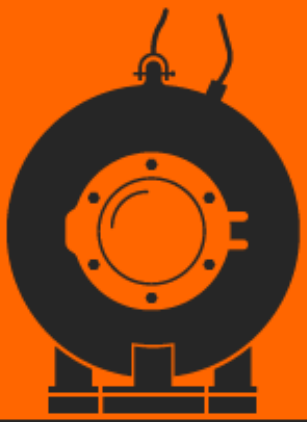


SOLID Principles

Succinctly®

by Gaurav Kumar Arora

Syncfusion | Technology Resource Portal

SOLID Principles Succinctly

By

Gaurav Kumar Arora

Foreword by Daniel Jebaraj

Copyright © 2016 by Syncfusion Inc.

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA
All rights reserved.

Important licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

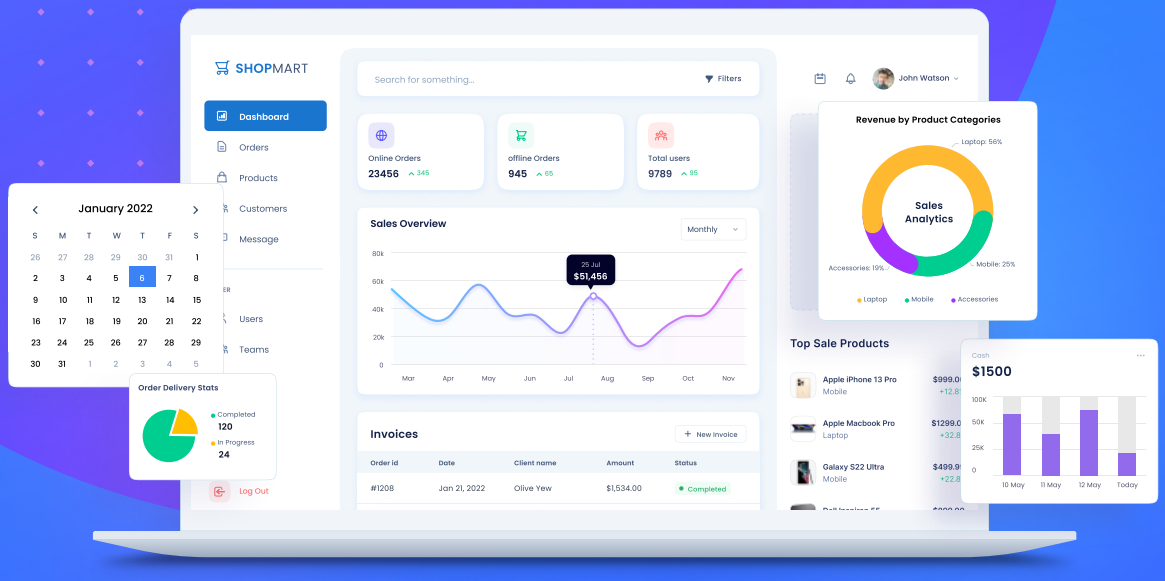
Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: James McCaffrey
Content Editor: Jacqueline Beringer, content producer, Syncfusion, Inc.
Copy Editor: John Elderkin
Acquisitions Coordinator: Morgan Weston, marketing coordinator, Syncfusion, Inc.
Proofreader: Darren West, content producer, Syncfusion, Inc.

THE WORLD'S BEST
UI COMPONENT SUITE
FOR BUILDING
POWERFUL APPS

4.6 out of 5 stars



GET YOUR FREE .NET AND JAVASCRIPT UI COMPONENTS

syncfusion.com/communitylicense

- 1,700+ components for mobile, web, and desktop platforms
- Support within 24 hours on all business days
- Uncompromising quality
- Hassle-free licensing
- 28000+ customers
- 20+ years in business

Trusted by the world's leading companies

Apple IBM SIEMENS intel VISA

Syncfusion

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet, and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just like everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

In the memory of

my angel

Kanchan
(1986–1997)

About the Author

Gaurav Kumar Arora is a Microsoft Most Valuable Professional. He has more than 15 years of experience in software development and application support using Microsoft technologies in construction, finance, aeronautics, media, insurance, and healthcare. He is a Microsoft Certified Technology Specialist and a Certified Scrum Master, and he is XEN certified for ITIL-F and APMG certified for PRINCE-F and PRINCE-P.

Gaurav currently works as a solution architect with Pyramid IT Consulting Private, Ltd. He is responsible for designing application architecture and providing solutions for technical challenges, continuous integration, and deployment.

Gaurav has an MPhil in computer science and enjoys learning new processes and technologies, reading, and sharing his experience with people.

A lifetime member of the Computer Society of India (CSI), Gaurav is a member of numerous technical communities, including CodeProject, DotNetSpider, and C# Corner, and he is a mentor at Indiamentor.

Gaurav is a writer, blogger, speaker, and open source contributor. You can find him on Twitter at @g_arora and at his blog at http://gaurav-arora.com. To discuss this e-book, email him at gaurav@gaurav-arora.com.

Introduction

The acronym SOLID stands for Single Responsibility Principle, Open-Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, and Dependency Inversion Principle. SOLID is a set of software design principles that can be used in conjunction with object-oriented programming and design. Robert C. Martin introduced the underlying concepts of SOLID in 1995 with his 11 commandments of object-oriented programming (OOP). Michael Feathers coined the acronym SOLID in the early 2000s as a mnemonic device for remembering the first five concepts.

Because SOLID is a set of design principles, it does not teach people how to program. Instead, SOLID principles help programmers write better code.

This e-book is not about design patterns or implementing an object-oriented paradigm with the use of any programming-language code. It was not written to guide you to write code in a particular way. My intention is to present information that gives readers ideas for taking their coding or programming from good to better and from better to best. This e-book will be short and concise so that readers can better understand the power of SOLID programming principles and their usage in code.

Any practical, day-to-day programmers—developers, architects, solution providers, consultants, and anyone who is writing a bit of code—should find this e-book useful, although you will need some understanding of the following:

- The basic concepts of any programming languages.
- The basic ideas of OOP.
- The basic fundamentals of writing or representing code, programs, and code snippets.
- The basic ideas of C# language (in order to understand the code-snippet examples).

Note: All code snippets in this e-book are in C# language.

How and where to get source code

Each chapter includes source code in working condition. Complete source code by chapter is available at https://bitbucket.org/syncfusiontech/solid-succinctly.

Source code in other programming languages

If you need these examples in languages other than the C# I am using, please feel free to contact me.

Chapter 1 Why SOLID?

Let's start with two questions: What are you writing as a developer? And, is it good code? As you answer, remember that simply following a few design patterns doesn't guarantee that you are writing good code.

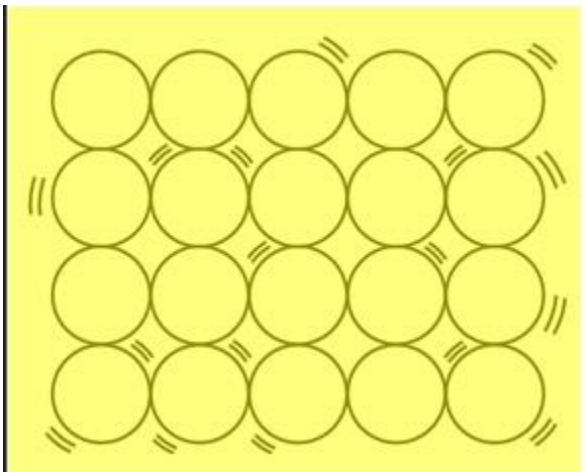


Figure 1: SOLID Image

Figure 1 shows the image that most programmers have in mind regarding their work with SOLID—unstable code loosely held together like a pile of pebbles in a bucket. Programmers typically imagine they are writing good code if they simply adopt a few design patterns, but of course, in reality, that is not enough. While our code isn't going to be as solid as a wall of bricks or a vessel containing pebbles, with the use of SOLID, our code will be more robust and usable.

