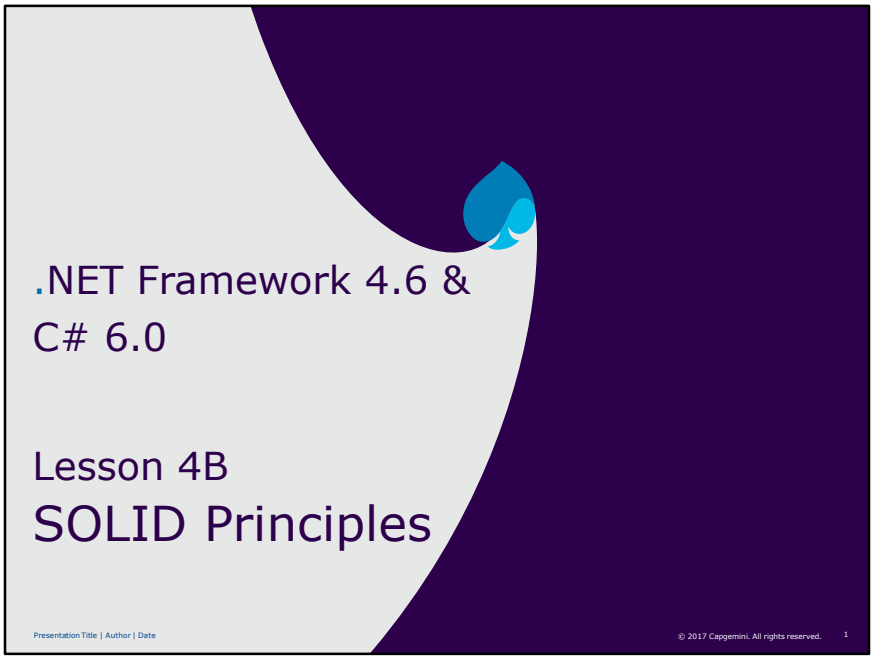
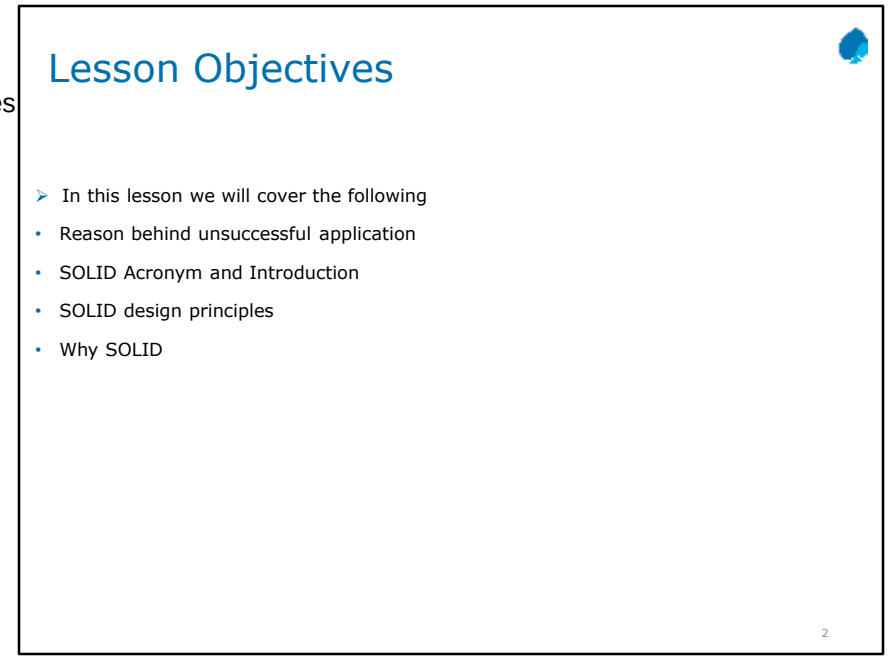

.NET Framework 4.6 &
C# 6.0
Lesson 4B
SOLID Principles
Presentation Title | Author | Date
© 2017 Capgemini. All rights reserved.
1

Add instructor notes here.



Reason behind unsuccessful application

- Developers build application with good and tidy design using their knowledge and experience
- But over the time application develop bugs.
- To overcome this problem , the application must be altered for every change request or new feature request
- Due to this developer needs to put in a lot of efforts, even for simple task and it might require full working knowledge of the entire system
- This is due to change and new feature request which is the part of the Software Development process.

3

- So, what is the reason for this?
- The reason for this is the design of the application

Reason behind unsuccessful application

- The following are the design flaws that cause the damage in software, mostly.
  - Putting more stress on classes by assigning more responsibilities to them. (A lot of functionality not related to a class.)
  - Forcing the classes to depend on each other. If classes are dependent on each other (in other words tightly coupled), then a change in one will affect the other.
  - Spreading duplicate code in the system/application.

