



SOLID Design Principles & Implementations


By:

Noha Ahmed Thabet

“Any fool can write code that a computer can understand. Good programmers write code that humans can understand”

Martin Fowler

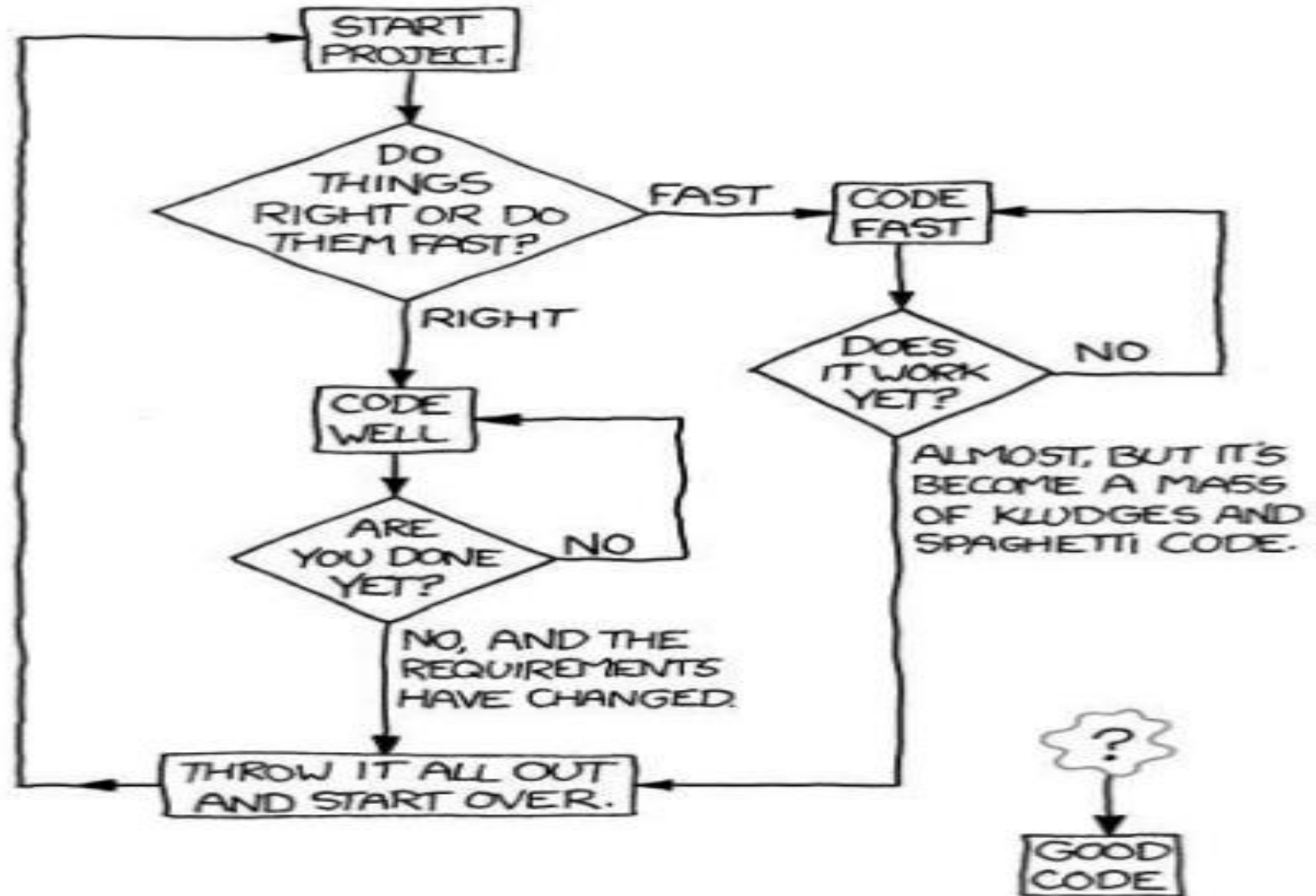




“Walking on water and developing software from a specification are easy if both are frozen.”

