UECS2344 Software Design: Lecture 6

Design Principles

The importance of software design can be stated in a single word – quality.

Software Quality Attributes

Functionality, Usability, Maintainability, Reliability, Performance, Testability, Extensibility, Adaptability, Configurability, Compatibility, etc.

Maintainability

Maintainability is the quality attribute you should give the highest priority when you design a system.

> *Maintainability refers to the degree to which code can handle changes without generating new issues or problems.*

Clean Code

- concept of Clean Code - developed by Robert C. Martin

Clean Code Principles

Loose Coupling

Two classes, components or modules are coupled when at least one of them uses the other. The less these items know about each other, the looser they are coupled.

A component that is only loosely coupled to its environment can be more easily changed or replaced than a strongly coupled component.

High Cohesion

Cohesion is the degree to which elements of a whole belong together.

Methods and fields in a single class and classes of a component should have high cohesion. High cohesion in classes and components results in simpler, more easily understandable code structure and design.

Code Smells

Needless Repetition

Remove code that contains exact code duplications or design duplicates (doing the same thing in a different way). Making a change to a duplicated piece of code is more expensive and more error-prone because the change has to be made in several places with the risk that one place is not changed accordingly.

Naming

Choose Descriptive / Unambiguous Names: Names have to reflect what a variable, field, property stands for. Names have to be precise.

Name Methods After What They Do: The name of a method should describe what is done, not how it is done.

Reduce Dependencies

Whenever a class, say A, uses another class, say B, (e.g. to call its methods), there is a coupling or dependency between the classes. When you copy the class A code to use in another application, you must also copy the code for class B.

This form of dependency might or might not be bad. It depends on the nature of the dependency.

- If the classes are logically correlated, it wouldn't be that bad to have coupling. The two classes form a logical unit.
- If they are not logically related, it is better to redesign the classes.

Look, for example, at the following code:

```
public class SomeComponent {

   public void doWork() {

      AuditLog auditLog = new AuditLog();

      String data = GetData();

      auditLog.record(data, java.util.Calendar.getInstance().getTime());
   }
   . . .
}
```

Is class `SomeComponent` dependent on class `AuditLog` or the other way around?

Is this dependency necessary? Yes, but it should be reduced. Next section describes how.

Separate Interface from Implementation Principle

The redesign needed for the code example above is to *separate the interface from the implementation*. This requires a refactoring.

> ***Refactoring*** *is the discipline of restructuring existing code in such a way that the external behaviour remains unaltered.*

You refactor your code not because it doesn't work but to improve some non-functional attribute such as readability, maintainability, or performance.

The `AuditLog` class needs to undergo a refactoring to extract an interface. Here's a possible definition for the `IAuditLog` interface:

```
public interface IAuditLog {

   public void record(String data, Date today);

}
```

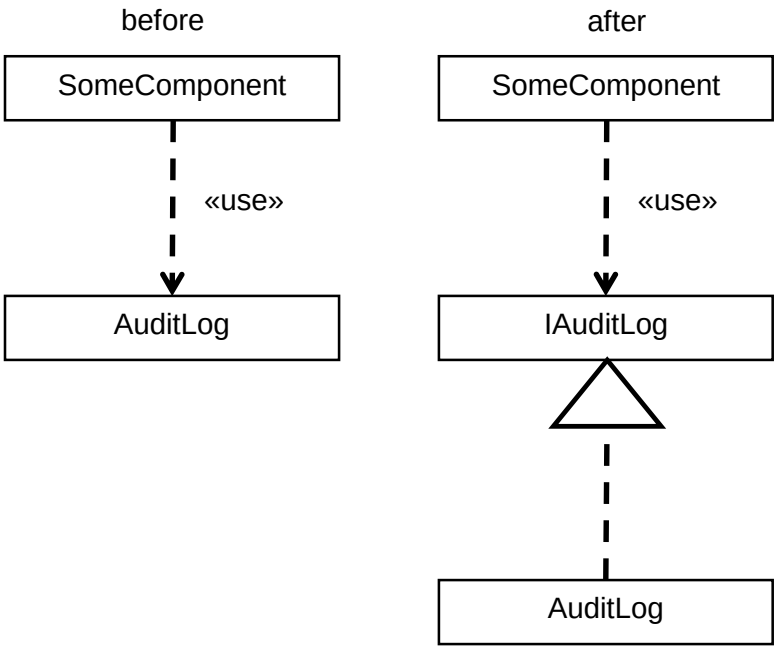
The refactoring step also produces the following AuditLog class:

```
public class AuditLog implements IAuditLog {

   public void record(String data, Date today) {
      . . .
   }
   . . .
}
```

SomeComponent class now makes use of the interface:

```
public class SomeComponent {

   public void doWork() {

      IAuditLog auditLog = new AuditLog();

      String data = GetData();

      auditLog.record(data, java.util.Calendar.getInstance().getTime());
   }
   . . .
}
```

This now changes the dependency of SomeComponent on AuditLog.



Note: Decoupling can be further improved by applying **Dependency Inversion Principle** (see below).

SOLID Principles - developed by Robert C. Martin

1) Single Responsibility Principle (S)

Strongly related to cohesion is the Single Responsibility principle. This principle says that each class should be given a single responsibility. A class with many responsibilities is less cohesive.

2) Open/Closed Principle (O)

A class should be open for extension but closed for modification. Open for extension means that an existing class should be extensible (for example, through subclasses) for building other related functionality. But to implement other related functionality, you should not change the existing code, so it remains closed for modification.