Frankly, the answer to "Why SOLID?" is simply that SOLID principles produce better code. If you're confused, don't worry—this will become clearer when we study a few live examples. But to sum up my rationale: I use SOLID principles to counteract my bad habits of writing code containing thousands of lines and numerous methods or functions in a class. Yes, I was one of those who intentionally and unintentionally wrote bad code, then had to put in great effort for code review.

Eventually, I made identifying dirty code a priority, and I organized a checklist to help pinpoint my issues. You can use these points to check whether or not you are writing bad, dirty code:

- Does the code implement a design pattern?
- Is the code tightly coupled?
- Is the code testable?
- Is the code human readable?
- Is the code duplicated?

- Is the code too lengthy to understand?

Perhaps you can add more checklist points that suit your own experiences, but whenever I analyze my code using the above checklist points, I am able to discern whether or not my code is dirty. However, this list doesn't tell me that my code is SOLID.

Let's first try to understand how the previous checklist points are helpful for analyzing our code.

Does the code implement a design pattern?

Adopting a design pattern in your code or program is not compulsory. No one can force you to do this. Use your best judgment to decide whether or not you want to implement patterns.

Figure 2: Pattern Image

As a programmer, I can imagine how patterns look, and I can easily relate these to something like a texture on a wall. Figure 2 is simply an image that correlates with the patterns in a program.

Of course, no one can prevent you from writing dirty code, such as opening on a button click and triggering the Save command to persist values in a database. Many of us did this in our student days. I still remember my C programming days, when I was struggling to write a simple console game that required me to save players' scores into a dat file. I'd written everything within the same class using a single function. At the time, that approach was fun for me. But now, when I remember those days, I think about what a silly mistake I made.

Look again at Figure 2 and imagine a plain floor. Now, think of a plain, wet floor. What happens when you walk on a wet floor? You can fall. Now, think of a floor coated with a surface that you can grip with your shoes or feet. On this floor, there is less reason to be afraid because there are fewer chances of falling even if the floor is wet.

Next, imagine you're working on a maintenance project in which the code is unmanaged or dirty (i.e. there is no pattern, are no principles, etc.). It's somewhat like walking on a wet floor, isn't it? On the other hand, if the same project is well managed, following patterns and principles, your life is easier while you're fixing or enhancing the project. Patterns provide an extra, coated layer to your program so that you can write robustly with little chance of falling.

In summary, dirty code is code that needs a lot of refactoring and does not adhere to any patterns or principles.

Do not be hurried, just think twice about your coding decisions. There are more points in the checklist to help you decide about your code.

Tip: Decide which design pattern suits your requirements and implement that design pattern.

Is the code tightly coupled?

From Wikipedia:

"In software engineering, coupling is the manner and degree of interdependence between software modules; a measure of how closely connected two routines or modules are."

In other words, coupling occurs when a module is fully attached to another module.

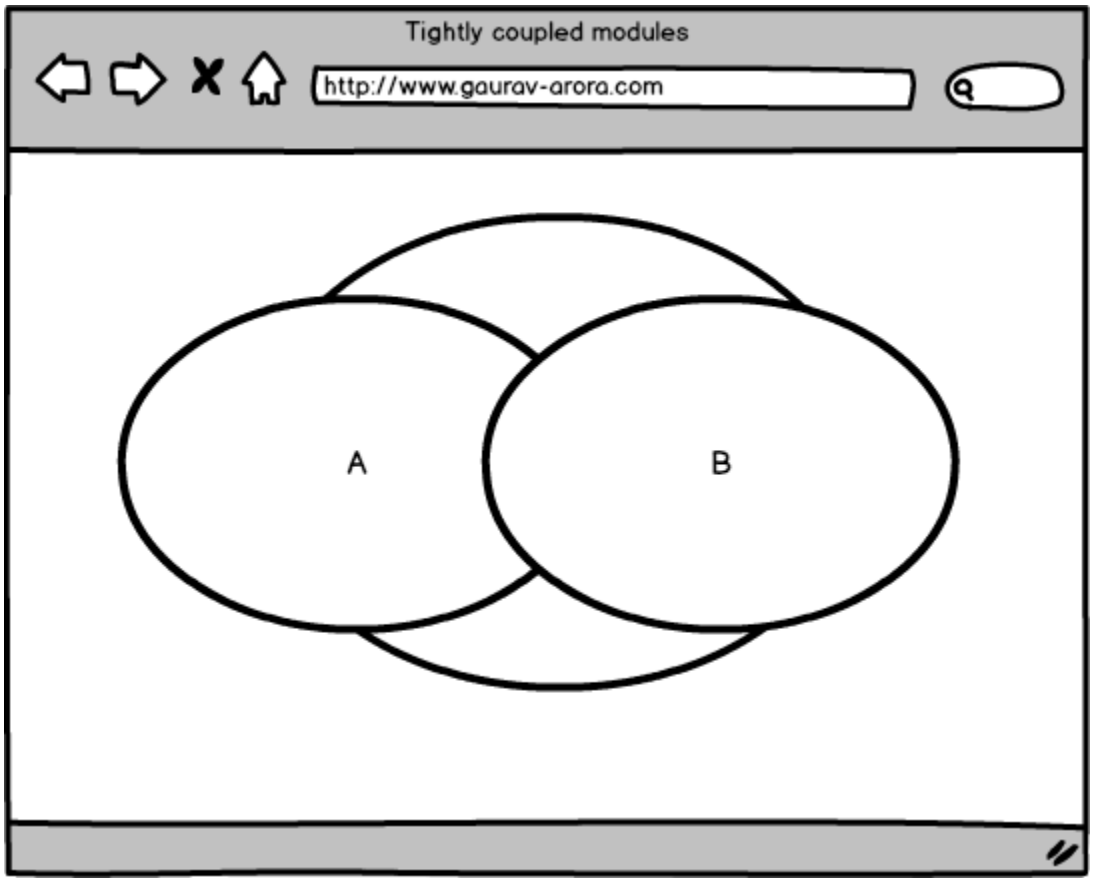


Figure 3: Tightly Coupled Modules

Imagine a scenario in which you're writing a program of math software and you need to implement all possible math operations. You've introduced two modules as a part of your implementation. In this circumstance, you'll need to make changes in both modules whenever you are required to make changes to either module. This means your application or software is tightly coupled, which means your code qualifies as dirty code.

The bottom line—you are writing dirty code if your code modules are dependent.

Is code testable?

It is good practice to always write tests and testable code for your code. Here are some of the advantages you gain by doing so:

- It reduces the reworking time needed due to any unrecovered issue.
- It provides ability-testing code within code.
- It reduces the time needed for debugging code—with the help of test-driven development (TDD) we can easily and quickly locate problems.

- It allows for easy determination of whether or not recent changes or refactoring broke the code.

TDD helps us to improve code quality and draw our focus to the main purpose of our task. It makes sure that new changes are tested and will not break any existing code functionality.

Writing tests may be unpleasant and time-consuming, and you might have a very limited timeframe in which to complete your application or code, but tests are important. Tests and code work together to achieve better code. Writing testable code is a good habit to get into as a way of avoiding future complications.

Is the code human readable?

Because your code will be read by humans, it should be meaningful code.

For example, if you are writing a function that simply returns a sum of two numbers, you should choose a meaningful name that describes this function. That way, before diving into the code, another developer can look into the function and easily understand its purpose. In this case, you might think of a function name like Sum or Add.

Tip: In order to make your method name meaningful, don't hesitate to use a long name such as* Is ValidForCharactersOnly*.

While writing code, give meaningful names to your classes, methods, functions, and module names. Your program will be read by people, not by machines. Machines can understand your code, but you should write so that humans can understand it, too.

Note: Visit your code and see if you can read it. If not, that means you are writing bad code.

Is the code duplicated?

It's possible that you are doing the same things in many places—you are creating something like a business rule to validate any input, and you are writing the same code for each and every input. Revisit and check to see if you have such code. If so, try to move that code to a common place and use it throughout your application.