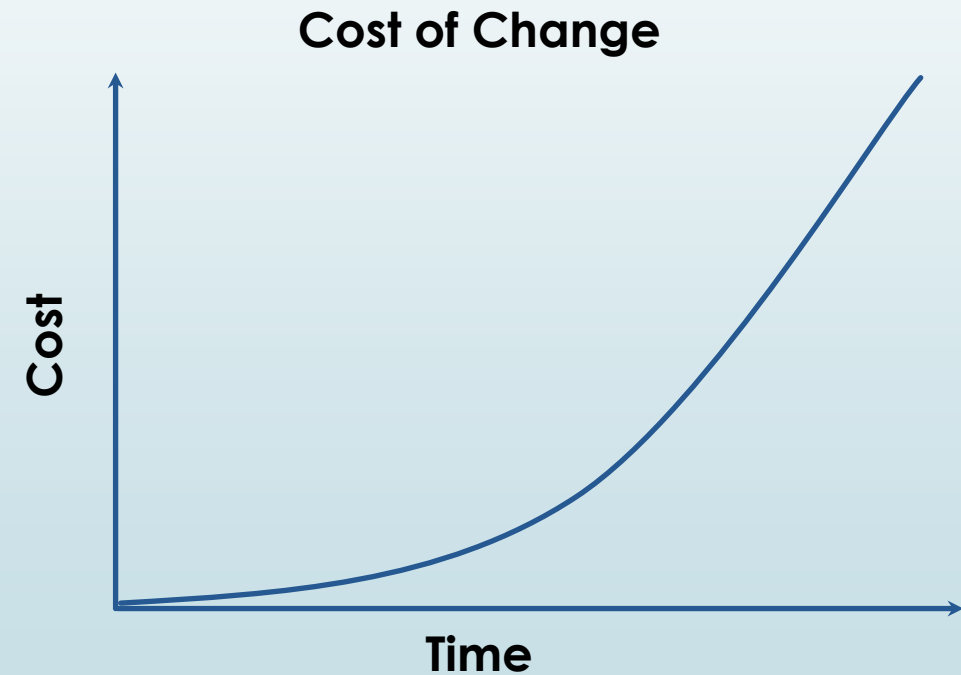
Edward V. Berard

HOW TO WRITE GOOD CODE:



Why do we need a good design ?

- To deliver fast
- To manage change easily
- To deal with complexity





How to identify a bad design ?

➤ Design/Code Smells

- **Rigidity** – The design is difficult to change. Without proper use of abstraction and interface the design becomes very rigid. For each and every small change in the functionality we have to test whole logic from start.
- **Fragility** – Design is easy to break. As discussed with a small change there are very high chances of the whole design going for a toss.
- **Immobility** – The design is difficult to reuse. Or rather we cannot easily extend the current design.
- **Viscosity** – It is difficult to do the right thing.
- **Needless Complexity** – Over design.



How to achieve a good design ?

- **Recommendations:**

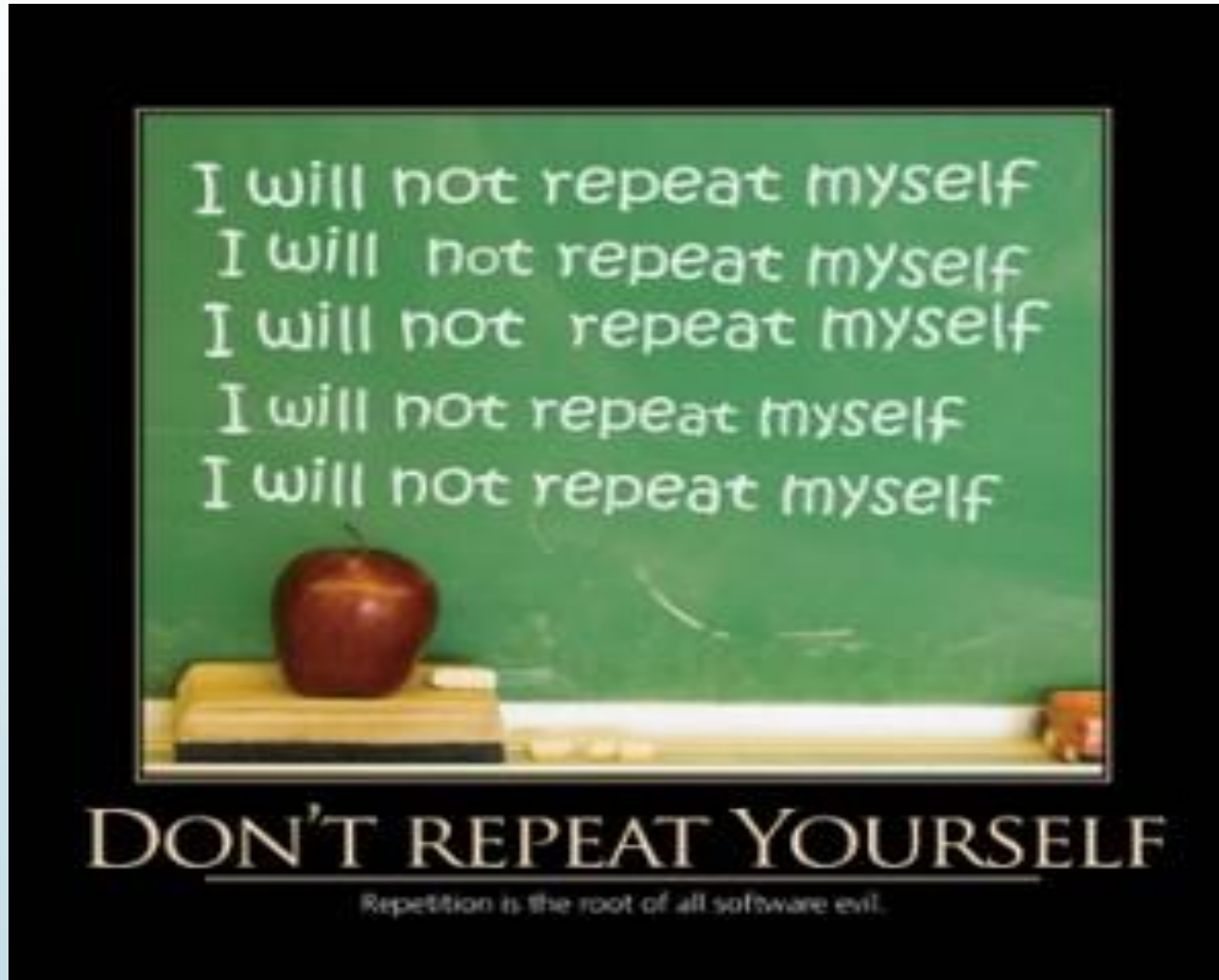
- Follow programming practices of your language/framework
- Follow OO design principles
- Use design patterns