4

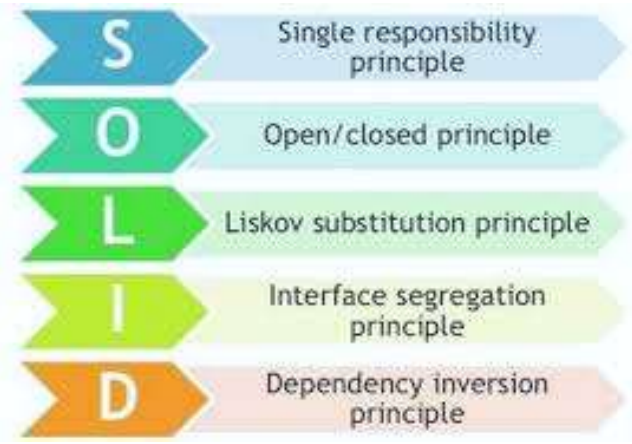
Traditional Web Application Development

- Solution
  - Choosing the correct architecture (in other words MVC, 3-tier, Layered, MVP, MVVP and so on).
  - Following Design Principles.
  - Choosing correct Design Patterns to build the software based on it's specifications.
  - SOLID is one of the design principle which can be used for application development.

SOLID Introduction

- SOLID principles are the design principles that enable us to manage with most of the software design problems.
- Robert C. Martin compiled these principles in the 1990s.
- These principles provide us ways to move from tightly coupled code and little encapsulation to the desired results of loosely coupled and encapsulated real needs of a business properly.

SOLID Acronym

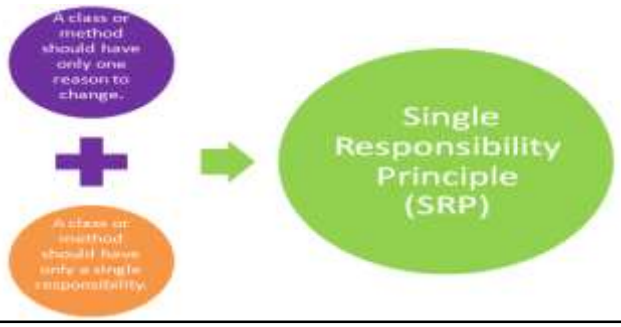


7

- S : Single Responsibility Principle (SRP)
- O : Open Closed Principle (OCP)
- L : Liskov Substitution Principle (LSP)
- I : Interface Segregation Principle (ISP)
- D: Dependency Inversion Principle (DIP)

Single Responsibility Principle

- Robert C Martin expresses the principle as "A Class should have only one reason to change"
- Every Module or class should have responsibility over a single part of the functionality provided by the software and that responsibility should be entirely encapsulated by the class



8

The Single Responsibility Principle gives us a good way of identifying classes at the design phase of an application and it makes you think of all the ways a class can change. A good separation of responsibilities is done only when the full picture of how the application should work.

Open Closed Principle

- "Software entities should be open for extension , but closed for modification"
- The design and writing of the code should be done in a way that new functionality should be added with minimum changes in the existing code.
- The design should be done in way to allow the adding of new functionality as new classes keeping as much as possible existing code unchanged.

9

Here "Open for extension" means, we need to design our module/class in such a way that the new functionality can be added only when new requirements are generated. "Closed for modification" means we have already developed a class and it has gone through unit testing. We should then not alter it until we find bugs. As it says, a class should be open for extensions, we can use inheritance to do this.

Liskov Substitution Principle

- Introduced by Barbara Liskov
- "Objects in a program should be replaceable with instance of their subtypes without altering the correctness of that program"
- If a program module is using a Base class then the reference to the Base class can be replaced with a Derived class without affecting the functionality of the program module
- We can also state that Derived types must be substitutable for their base types

This principle is just an extension of the Open Close Principle and it means that we must ensure that new derived classes extend the base classes without changing their behavior. I will explain this with a real world example that violates LSP.

A father is a doctor whereas his son wants to become a cricketer. So here the son can't replace his father even though they both belong to the same family hierarchy.

Interface Segregation Principle

- "Many client-specific interfaces are better than one general purpose interface"
- We should not enforce clients to implement interfaces that they don't use , instead of creating one big interface we can break down it to smaller interfaces

11

An interface should be more closely related to the code that uses it than code that implements it. So the methods on the interface are defined by which methods the client code needs than which methods the class implements. So clients should not be forced to depend upon interfaces that they don't use.

Like classes, each interface should have a specific purpose/responsibility (refer to SRP). You shouldn't be forced to implement an interface when your object doesn't share that purpose. The larger the interface, the more likely it includes methods that not all implementers can do. That's the essence of the Interface Segregation Principle. Let's start with an example that breaks ISP. Suppose we need to build a system for an IT firm that contains roles like TeamLead and Programmer where TeamLead divides a huge task into smaller tasks and assigns them to his/her programmers or can directly work on them.