Tip: Create a* Common *class if you can't decide where to place your common code.

Also remember to avoid and remove any redundant code. It's a good idea to visit your code many times, and in different ways, in order to check for redundancy. For example, you can write `Console.WriteLine("SOLID Principles Succinctly!");` instead of `System.Console.WriteLine("SOLID Principles Succinctly!");` if you have already added 'namespace.' In this case, 'System' is redundant and can be removed.

Let's look at Code Listing 1 in order to better understand redundant code.

Code Listing 1

```
using System;

public class Program
{
    public static void Main()
    {
        //System can be removed as it is redundant code here.
        System.Console.WriteLine("SOLID Principles Succinctly!");

        //This is fine.
        Console.WriteLine("SOLID Principles Succinctly!");
    }
}
```

Is the code too lengthy to understand?

Avoid writing lengthy code. A function can be hard to understand if it contains thousands of lines. Generally, we developers write lots of code that would be better split into pieces like small functions or properties.

Lengthy code always creates confusion, especially for new developers who did not actually write that code. These kinds of programs, functions, and methods might meet stated requirements and produce expected results, but they are nevertheless hard to understand. Luckily for us, these kinds of code generally can be grouped into meaningful functions.

We can mark our code as bad code if we are writing lengthy functions or methods when we could instead break them into small and meaningful functions.

Finally, here is the complete checklist to analyze your code by category as bad or dirty code.

Table 1: Dirty Code Checklist

Dirty Code Checklist	
Checkpoint	**Yes/No**
Design patterns	Yes/No
Coupling	Yes/No
Testable code	Yes/No

Dirty Code Checklist	
Human readable code	Yes/No
Duplicate/redundant code	Yes/No
Lengthy code	Yes/No

You are writing dirty code if your checklist contains a Yes for any of the above points.

Note: Send me more points if you'd like to add to this checklist.

Next, we'll look at how SOLID helps prevent us from writing bad or dirty code.

Chapter 2 Should I Care about SOLID?

Before we examine the many reasons why we should care about SOLID, let's address the reasons people give for not using it:

"I implement OOP."

People who use OOP often say they do not want to use SOLID. However, while OOP provides a paradigm to write programming, paradigms are not principles that instruct you about the kind of responsibilities a class should have. Writing OOP doesn't guarantee that you follow design principles.

So yes, you should care about SOLID even if you're writing object-oriented programming.

"I can implement design patterns."

What are design patterns meant for? These patterns only tell us how to design programs and software. But SOLID principles guide us in making our code better and cleaner.

For example, consider a scenario of observer pattern. It tells us about the pub or sub model, how the receiver or sender gets notified, and how we can design our classes in order to fulfill observer patterns. But it never guides us on making our code cleaner and better placed.

"I use Java, not .NET/C#."

SOLID principles are not related to any programming language, which means they are not built for any *specific* programming languages—the language in which the program has been written doesn't matter. SOLID principles are simply guidelines for making our code and programming robust, even if we are working in JAVA.

"I am an architect."

If someone says these principles are not for architects, they are not being realistic. Architects think about robust design, scalability, and components distribution, and while designing any software or application, they should also be keeping in mind SOLID principles. Yes, there are chances for overlap, but in order to make application design robust, these people in particular should follow SOLID principles:

- A developer who needs to write code.
- A reviewer who reviews code/area.
- A tech lead who guides a team.

“I am working on a maintenance project.”

You should care about SOLID principles even if you are working on a maintenance project. Let me share a relevant experience from my professional life.

A long time ago, I worked on a maintenance project. By the time I joined that team, the project was almost done. I got an assignment and noticed a much-repeated code snippet throughout the application, but the assignment, including QA efforts, was due in two days.

I approached my team manager about the situation, but he said, “We are almost done with this project. I don’t care about the SOLID principles. Besides, we have a green flag from our clients. If you can't complete this task on time, I’ll have to assign it to someone else. The choice is yours.”

That really wasn’t my day.

What did I do? Well, I am a developer, and we developers don't want to leave tasks incomplete. So I completed the assignment before it was due, but I did so in my own style. I wrapped up a new class and added new functionality related to my task, wrote it using SOLID, and attached the similar things with this class so that other areas of code could feel SOLID. When I sent my changes for review, I sent them with these notes:

“I noticed a lot of code is repeating itself. Code at some places is really unmanaged and needs to be cleaned up. I implemented my changes by obeying SOLID principles.”

My changes were approved after QA and deployed with release, but afterward we got a call for code review by the client. A few senior people from our team attended the video conference call. Suddenly, I got a call at my desk from my manager**,** who said, "Come to the conference hall—the client is calling.”

I was shocked when the client asked about my work experience and why I had written the code in this way. Man! I wasn’t sure what to think.

Then the client clapped and said, “Great. You've done it the right way.”

What if I had thought, “I am a developer working on a maintenance project, and my project is undergoing final release. Why should I bother about all these principles?”

“I do QA and work with Selenium.”

I recently attended a seminar at an engineering college (for its science festival), and one QA developer said to me, “I am a quality-assurance person, and I work with Selenium. Why should I care about SOLID?”

My answer is simple. In some companies, the QA developer must write all the code in order to automate his or her QA process. In this scenario, SOLID principles are especially important.

“I am a technical consultant.”

If you’re a technical consultant, it’s your duty not only to provide technical consultancy, but also to guide your client to write good code, which means you must adhere to SOLID.

That dispenses with the mythology about not needing to care about SOLID while writing code, programs, or software.

Table 2: Should I Care about SOLID?

Should I Care about SOLID?	
I am	**Should I?**
An OOP programmer	☒
A developer	☒
A QA-developer	☒
An architect	☒
A code-maintenance checker	☒
A technical consultant	☒

Keep in mind that this list isn’t comprehensive. Depending upon the situation, others involved in coding might also need to follow SOLID principles.

Chapter 3 Before Starting SOLID

SOLID is a set of design principles, and these principles help a programmer write better code. Table 3 shows what the acronym SOLID stands for.

Table 3: SOLID at a Glance

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

Whenever I address SOLID, some people find it confusing, but that's okay—at first the principles can indeed be frustrating. Table 4 depicts the most commonly confusing aspects of the principles.

Table 4: Confusing Aspects of SOLID

Confusing Aspects of SOLID	
Design patterns	Design principles
Object-oriented programming	Design principles

Design patterns vs. design principles

In particular, people get confused between design patterns and SOLID principles. Table 5 addresses the most confusing aspects of this work.

Table 5: Design Patterns vs. Design Principles

Design patterns	Design principles
Are solutions to design problems (we are in the software world, so we'll consider any software problem).	Are guidelines that tell us how to go with a specific design pattern.
Are related to the implementation of the actual problem.	Can be applied anywhere, regardless of the specific context, issue, or problem.

Design patterns	Design principles
Are high-level concerns while working with solutions to any problem.	Are building blocks and used with patterns to achieve goals or to solve problems.
Example: Think about a strategy pattern and you are implementing or following principle (check the right-side column of this table).	Example: In the case of a strategy design pattern, we use SOLID as follows— • **SRP:** Defines code that is responsible for an algorithm and for extracting it from another code. • **OCP:** Represents all different algorithms and uses it. • **LSP:** Doesn't use concrete algorithm classes in client code, only in abstraction.

Why coding style matters

I remember in the old days, when I was learning computer programming, I found it difficult to grasp data structure, algorithms, artificial intelligence, computer architecture, and plenty more. In those days, I didn't take much care with my coding styles. I didn't worry about what kind of messages I was writing to show user or what kind of messy code I was writing. I hated the tedious work of checking my code with perfect check-in messages.

As I've mentioned, I later came to understand how important styles are in coding. In fact, if you are not using any IDE such as Visual Studio, it's a good idea to create your own coding styles or get a coding style guideline from your senior if you're working on a team.

While I was doing code review, I could always identify the developer because all developers have their own unique styles of coding.