Principles of OO Design

0. Don't Repeat Yourself (DRY)
1. **S** - Single Responsibility Principle (SRP)
2. **O** - Open Closed Principle (OCP)
3. **L** - Liskov Substitution Principle (LSP)
4. **I** - Interface Segregation Principle (ISP)
5. **D** - Dependency Inversion Principle (DIP)

Don't Repeat Yourself ... DRY





Don't Repeat Yourself ... DRY (Cont.)

- "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system". – [Wikipedia](#)

“The principle has been formulated by [Andy Hunt](#) and [Dave Thomas](#) in their book [The Pragmatic Programmer](#)”

Related Fundamentals:

- The [Open/Closed Principle](#) only works when DRY is followed.
- The [Single Responsibility Principle](#) relies on DRY.



SOLID design principles

- SOLID is not a framework
- SOLID is not a library. It is not technology bound. It is not pattern.
- Most of the code written today, though they are written in object oriented way but they are not object oriented and more or less procedural. Writing code in OO language is not a guarantee that the code will be OO.
- But we can make our code OO by using solid principles. We can more productive by using SOLID.
- SOLID can come to rescue when all the requirements are not upfront while developing code.
- Code is more maintainable and understandable by using SOLID.
- When code suffer from design smell then SOLID is answer.



The Single Responsibility Principle

- SRP Definition
- The Problem
- Case Study
- Refactoring to Apply SRP
- Related Fundamentals

SRP: Single Responsibility Principle



SINGLE RESPONSIBILITY PRINCIPLE

Just Because You Can, Doesn't Mean You Should



SRP: Single Responsibility Principle

“The Single Responsibility Principle states that every object should have a single responsibility, and that responsibility should be entirely encapsulated by the class.” – [Wikipedia](#)

