the length of the internal array. If a subclass of List<T> makes it so that Length can be greater than the length of the array -> violation of that invariant
* *Why it important*: when a class does not obey this principle, it leads to some bugs that are hard to detect

## Interface Segregation Principle

**"clients should not be forced to implement interfaces they don't use. Instead of one fat interface, many small interfaces are preferred based on groups of methods, each serving one submodule"**

* The larger the interface, the more likely it includes methods not all implementers can use
* Many client-specific interfaces are better than one general-purpose interface
* *Why it important*: easier to implement interfaces, increases the readability and maintainability
* example:

public interface ILead  
{  
 void CreateSubTask();  
 void AssginTask();  
 void WorkOnTask();  
}  
  
public class TeamLead : ILead  
{  
 public void AssignTask() {}  
 public void CreateSubTask() {}  
 public void WorkOnTask() {}  
}  
  
public class Manager : ILead  
{  
 public void AssignTask() {}  
 public void CreateSubTask() {}  
 public void WorkOnTask()  
 {  
 throw new Exception("Manager can't work on Task");  
 }  
}  
Copy

* Manager can't work on a task, so we are forcing the Manager class to implement a WorkOnTask() method without a purpose

public interface IProgrammer  
{  
 void WorkOnTask();  
}  
  
public class Programmer : IProgrammer  
{  
 public void WorkOnTask() {}  
}  
  
public class Manager : ILead  
{  
 public void AssignTask() {}  
 public void CreateSubTask() {}  
}  
  
public class TeamLead: IProgrammer, ILead  
{  
 public void AssignTask() {}  
 public void CreateSubTask() {}  
 public void WorkOnTask() {}  
}  
Copy

## Dependency Inversion Principle

**"high-level modules/classes should not depend on low-level modules/classes, both should depend upon abstractions"**

**"abstractions should not rely upon details. Details should depend upon abstractions"**

* High-level modules/classes implement business rules or logic in a system. Low-level modules/classes deal with more detailed operations
* If high-level modules depend on low-level modules, it raises the risk that changes to one class will break the other -> need to make both of them dependent on abstractions instead of knowing each other
* *Why it important*:
  + **Decoupling**: the code becomes more modular, making it easier to maintain, extend or replace completely
  + **Testability**: each module can be tested independently
  + **Reusability**: Modules become more reusable since they are less dependent on other modules
* example: the EmailNotification class is tightly coupled to the SmtpClient class

public class EmailNotification  
{  
 private SmtpClient \_smtpClient;  
  
 public EmailNotification()  
 {  
 \_smtpClient = new SmtpClient();  
 }  
  
 public void Send(string emailAddress, string message)  
 {  
 \_smtpClient.Send(emailAddress, message);  
 }  
}  
Copy

* Using DIP:

public interface IEmailSender  
{  
 void Send(string emailAddress, string message);  
}  
  
public class SmtpClient : IEmailSender  
{  
 public void Send(string emailAddress, string message)  
 {  
 // Implementation of email sending using SMTP  
 }  
}  
  
public class EmailNotification  
{  
 private IEmailSender \_emailSender;  
  
 public EmailNotification(IEmailSender emailSender)  
 {  
 \_emailSender = emailSender;  
 }  
  
 public void Send(string emailAddress, string message)  
 {  
 \_emailSender.Send(emailAddress, message);  
 }  
}  
Copy