 ***Note: For C# coders, refer to: https://msdn.microsoft.com/en-us/library/ff926074.aspx***.

Object-Oriented Analysis and Design (OOAD)

OOAD is a well-known and popular technical approach to designing and analyzing an application system with the use of object orientation and the modeling of the development life cycle. Let's look at a pictorial view of OOAD.

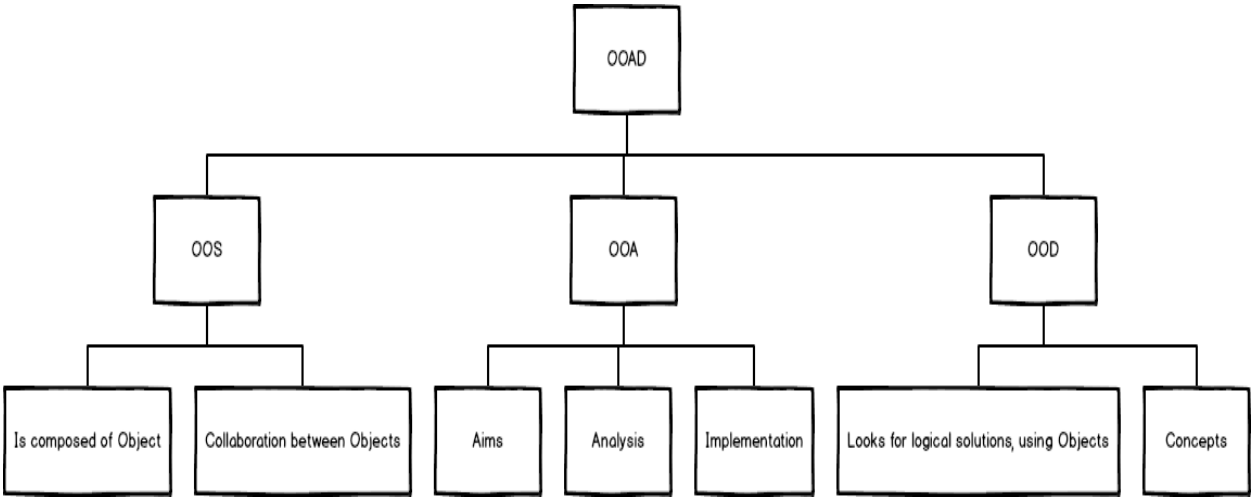


Figure 4: OOAD at a Glance

Note: We are not going into the details of OOAD.

Principles of Object-Oriented Design (OOD)

Object-oriented design principles are not only principles to design with, they are also roadmaps to creating well-mannered design. In the coming chapter, we will learn about SOLID in detail, but for now let's look at an overview of all 11 principles of OOD in the following table.

Table 6: Principles of OOD

SOLID at a Glance—Referring to Class Design	
S for SRP	Single Responsibility Principle
O for OCP	Open-Closed Principle
L for LSP	Liskov Substitution Principle
I for ISP	Interface Segregation Principle
D for DIP	Dependency Inversion Principle
Cohesion at a Glance—Referring to Packages and Namespaces	
R for REP	The Release Reuse Equivalency Principle
C for CCP	The Common Closure Principle
C for CRP	The Common Reuse Principle

Coupling at a Glance—Referring to Packages and Namespaces	
A for ADP	The Acyclic Dependencies Principle
S for SDP	The Stable Dependencies Principle
S for SAP	The Stable Abstractions Principle

The following figure shows another way of understanding the same ideas.

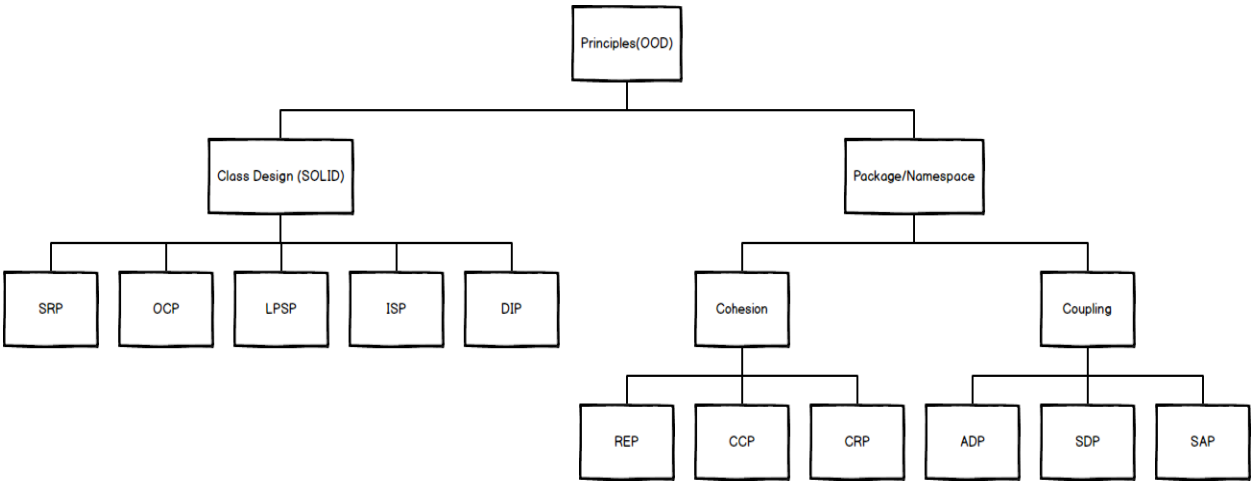


Figure 5: Principles of OOD

We are not going to discuss each and every principle of OOD, but we will go over the last two parts as described by Robert C. Martin before we move forward.

Note: Refer to: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.

Principles of package and namespace cohesion

While it's beyond the scope of this e-book to go into great detail here, the principles of cohesion are summarized in Table 5.

Table 7: Principles of Cohesion

The Release Reuse Equivalency Principle	The granule of reuse is the same as the granule of release. Only components that are released through a tracking system can be effectively reused.
The Common Closure Principle	Classes that change together, belong together.
The Common Reuse Principle	Classes that aren't reused together should not be grouped together.

Principles of package and namespace coupling

Again, we won't go into great detail here, but the principles of package and namespace coupling are summarized in the following table.

Table 8: Principles of Coupling

The Acyclic Dependencies Principle	The dependency structure for released components must be a directed, acyclic graph. There can be no cycles.
The Stable Dependencies Principle	Dependencies between released categories must run in the direction of stability. The dependee must be more stable than the depender.
The Stable Abstractions Principle	The more stable a class category is, the more it must consist of abstract classes. A completely stable category should consist of nothing but abstract classes.

OOP and SOLID

OOP provides a way to write and polish our programs into better shape. These are basic points that guide us in writing a good program.

For great object-oriented designs, we need to think beyond the basics, and SOLID principles provide us a way to achieve great design. Yes, there are certain design patterns that guide us in similar ways, but SOLID exists before these patterns.

So, what is SOLID? We have seen the 11 principles of OOD—those principles are the backbone of our development, design, and analysis of applications or systems. Now, let's briefly discuss each of the SOLID principles.

SOLID principles at a glance

SOLID principles are neither laws nor rules—they are principles intended to help us to write neat code. By following SOLID principles, our code can be easily extended and maintained.

Single Responsibility Principle

A class should have only one responsibility. Let's say our class is responsible for saving data. That means it should not also be responsible for retrieving data or any other tasks.

Open-Closed Principle

Once a class has been written, it should not allow anyone to make changes. No one should be able to go back and amend the class code in order to implement new functionalities.

Liskov Substitution Principle

A dependent object should be able to use any object of type parent object.

Interface Segregation Principle

A smaller interface is recommended. If an interface has one method, there will be only one place to change if we need to change the code. However, in the case of interfaces with more methods, there might be more reasons for change.

Dependency Inversion Principle

Put simply, this addresses loose coupling. With the help of DIP, we can write code that does not depend upon concrete classes.

Chapter 4 Single Responsibility Principle

A class should have a single responsibility.