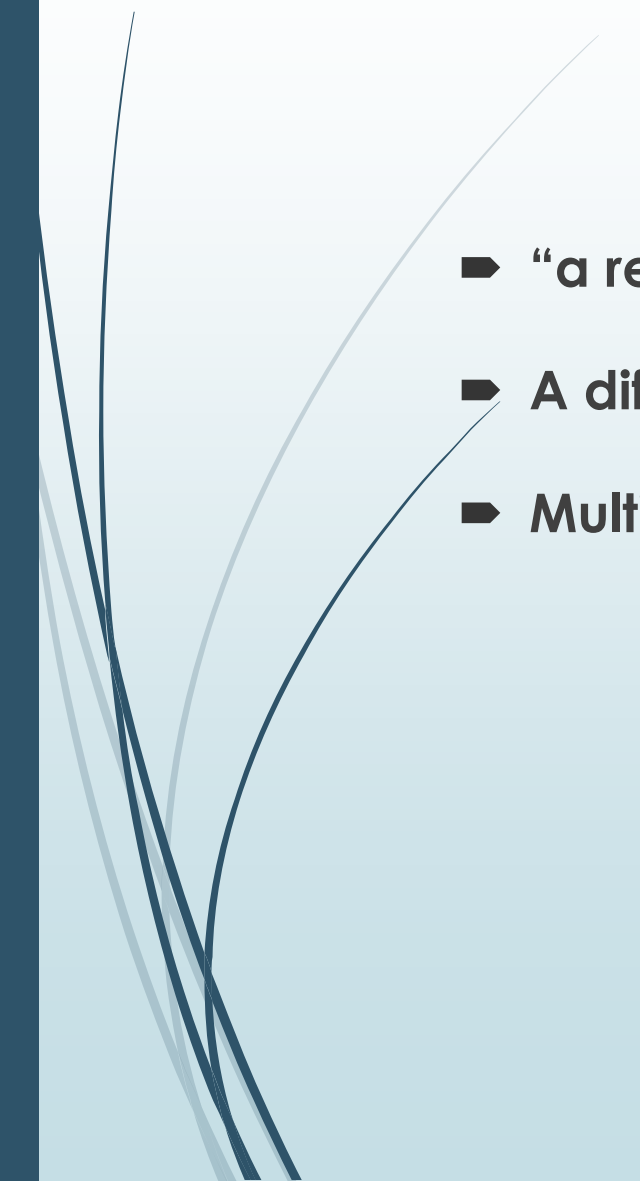
“There should never be more than one reason for a class to change.”

Robert C. “Uncle Bob” Martin

“There is always only one reason to change...changing requirements”



What is a Responsibility?

- “a reason to change”
 - A difference in usage scenarios from the client’s perspective
 - Multiple small interfaces (follow ISP) can help to achieve SRP
- 

Coupling

“The degree to which each program module relies on each one of the other modules” – [Wikipedia](#)



Cohesion

“A measure of how strongly-related and focused the various responsibilities of a software module are” – [Wikipedia](#)





Responsibilities are Axes of Change

- Requirements changes typically map to responsibilities
- More responsibilities == More likelihood of change
- Having multiple responsibilities within a class couples together these responsibilities
- The more classes a change affects, the more likely the change will introduce errors.



Case Study

The Problem With Too Many Responsibilities
Refactoring to a Better Design



Summary

- *“Strive for low coupling and high cohesion!”*
- Following SRP leads to lower coupling and higher cohesion
- Many small classes with distinct responsibilities result in a more flexible design
- **Related Fundamentals:**
 - Open/Closed Principle
 - Interface Segregation Principle
 - Separation of Concerns



The Open/Closed Principle

- OCP Definition
- The Problem
- Case Study
- Refactoring to Apply OCP
- Related Fundamentals

The Open/Closed Principle




OPEN CLOSED PRINCIPLE

Open Chest Surgery Is Not Needed When Putting On A Coat



The Open/Closed Principle

“The Open/Closed Principle states that software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.” – [Wikipedia](#)





The Open/Closed Principle

- **Open for Extension**
 - New behavior can be added in the future
- **Closed for Modification**
 - Changes to source or binary code are not required

“**Dr. Bertrand Meyer** originated the OCP term in his 1988 book, *Object Oriented Software Construction*”



Change behavior without changing code?

- Rely on abstractions
- No limit to variety of implementations of each abstraction
- In .NET, abstractions include:
 - Interfaces
 - Abstract Base Classes
- In procedural code, some level of OCP can be achieved via parameters