3) Liskov's Substitution Principle (L)

This principle (introduced by Barbara Liskov) says that subclasses should be substitutable for their base classes, that is, wherever a base class object is required, we must be able to substitute it with any subclass object.

4) Interface Segregation Principle (I)

This principle recommends that interfaces should have a minimum number of methods. If an interface contains unrelated methods, we should separate/segregate them into multiple interfaces.

For example, suppose we have the following interface:

```
public interface IDoor {
      public void lock()
      public void unlock();
      public boolean isDoorOpen();
      public void setTimeOut(int time);
      public int getTimeOut();
}
```

Suppose a class that implements the interface only needs the first 3 methods of `IDoor` interface. It is should be unnecessarily forced to implement all the methods.

The `IDoor` interface can be simplified and split into two interfaces to reflect two types of door:

```
public interface IDoor {
      public void lock()
      public void unlock();
      public boolean isDoorOpen();
}
public interface ITimedDoor extends IDoor{
      public void setTimeout(int time);
      public int getTimeout();
}
```

Now the class that implements the IDoor interfaces only needs to implement the 3 methods. A class that implements ITimedDoor interface needs to implement all 5 methods.

5) Dependency Inversion Principle (or Inversion of Control Principle) (D)

This principle says that the most flexible/maintainable systems are those in which code is dependent only on abstractions (e.g. interfaces) and not on concrete classes. This principle is related to the 'Separate Interface from Implementation' principle (refer above).

Example (The Problem)

```
public class CopyService {

	private Keyboard keyboard; // dependency on Keyboard

	private Printer printer;   // dependency on Printer

	// constructor creates the dependencies
	public CopyService() {
		keyboard = new Keyboard();
		printer = new Printer();
	}

	// CopyService does something with the dependencies
	public void copy() {
		String data = keyboard.read();
		printer.write(data);
	}
}
```

The CopyService class is not flexible or reusable.

Suppose we want to write documents to a PDF file instead of printing them. So, we create a new class PdfWriter. Now we have to stop the application, replace references to Printer with PdfWriter. Then we redeploy and restart the application.

A similar problem occurs if we want to use a VoiceReader class instead of the Keyboard class.

The Solution – Step 1: Separate Interface from Implementation

To make the application more flexible, we create interfaces IReader and IWriter:

```
public interface IReader {
	public String read();
}

public interface IWriter {
	public void write(String data);
}
```

The classes must then implement those interfaces accordingly:

```
public class Keyboard implements IReader {
	. . .
	public String read() {
		. . .
	}
}

public class VoiceReader implements IReader {
	. . .
	public String read() {
		. . .
	}
}

public class Printer implements IWriter {
	. . .
	public void write(String data) {
		. . .
	}
}

public class PdfWriter implements IWriter {
	. . .
	public void write(String data) {
		. . .
	}
}
```

Now we rewrite the `CopyService` class to use those interfaces.

```
public class CopyService {

	private IReader reader;

	private IWriter writer;

	. . .

	public void copy() {
		String data = reader.read();
		writer.write(data);
	}
}
```

But how do we specify which `IReader` object and `IWriter` object to use – `Keyboard` or `VoiceReader`, `Printer` or `PdfWriter`?

The Solution – Step 2: Dependency Injection

We apply dependency injection. There are two ways to inject the dependencies:

1. using the constructor or
2. using setter methods:

```
public class CopyService {

	private IReader reader;

	private IWriter writer;

	// using the constructor
	public CopyService(IReader reader, IWriter writer) {
		this.reader = reader;
		this.writer = writer;
	}

	// using setter methods
	public void setReader(IReader reader) {
		this.reader = reader;
	}

	public void setWriter(IWriter writer) {
		this.writer = writer;
	}

	public void copy() {
		String data = reader.read();
		writer.write(data);
	}
}

public class Application {

	public static void main(String args[]) {
		. . .
		// using Keybaord and Printer classes
		IReader reader = new Keyboard();
		IWriter writer = new Printer();

		CopyService service = new CopyService(reader, writer);
		service.copy();

		// using Keyboard and PdfWriter classes
		writer = new PdfWriter();
		service.setWriter(writer);
		service.copy();
	}
}
```

How can Dependency Injection principle be applied for the `SomeComponent-AuditLog` dependency problem?

```
public class SomeComponent {

   // pass the AuditLog object as a parameter (of type IAuditLog) to the method
   public void doWork(IAuditLog auditLog) {

      String data = GetData();

      auditLog.record(data, java.util.Calendar.getInstance().getTime());
   }
   . . .
}
```

Separation of Concerns Principle

Remember, we always aim for **high cohesion** and **low coupling**. A principle that is helpful to achieve this is Separation of Concerns, introduced in 1974, by Edsger W. Djikstra.

Concerns are different pieces of software functionality, like business logic or presentation.

Separation of Concerns is all about breaking the system into distinct and preferably non-overlapping parts. It is achieved through modular code. **Modular programming** encourages the use of separate modules for different functionalities. In object-oriented programming, modules may be classes or methods. So, with separation of concerns, different classes and their methods are defined to handle different aspects or parts of the functionality of the overall system.

How to Modularise Software?

- structured approach – hierarchical decomposition into functions
- object-oriented approach – identify classes and their relationships

Why Modularise Software?

So that:

- software can be developed iteratively and incrementally
- changes can be more easily accommodated and hence maintainability of the system is improved
- testing and debugging can be conducted more efficiently