Let's dive into this ocean. We can read the above statement as: a class should not be designed to do multiple activities. But that begs the question of what kind of activities a class should or should not do.

For example, let’s say I need to design a system that provides me with employee details. This system should include activities, and I rely on Create, Read, Update, and Delete (CRUD) operations. However, according to the Single Responsibility Principle, I need to design several classes that do any one of these operations but not more than one of them.

In the past, particularly when I was learning C++, I wrote thousands of lines of code in one program containing many **`if…else`** statements.

When I was learning this principle, I questioned why a class should not be responsible for multiple activities. In those days I would design a class responsible for modifying data, retrieving data, and saving data. Later, if some kind of business requirement for which our modification or data retrieval logic changed, we would need to change our classes many times and in many places, and this would introduce more bugs and more code changes.

I came to find that keeping classes to one responsibility is really a matter of foreseeing that the responsibilities of classes will be tied to more changes in the future. Today, I don’t like a method that contains more than 4-5 lines. How the world has changed!

We will discuss SRP with a code example. Let’s consider the following:

There is a problem-solving scenario in which we need to import and sync data from different database servers after fetch, scan, and store operations. We also need to save or persist the database on our servers.

Here are the steps we can draft:

1. Contact external servers
2. Sync/fetch data
3. Analyze/scan data
4. Store in temporary storage
5. Save/persist database on local server(s)

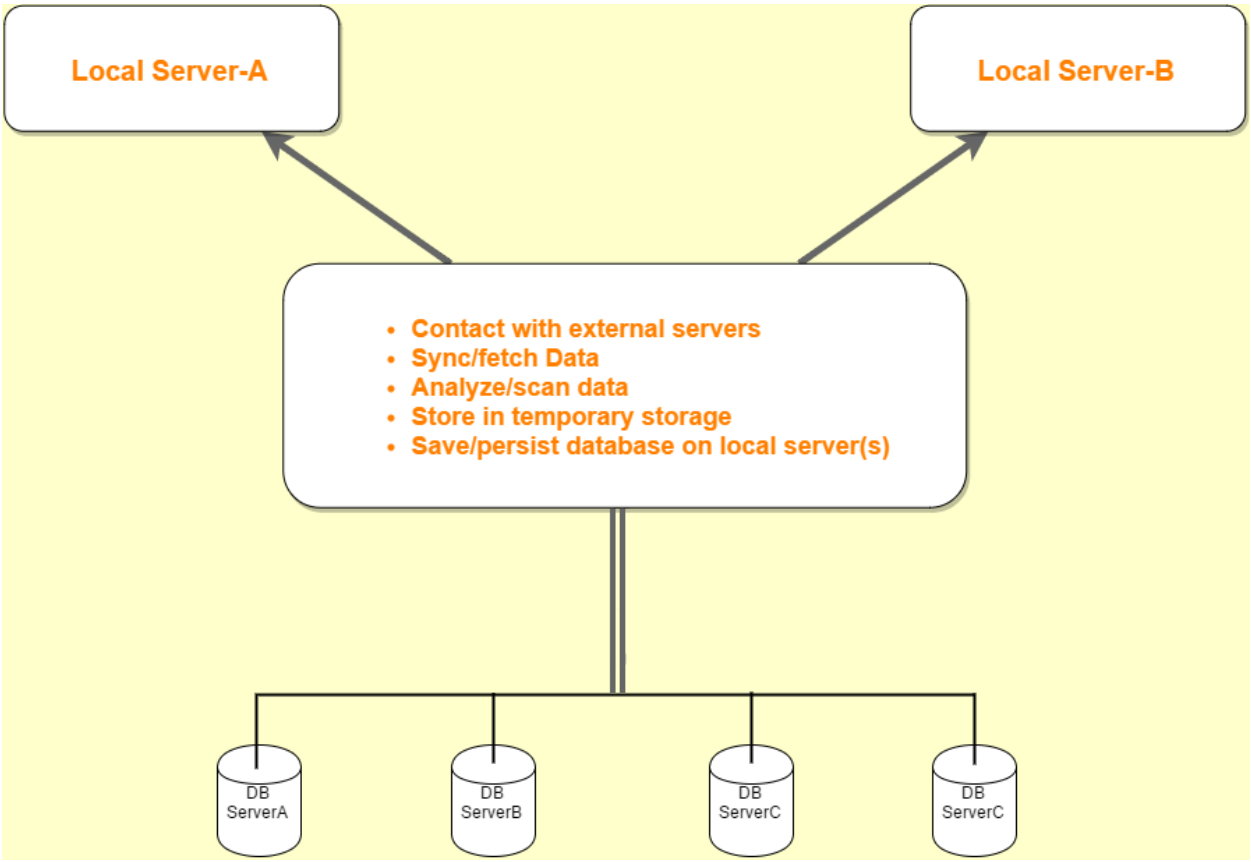


Figure 6: Overview of DataSync App

By referencing Figure 6's overview of our Data Sync App/Utility, we can define the following operations or steps to design our class:

1. Create a class (e.g., **`SyncOperation)`**
2. Add a method to contact external (database) servers (e.g., **`Connect`**)
3. Fetch data method (e.g., **`Sync`**)
4. Call temporary storage (e.g., **`Store`**)
5. Persist data (e.g., **`Save`**)

Note: You can choose whatever class names or method names you like.

After this, we are ready to start designing our class for **`DBSync tool`**.

We will analyze each step of this process—but first, let's consider the following design for our class.

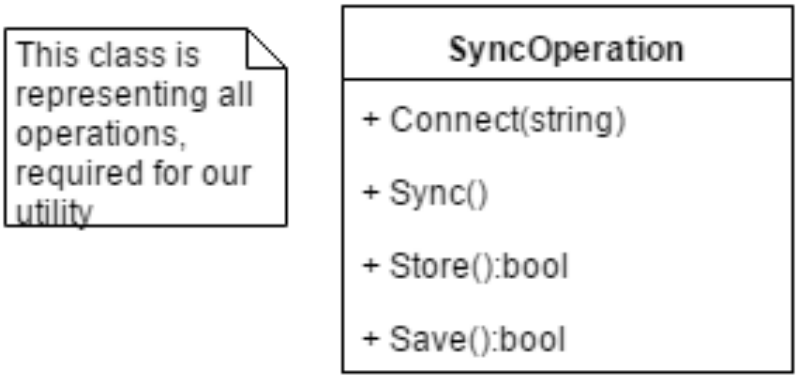


Figure 7: Class Diagram

Using the class design in Figure 7, we can write code as shown in the following code snippet.

Code Listing 2

```
public interface ISyncOperation
{
    void Connect(string serverName);
    void Sync();
    bool Store();
    bool Save();
}
```

With Code Listing 2, we wrote a simple interface **`ISyncOperation`** and defined a certain operational method required for our work-database utility.

Next, we will implement this interface in order to define our actual operation of these methods.

Code Listing 3

```
public class SyncOperation : ISyncOperation
{
    public void Connect(string serverName)
    {
        //TODO—logic to connect external database.
        //Refer to http://www.connectionstrings.com/
    }

    public bool Save()
    {
        //Permanently persist data from temporary storage.
        //This is the final operation.

        return true;
    }
```

```
    public bool Store()
    {
        //Store synced data in temporary storage.
        return true;
    }

    public void Sync()
    {
        //Start syncing external data.
    }
}
```

Can you find what is violating SRP in that code snippet?

Note: Code Listing 3 is not complete. Please refer to Bitbucket for the complete source code.

Next let's see if Code Listing 4 is following SRP.

Code Listing 4

```
namespace SRP
{
    class Program
    {
        static void Main(string[] args)
        {
            //Client stuff goes here.
        }
    }
}


public interface ISyncOperation
{
    void Connect(string serverName);
    void Sync();
    bool Store();
    bool Save();
}
```

At this stage, we have a design in our hands and actual code implementation, so let's go back and read the definition of SRP in order to see how we can apply it. Basically, we have to make a single operational class that must also be a single responsibility class.