Case Study

The Problem

Refactoring to a Better Design




Summary

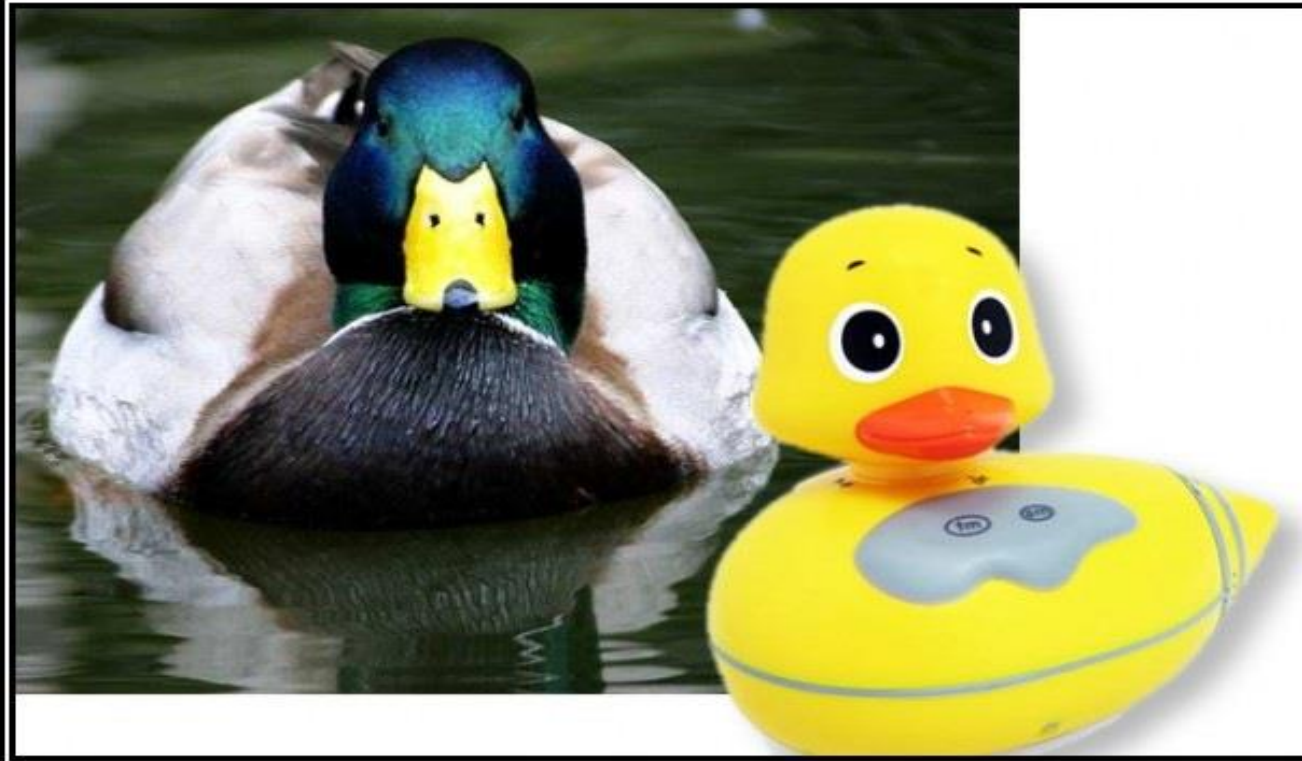
- Conformance to OCP yields flexibility, reusability, and maintainability
- Know which changes to guard against, and resist premature abstraction
- **Related Fundamentals:**
 - Single Responsibility Principle
 - Strategy Pattern
 - Template Method Pattern



The Liskov Substitution Principle

- LSP Definition
 - The Problem
 - Case Study
 - Refactoring to Apply LSP
 - Related Fundamentals
- 

The Liskov Substitution Principle



LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction



LSP: The Liskov Substitution Principle

“The Liskov Substitution Principle states that Subtypes must be substitutable for their base types.”

➤ Agile Principles, Patterns, and Practices in C#

“Named for **Barbara Liskov**, who first described the principle in 1988.”



Substitutability



- Child classes must not:
 - Remove base class behavior
 - Violate base class invariants
- And in general must not require calling code to know they are different from their base type.



Inheritance and the IS-A Relationship

- Naïve OOP teaches use of IS-A to describe child classes' relationship to base classes
- LSP suggests that IS-A should be replaced with
 - IS-SUBSTITUTABLE-FOR



Invariants

- Consist of reasonable assumptions of behavior by clients
- Can be expressed as preconditions and postconditions for methods
- Frequently, unit tests are used to specify expected behavior of a method or class
- To follow LSP, derived classes must not violate any constraints defined (or assumed by clients) on the base classes



Case Study

The Problem

Refactoring to a Better Design



LSP Tips

- **Consider Refactoring to a new Base Class**
 - Given two classes that share a lot of behavior but are not substitutable...
 - Create a third class that both can derive from
 - Ensure substitutability is retained between each class and the new base

