Object Oriented Programming with Applications
Lecture 11

Witold Gawlikowicz

School of Mathematics, University of Edinburgh

2018-19[1]

Lecture 9

- S.O.L.I.D design principles
- Problem Sheet 7 overview

S.O.L.I.D design principles

A set of principles aimed at designing robust, readable and easily extendible code. These are especially useful as the codebase grows and matures. The five priciples are:

- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Single Responsibility Principle

- Every class should have only one responsibility.
- Whenever a class changes all the other classes depending on it might be affect and will need to be refactored and tested accordingly. Keeping classes small and resposible for one thing only will reduce that burden, making the code easier to modify and more robust.
- It is sometimes useful to think of a class as having only one reason to change - if one can think of more than one reason why a class may need to be changed it's often an indication that it has more than one responsibility and should therefore be broken down.

Single Responsibility Principle

Examples of resposibilities of a class include:

- Persistence
- Validation
- Notification
- Error Handling
- Logging
- Class Selection / Instantiation
- Formatting
- Parsing
- Mapping

Open/Closed Principle

"Software entities, i.e. classes, modules, functions and so forth should be open for extension, but closed for modification."

Liskov Substitution Principle

Subtypes must be substitutable for their base types. s
A very common violation of this principle is the partial
implementation of interfaces or base class functionality, leaving
unimplemented methods or properties to throw an exception (e.g.
NotImplementedException)

Interface Segregation Principle

- Clients (e.g. class or method depending on interface) shouldn't be forced to depend on methods they don't use.
- In practice this means using thin and focused interfaces, rather than broad, all-encompassing ones.
- Interfaces should only group elements that logically belong together.
- In C# interface inheritance can always be used to segragate interfaces.

Interface Segregation Principle

Consider:

```
public interface IMembership
{
  bool Login(string username, string password);
  void Logout(string username);
  string Register(string username, string password, string email);
  void ForgotPassword(string username);
}
```

It's easy to imagine such an interface growing completely out of control and having more functionality than any one class would ever require.

Interface Segregation Principle

It might make more sense to modify it as:

```
public interface ILogin
{
  bool Login(string username, string password);
  void Logout(string username);
}
public interface IRegister
{
  Guid Register(string username, string password, string email);
}
public interface IForgotPassword
{
  void ForgotPassword(string username);
}
```

Smaller interfaces are also easier to fully implement, thus reducing a chance of violating Liskov Substitution Principle.

Interface Segregation Principle

Of course there might still be scenarios where it makes sense to
define the original IMembership interface to bundle all that
functionality together:

```
public interface IMembership : ILogin, IRegister, IForgotPassword
{
}
```

IMembership ends up the same, but the other interfaces can be
used independently in other contexts.

Dependency Inversion Principle

- High level modules should not depend on low level modules - both should depend on abstractions.
- Abstractions should not depend on details - details should depend upon abstractions.

Problem Sheet 7

Available at:
https://github.com/OOPA2018/Problem-sheets/blob/
master/ProblemSheet7/ProblemSheet7.pdf

References

- "Agile Principles, Patterns, and Practices in C#" by Robert C. Martin and Micah Martin
- "The Elements of C# Style" by Kenneth Baldwin, Andrewy Grey and Trevor Misfeldt.
- https://deviq.com/solid/