Unfortunately, our class has several responsibilities, so it's violating the Single Responsibility Principle.

The best way to follow SRP in Code Listing 4 is to segregate responsibilities. Our class is meant for **Sync** data, which means it should bear only Sync responsibility.

At the very first stage, we have already segregated interfaces, which is a step toward Separating Coupled Responsibilities (SCR).

Note: SCR is beyond the scope of this e-book, hence we will not address the topic here.

Let's look at the next code snippet, which does serve SRP.

Code Listing 5

```
    /// <summary>
    /// This class should be responsible only for the sync operation.
    /// </summary>
    public class SyncOperation
    {
        private IList<ExternalServerData> _data;
        private DataStore _dataStore;
        private DatabaseServer _dbServer;
        private IExternalServerDataRepository _repository;

        public SyncOperation(IList<ExternalServerData> data,
IExternalServerDataRepository repository, DatabaseServer dbServer)
        {
            _data = data;
            _repository = repository;
            _dbServer = dbServer;
            _dataStore = new DataStore(new TempStoreRepository());
        }

        public void Sync()
        {
            //Start syncing of data as per requested server.
            _dataStore.Store(_repository.Sync());

        }
    }
```

In Code Listing 5, our **SyncOperation** class is responsible only for the **Sync** operation. In any case, if we need to customize or change this class, it now deals with only a single responsibility.

Here, our sync class will work on the requested **DatabaseServer**. First, the application connects with an external server and requests a specific database to **Sync** class. **Sync** class will simply sync the data from the external database and send it to **DataStore** class to preserve in temporary storage for further analysis. Afterward, it will be persisted permanently by **Save** class.

Code Listing 6

```
    public class ExternalServerData
    {

        public int Id { get; set; }
        public DateTime InitialDate { get; set; }
        public DateTime EndDate { get; set; }
        public int OrderNumber { get; set; }
        public bool IsDirty { get; set; }
        public string IP { get; set; }
        public int Type { get; set; }
        public int RecordIdentifier { get; set; }

    }
```

Code Listing 7

```
namespace SRP_Follow
{
    public class InternalServerData
    {
        //Stuff goes here.
    }
}
```

Code Listing 7 shows that in **ExternalServerData** class, certain fields are required to sync data from external database servers.

Code Listing 8

```
    public interface IExternalServerDataRepository
    {
        IEnumerable<ExternalServerData> Sync();
    }
```

In Code Listing 8, **IExternalServerDataRepository** interface is merely defined for **Sync** operation.

Code Listing 9

```
    public class ExternalServerDataRepository :
IExternalServerDataRepository
{
    private readonly List<ExternalServerData> _dataList =
GetServerData();

    public IEnumerable<ExternalServerData> Sync()
```

```
    {
        return _dataList;
    }

    //This is sample data.
    private static List<ExternalServerData> GetServerData()
    {
        return new List<ExternalServerData>(new[]
        {
                new ExternalServerData
                {
                    Id = 1,
                    InitialDate= new DateTime(2014,01,01),
                    EndDate= new DateTime(2015,01,30),
                    OrderNumber=1,
                    IsDirty=false,
                    Type = 1,
                    IP ="127.0.0.1"
                },

                new ExternalServerData
                {
                    Id = 2,
                    InitialDate= new DateTime(2014,01,15),
                    EndDate= new DateTime(2015,01,30),
                    OrderNumber=2,
                    IsDirty=true,
                    Type=1,
                    IP ="127.0.0.1"
                }
            });
    }
}
```

In Code Listing 9, **`ExternalServerDataRepository`** implements the sync operation in the previous code snippet (we created fake data for demonstration purposes).

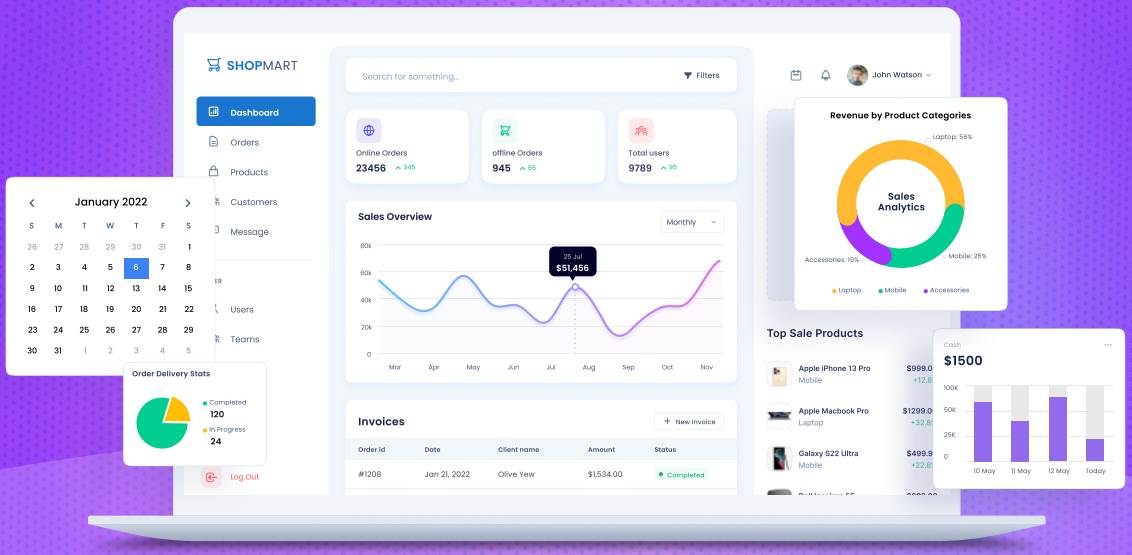
Note: For complete code, please refer to source code repository.

Conclusion

The Single Responsibility Principle, SOLID's first principle, tells us to distribute responsibilities and create classes that stick with only a single responsibility. From the code example, we can conclude that this is a good practice for making our classes decent, neat, and clean. We can think of many related operations a class could bear, but we must think about the drawbacks and implications of classes overburdened with too many responsibilities. Therefore, a single class should be responsible for a single operation.

4.6 out of 5 stars

THE WORLD'S BEST UI COMPONENT SUITE FOR BUILDING POWERFUL APPS



GET YOUR FREE .NET AND JAVASCRIPT UI COMPONENTS

syncfusion.com/communitylicense

- 1,700+ components for mobile, web, and desktop platforms
- Support within 24 hours on all business days
- Uncompromising quality
- Hassle-free licensing
- 28000+ customers
- 20+ years in business

Trusted by the world's leading companies

Syncfusion

Chapter 5 Open-Closed Principle

When I first read about this principle, I thought it meant my class should be open or closed, either one or the other. But then I read the following definition from Wikipedia:

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

I was shocked, wondering how it was possible to make my classes both open and closed and *not* open and closed. In other words, I couldn't understand how to allow things to be modified without doing actual modifications to my object. This sounds confusing, doesn't it?

I dove into OOP for answers. Let's think about abstraction: We can create a base class and a function that can be overridden (i.e., functions with different behaviors). Yes, we can allow changes by keeping objects unchanged.

Next, let's take an example: We need to send emails using various operations. The bodies of emails depend on certain business rules and can contain the same or different messages. Now, what do we need to do here?

We can create a class like **`CreateEmail`**, or whatever you want to name it, with one method—**`BuildMessage`**. So, this class is only responsible for building email messages as per different logic. Because this method can be overridden, I can define its functionality as I choose.

What do you think of the example so far? Is it following OCP or not? In fact, we can't say it is OCP compliant, but it is SRP compliant.

Let's recall the problem-solving scenario we discussed in the preceding chapter. We need to import and sync data from different database servers after fetch, scan, and store operations. Finally, we need to save or persist the database on our servers.

Here are the steps we can draft:

1. Contact external servers
2. Sync/fetch data
3. Analyze/scan data
4. Store in temporary storage
5. Save/persist database on local server(s)

Notice that in step 3 we need to analyze/scan the data. What does this mean? Is it something we need to validate our data? The simple answer is yes. Let's elaborate on this.