Dependency Inversion Principle

- One should "depend upon abstractions not concretions"
- Abstraction should not depend on the details where as the details should depend on abstractions
- High level modules should not depend on low level modules

12

The Dependency Inversion Principle (DIP) states that high-level modules/classes should not depend upon low-level modules/classes. Both should depend upon abstractions. Secondly, abstractions should not depend upon details. Details should depend upon abstractions.

High-level modules/classes implement business rules or logic in a system (application). Low-level modules/classes deal with more detailed operations, in other words they may deal with writing information to databases or passing messages to the operating system or services.

A high-level module/class that has dependency on low-level modules/classes or some other class and knows a lot about the other classes it interacts with is said to be tightly coupled. When a class knows explicitly about the design and implementation of another class, it raises the risk that changes to one class will break the other class. So we must keep these high-level and low-level modules/class loosely coupled as much as we can. To do that, we need to make both of them dependent on abstractions instead of knowing each other.

Why SOLID

- If we don't follow SOLID Principles then We
  - End up with tight or strong coupling of the code with many other modules/application
  - Tight coupling causes time to implement any new requirement, features or any bug fixes and some times it creates unknown issues
  - End up with a code which is not testable
  - End up with duplication of code
  - End up creating new bugs by fixing another bug
  - End up with many unknown issues in the application development cycle

13

Why SOLID

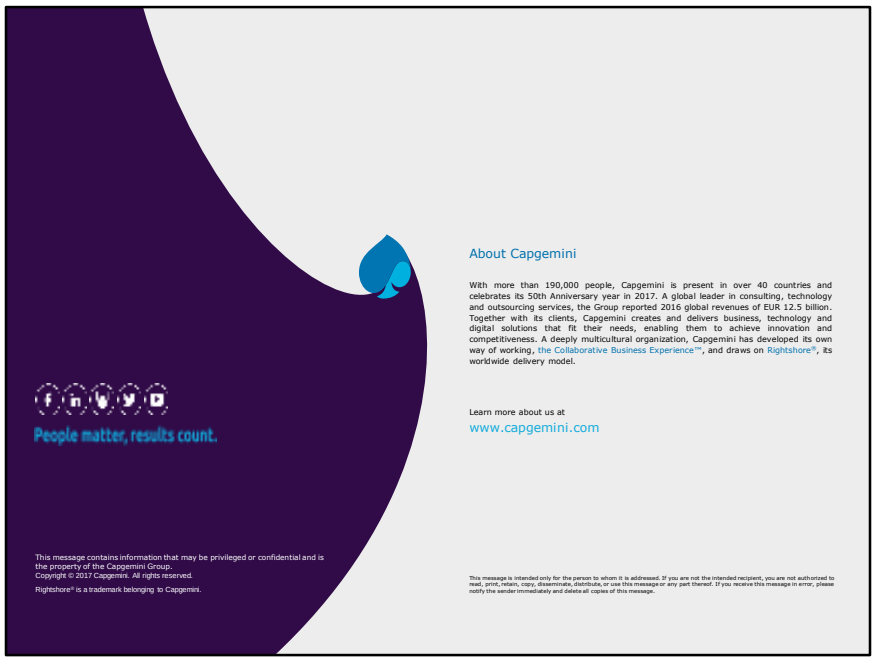
- SOLID Principles helps us to
  - Achieve reduction in complexity of code
  - Increase readability , extensibility and maintenance
  - Reduce error and implement reusability
  - Achieve better testability
  - Reduce tight coupling

Demos

Summary

- In this Lesson we learned,
- What are SOLID Principles
- When and How to apply the principles
- Why SOLID principles are needed
- Advantages of following the SOLID Principles

16


People matter, results count.
This message contains information that may be privileged or confidential and is the property of the Capgemini Group.
Copyright © 2017 Capgemini. All rights reserved.
Rightshore® is a trademark belonging to Capgemini.
About Capgemini
With more than 190,000 people, Capgemini is present in over 40 countries and celebrates its 50th Anniversary year in 2017. A global leader in consulting, technology and outsourcing services, the Group reported 2016 global revenues of EUR 12.5 billion. Together with its clients, Capgemini creates and delivers business, technology and digital solutions that fit their needs, enabling them to achieve innovation and competitiveness. A deeply multicultural organization, Capgemini has developed its own way of working, the Collaborative Business Experience™, and draws on Rightshore®, its worldwide delivery model.
Learn more about us at
www.capgemini.com
This message is intended only for the person to whom it is addressed. If you are not the intended recipient, you are not authorized to read, print, retain, copy, disseminate, distribute, or use this message or any part thereof. If you receive this message in error, please notify the sender immediately and delete all copies of this message.