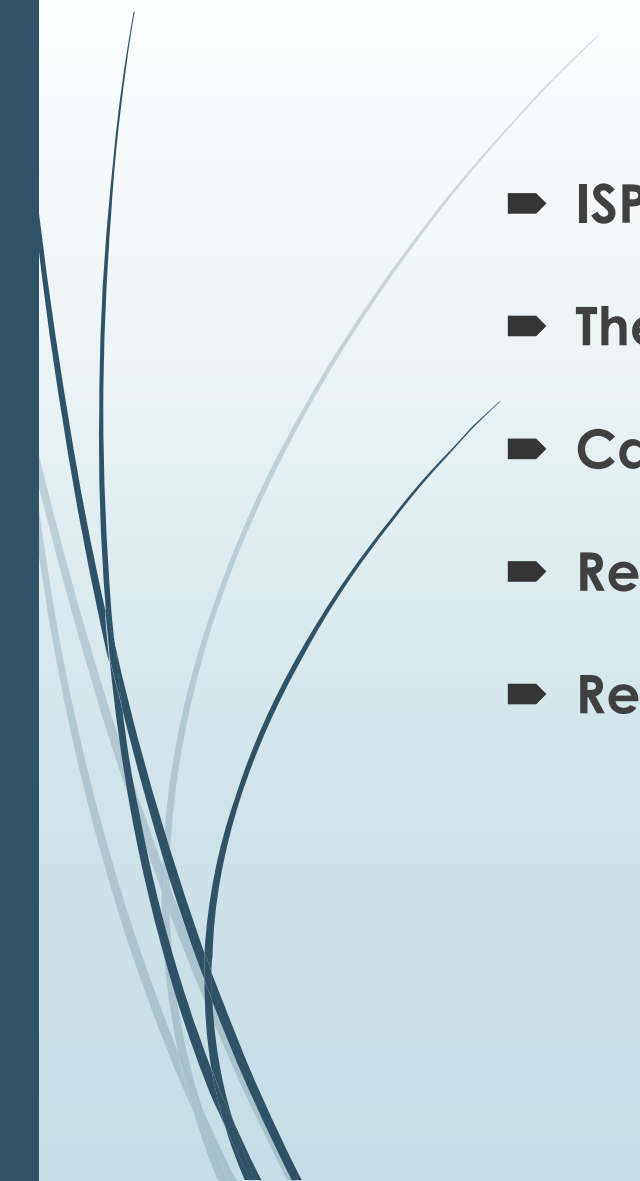


Summary

- Conformance to LSP allows for proper use of polymorphism and produces more maintainable code
- Remember IS-SUBSTITUTABLE-FOR instead of IS-A
- **Related Fundamentals:**
 - Polymorphism
 - Inheritance
 - Interface Segregation Principle
 - Open/Closed Principle



The Interface Segregation Principle

- 
- **ISP Definition**
 - **The Problem**
 - **Case Study**
 - **Refactoring to Apply ISP**
 - **Related Fundamentals**

The Interface Segregation Principle



INTERFACE SEGREGATION PRINCIPLE

You Want Me To Plug This In, Where?

The Interface Segregation Principle


- *The Interface Segregation Principle states that Clients should not be forced to depend on methods they do not use.*
- Agile Principles, Patterns, and Practices in C#
- Corollary:
 - Prefer small, cohesive interfaces to “fat” interfaces



Case Study

The Problem

Refactoring to a Better Design



Interface Segregation violations result in classes that depend on things they do not need, increasing coupling and reducing flexibility and maintainability



ISP Smells

- Unimplemented interface methods:

```
public override string ResetPassword(  
    string username, string answer)  
{  
    throw new NotImplementedException();  
}
```

- Remember these violate Liskov Substitution Principle!

When do we fix ISP?

- **Once there is pain**
 - If there is no pain, there's no problem to address.
- **If you find yourself depending on a “fat” interface you own**
 - Create a smaller interface with just what you need
 - Have the fat interface implement your new interface
 - Reference the new interface with your code
- **If you find “fat” interfaces are problematic but you do not own them**
 - Create a smaller interface with just what you need



Summary

- Don't force client code to depend on things it doesn't need
- Keep interfaces lean and focused
- Refactor large interfaces so they inherit smaller interfaces
- **Related Fundamentals:**
 - Polymorphism
 - Inheritance
 - Liskov Substitution Principle
 - Façade Pattern



References



- Clean Code by Robert C. Martin
- Agile Principles, Patterns, and Practices by Robert C. Martin and Micah Martin
- <http://www.oodeesign.com/design-principles.html>
- The Principles of OOD
<http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>
- <http://en.wikipedia.org/>