We need to import and sync the database or data from external servers to the internal server.

During this operation, we should make sure that the correct and relevant data is being imported and synced. For that, we need to create specific rules—e.g., we need to update our development database server from our production/staging/preproduction database server. So, we need to make sure that the correct data is synced.

Consider a couple of real-time scenarios that could cause problems:

- Our internal server has new tables but the external does not.
- Our internal server's table XYZ column data type got changed from VarChar to nVarChar.

Note: Here we are discussing Microsoft SQL Server 2008R2, which is a Relational Database Management System (RDBMS) developed by Microsoft. Its primary function is to store and retrieve data as per requests coming from the same computer or from computers across a network. Refer to: https://en.wikipedia.org/wiki/Relational_database_management_system to learn more about RDBMS.

Let's write a simple code snippet in Code Listing 10 that will implement the previous scenario.

Code Listing 10

```
namespace OCP_Violation
{
    public class ValidateData
    {
        public void SyncronizeData(ServerData data, SourceServerData
sourceData)
        {
            if (IsValid(data, sourceData))
            {
                //First validate data and then send to persist.
            }
        }

        private bool IsValid(ServerData data, SourceServerData
sourceData)
        {
            var result = false;
            if (data.Type == sourceData.Type)
                result = true;
            if (data.IP != sourceData.IP)
                result = true;
            //Other checks or rules to validate incoming data.
            return result;
        }
    }
}
```

Code Listing 11

```
namespace OCP_Violation
{
    public class SourceServerData
```

```
    {
        public string IP { get; set; }
        public string Type { get; set; }

        //Other stuff goes here.
    }
}
```

Code Listing 12

```
namespace OCP_Violation
{
    public class ServerData
    {
        public string IP { get; set; }
        public string Type { get; set; }

        //Other stuff goes here.
    }
}
```

What's wrong with the preceding code snippet?

In order to find out, let's revisit the class **`ValidateData`**, which was constructed in Code Listing 10.

Our class is doing two things:

- Validating the data.
- Saving the data.

Note: Remember, following OCP, classes shouldn't allow for modifications, but they can be extendable.

Next, let's check if Code Listing 13 follows OCP.

Code Listing 13

```
namespace OCP_Violation
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Unfortunately, this code is not following OCP. Our class **`ValidateData`** is open for new additions or operations, and if we want to extend it, that will be a tedious job.

Let's think of a scenario in which we want to extend this class so that it can use another external service. In such a scenario, as developers we would have no choice but to modify the **`IsValid`** method. Also, if the class must be made a component and provided to third parties for use, these third-party users would have no way to add another service, and that would mean this class is not open for extensions. Also, if we need to modify the behavior in order to persist data, we would need to change the actual class.

To sum up, this class is directly violating OCP because it is neither open for extensions nor closed for modifications.

What does it mean to be open for extension?

It means that our module is flexible enough that we are able to extend its behavior in order to meet the requirements of the application properly.

In Code Listing 13, our validators should be extendable so that any kind of validate should fit with a single validator. We will discuss this further in the upcoming code snippet.

What does it mean to be closed for modification?

It means our code should be inviolate. The source code shouldn't allow changes to be made.

Again note that in Code Listing 13 our **`ValidateData`** class should not allow modification by external sources.

Is it possible to extend behavior without modifying code? We can extend or add our own implementation to the existing code with the use of abstraction. In C# language, we can use abstract classes or interfaces (however, accepting interfaces as pure abstraction depends on the situation—http://blog.ploeh.dk/2010/12/02/Interfacesarenotabstractions/).

Think abstraction

We are not going to discuss abstraction in detail, but as a review, remember that it is an OOP principle providing the facility to create abstract base classes so that we can perform abstraction (using language C#). So, derived classes automatically follow OCP by manipulating an abstraction. Here we can create new classes that are derivatives of the abstract classes, thus extending our modules or classes rather than modifying them.

Let's revisit Code Listing 13. In it, we are simply validating the incoming data with our local data. The main intention here is to validate data. How many validators can you think of?

Unfortunately, our **`ValidateData`** class is trying to get **`IsValid()`**, then trying to persist it in the repositories, which is not a good way to perform. We can say that it's not a good design.

Our next step would be to make our Code Listing 13 example OCP compliant.

This is a good time to rewrite our application or code.

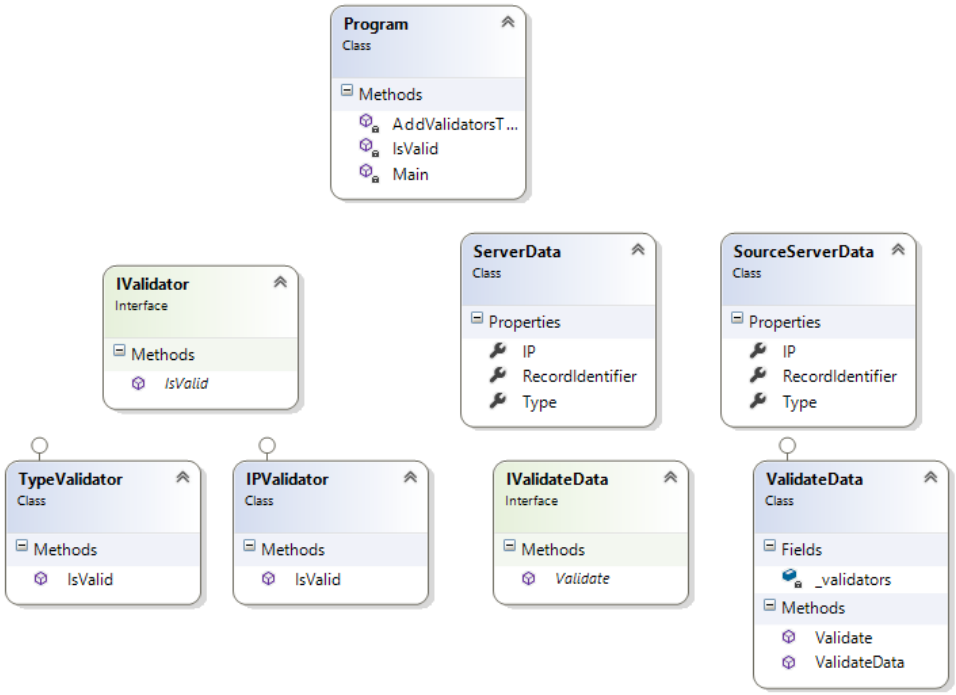


Figure 4: Class Diagram OCP

From the beginning, let's make the code purely abstract for **`DataValidator`** so that we need only to pass a type and it'll go to validate.

First, let's create an interface: **`IValidateData`**.

Code Listing 14

```
public interface IValidateData
{
    bool Validate(ServerData data, SourceServerData sourceData);
}
```

Here, our types of **`IValidateData`** classes or our classes that implement **`IValidateData`** must have the **`Validate()`** method.

In Code Listing 15, let's create a class that implements the **`IValidateData`** interface.

Code Listing 15

```
using System.Collections.Generic;
using System.Linq;

namespace OCP_Follow
{
```

```
    public class ValidateData : IValidateData
    {
        private readonly IEnumerable<IValidator> _validators;
        public ValidateData(IEnumerable<IValidator> validators)
        {
            _validators = validators;
        }
        public bool Validate(ServerData data, SourceServerData
sourceData)
        {
            return _validators.Any(validator => validator.IsValid(data,
sourceData));
        }
    }
}
```

Wait, how does this follow OCP?

Here, we can extend the behavior of our validators (it is a type of **`IValidator`**). We have to stick with the **`Validate()`** method. In this scenario, we can't modify the source code, but we can extend our validators as we are passing a list of validators of the **`IValidator`** type.

Code Listing 16 shows how we can define various validators.

Code Listing 16

```
namespace OCP_Follow
{
    public interface IValidator
    {
        bool IsValid(ServerData data, SourceServerData sourceData);
    }
}
```

We've defined a type of validator, so now let's define our actual validators in Code Listing 17.

Code Listing 17

