Design Defects and Restructuring

LECTURE 05

SAT, SEP 21, 2019

Symptoms of Poor Design

Rigidity	• The design is hard to change
Fragility	• The design is easy to break
Immobility	• The design is hard to reuse
Viscosity	• It is hard to do right thing
Needless Complexity	• Overdesign
Needless Repetition	• Mouse abuse
Opacity	• Disorganized expressions

Rigidity

The system is hard to change because every change forces many other changes to other part of the system

Single change causes cascade of subsequent changes in dependent modules

The more modules must be changed the more rigid the design

Examples

- Status fields – New status values
- Dependency on concrete classes

Fragility

Changes cause the system to break in places that have no conceptual relationship to the part that was changed

The tendency of a program to break in many places when a single change is made

New problems in area that have no conceptual relationship with the area that was changed

On every fix the software breaks in unexpected ways

Examples

- Dependency on concrete classes
- No edge cases or bound checks

Immobility

It is hard to disentangle the system into components that can be reused in other systems

It contains parts that could be useful in other systems, but the effort and risk involved separating those parts from original system is too much

The useful modules have too many dependencies

The cost of rewriting is less compared to the risk to separate those parts

Examples

- Too many constants
- Too many global variables
- Too much coupling

Viscosity

Doing things right is harder than doing things wrong

When the design preserving methods are more difficult to use than the hacks, the viscosity of the design is high – the hack is cheaper to implement than the solution within the design

When the development environment is slow and inefficient, developers will be tempted to do wrong things

Examples

- Use of public member fields
- Insufficient documentation of the implemented classes
- Highly complex design
- Low performing code

Needless Complexity

The design contains infrastructure that adds no direct benefit

The design becomes littered with constructs that are never used

Makes the software complex and difficult to understand

Examples

- Too much generic code
- Make use of middle men
- Long hierarchy of interfaces and classes

Needless Repetition

The design contains repeating structures that could be unified under a single abstraction

The problem is due to developer's abuse of cut and paste

It is really hard to maintain and understand the system with duplicate code

Examples

- Make use of switch statements
- Duplicate code in different classes
- Avoiding abstraction – copying the existing code and tweak according to the requirements

Opacity

It is hard to read and understand; It does not express its intent well

The code does not express its intent well

The code is written in a convoluted manner

Example

- Too much commented code
- Incorrect naming convention
- Not well indented code
- Algorithm written in a complex manner

Signs of Good Design

Adaptability	• The design is easy to change
Robustness	• The design is hard to break
Reusability	• The design can be reused
Fluidity	• It is easy to do the right thing
Simplicity	• The design is the “simplest thing that will work”
Terseness	• No unneeded duplication of code
Perspicuity	• Organized and clear

Design Principles – SOLID

Single Responsibility Principle

Open Close Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

The Single Responsibility Principle

A class should have one reason to change

Responsibility: a reason for change

Why? Because each responsibility is an axis of change

When the requirements change, that change will be manifest through a change in responsibility amongst the classes

If a class assumes more than one responsibility, then there will be more than one reason for it to change

If a class has more than one responsibility, then responsibilities become coupled

This kind of coupling lead to fragile design that break in unexpected ways when changed

The Open – Close Principle

Software entities (classes, functions, modules, etc.) should be open for extension, but closed for modification

Open for extension: This means that the behavior of the module can be extended

Closed for modification: Extending the behavior of a module does not result in changes to the source or binary code of the module

The Liskov Substitution Principle

Subtypes must be substitutable for their base types

A violation of LSP is a latent violation of OCP

The Interface Segregation Principle

Clients should not be forced to depend upon methods which they do not use

Interfaces belong to clients, not to hierarchies

The Dependency Inversion Principle

High level modules should not depend upon low level modules

- Both should depend upon abstraction

Abstractions should not depend upon details

- Details should depend upon abstractions

Depend on Abstractions

- No variable should hold a pointer or reference to a concrete class
- No class should derive from a concrete class
- No method should override an implemented method of any of its base class