```
namespace OCP_Follow
{
    public class TypeValidator : IValidator
    {
        public bool IsValid(ServerData data, SourceServerData sourceData)
        {
            return data.Type == sourceData.Type;
        }
    }
}
```

Here, **`TypeValidator`** `is` a type of **`IValidator`**, which means there is nothing much to go beyond this type.

Tip: Keep these terms in mind while implementing OCP: abstraction, closures, extenders.

Let's create a client in Code Listing 18 and see how our validators work.

Code Listing 18

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace OCP_Follow
{
    class Program
    {
        static void Main(string[] args)
        {
            //Only for demonstration purposes.
            var sourceServerData = new List<SourceServerData>();
            var serverData = new List<ServerData>();
            foreach (var data in serverData)
            {
                var sourceData = sourceServerData.FirstOrDefault(s =>
s.RecordIdentifier == data.RecordIdentifier);

                var isValid = IsValid(data, sourceData);
                Console.WriteLine("Record with Id {0} is {1}",
data.RecordIdentifier, isValid);
            }

            Console.ReadLine();

        }

        private static bool IsValid(ServerData data, SourceServerData
sourceData)
        {
            List<IValidator> validators = AddValidatorsToValidate();
            IValidateData validateData = new ValidateData(validators);
            return validateData.Validate(data, sourceData);
        }

        private static List<IValidator> AddValidatorsToValidate()
        {
            return new List<IValidator>
            {
```

```
                new IPValidator(),
                new TypeValidator()
            };
        }
    }
}
```

The code snippet in Code Listing 18 is completely understandable. It does not require any further explanation. If you think differently, contact me, as I would love to discuss this.

In Code Listing 18, we have a **`ValidateData`** class that is responsible only for validating data by certain validations or rules.

With the changes made, our class is now more stable and robust; we can add as many validators as we want. We can also use this validator to save our data.

Another way we can save the data is by calling this validator from another class, and it could be a repository class or our own custom class in which we persist our data.

Conclusion

In this chapter, we examined the Open-Closed Principle, the second principle in SOLID, which tells us how to handle code base while interacting with or writing for external parties.

With the help of code examples, we have come to the conclusion that, according to SOLID principles, a good class is a class that does not allow code changes but does allow extension for functionalities. There can be more variations of the class, but there should not be a way for anyone to modify the code of an existing class.

Chapter 6 Liskov Substitution Principle

Let's look at Wikipedia's definition of the Liskov Substitution Principle:

"If S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e. objects of type S may substitute objects of type T) without altering any of the desirable properties of that program (correctness, task performed, etc.)."

I interpret this definition to mean that the parent should be easily replaced by the child object.

To understand the definition a bit more, let's look into another example using email operations such as the one in our OCP chapter. First, we have a class named **`EmailNotifications`**. If we also need to send emails for printing, what can we do?

We can create a new class. Let's call it **`NotificationsForPrint`**. It will inherit our class **`EmailNotifications`**. Both classes, the base and child, have at least one similar method.

Can we use our child class to substitute our base class? No, in this situation, we can never do that—which means we need to use inheritance. We'll define two separate interfaces, one for building the message and another for sending the message, then we'll decide on the implementation of where and for what we need to build and send messages.

Let me revisit our discussion of the Open-Closed Principle. OCP provides us with a way to create code that is maintainable and reusable. In other words, OCP is a guideline we can follow in order to extend code by changing old code (or working code).

We know that OCP is abstraction and polymorphism, and that raises questions. In fact, during a presentation I gave at a conference in Chandigarh, India, I was asked two questions that will lead us into an analysis of the Liskov Substitution Principle:

- What is the best way to achieve inheritance here?
- What are scenarios in which we violate OCP?

We'll answer these questions by taking a look at what Barbara Liskov actually wrote about the principle:

"What is wanted here is something like the following substitution property: If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T the behavior of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T."

Note: Barbara Liskov is a computer scientist at the Massachusetts Institute of Technology.

Liskov's definition is simple and straightforward, but I had difficulty correlating it with the real world, and I struggled to map this principle with my project. I am very thankful to Robert C. Martin for defining it in a way that made sense to me:

“Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.”

I can correlate this with scenarios and examples from real-world past projects I’ve worked on.

A couple of years ago, while I was working on a database-synchronization project, we refreshed our development database from the production database (with an algorithm so that actual data requiring secrecy or security wouldn’t be copied into actual form). Validation was the backbone of this project.

Actual validations can become much more complex than the examples we looked at earlier, so let me draft out a few code snippets from the validations module of this project.

We have our main validation interface in Code Listing 19.

Code Listing 19

```
namespace LSP_Violation
{
    public interface IValidator
    {
        void Load();
        bool IsValid();
    }
}
```

With the use of the **`Load()`** method, we simply load all validations or validation settings on the fly-in system, and, with the use of the **`IsValid()`** method, we perform the validation.

Here are few of the various important validations that implement the **`IValidator`** interface:

- **`TypeValidator`**. This validates the required type for the local database from the production/external database, as shown in Code Listing 20

Code Listing 20

```
using System;

namespace LSP_Violation
{
    public class TypeValidator : IValidator
    {
        public bool IsValid()
        {
            throw new NotImplementedException();
        }

        public void Load()
        {
```

```
            throw new NotImplementedException();
        }
    }
}
```

- **`IPValidator`**. This includes a few restrictions, such as not being able to sync data if someone trying to sync from that IP is not on our whitelist or if the IP is not on the list of valid external database IP. Code Listing 21 depicts such a scenario.

Note: A whitelist is a list of entities (such as IP, external database names, email ID, etc.). Refer to: https://en.wikipedia.org/wiki/Whitelist.

Note: There are several external database servers from which our systems sync and import the data. In Code Listing 21, we are using IP to get connected with external databases. Class IPValidator is responsible for making sure our system is syncing and importing the data from an authenticated database server by using a valid IP.

Code Listing 21

```
using System;

namespace LSP_Violation
{
    public class IPValidator : IValidator
    {
        public bool IsValid()
        {
            throw new NotImplementedException();
        }

        public void Load()
        {
            throw new NotImplementedException();
        }
    }
}
```

- **`DateValidator`**. There are several date validations on the basis of region, zone, and area. This validator simply validates quick-date type for the relevant base and performs the validation check, as we see in Code Listing 22.

Code Listing 22

```
using System;

namespace LSP_Violation
{
    public class DateValidator : IValidator
    {
        public bool IsValid()
        {
            throw new NotImplementedException();
        }

        public void Load()
        {
            throw new NotImplementedException();
        }
    }
}
```

This module works as a service that allow us to reduce the overburdening of our application. Code Listing 22 depicts a few code snippets that show the usage of these validators in a real project.

Next, let's make a client and take a look how our code works, although, as you'll see in Code Listing 23, it actually violates LSP.

Code Listing 23

```
using System;
using System.Collections.Generic;

namespace LSP_Violation
{
    class Program
    {
        static void Main(string[] args)
        {
            var validators = LoadAllValidationRules();

            Console.WriteLine("RuleValidations are {0}",
IsValidationRulePassed(validators) ? "passing" : "failing");
        }

        private static IEnumerable<IValidator> LoadAllValidationRules()
        {
            var validators = new List<IValidator> {
                                                    new TypeValidator(),
                                                    new IPValidator(),
```

```
                                                    new DateValidator(),
                                                    new
DynamicValidator()
                                                };
            validators.ForEach(v => v.Load());
            return validators;
        }

        private static bool
IsValidationRulePassed(IEnumerable<IValidator> validators)
        {
            bool isValid = false;
            foreach (var v in validators)
            {
                if (v is DynamicValidator)
                    continue;

                isValid = v.IsValid();

                if (!isValid)
                    return false;

            }
            return false; ;
        }
    }
}
```

Note: The previous code is not a complete code.

Can you revisit Code Listing 23 and identify which SOLID principle it's violating?

It looks good. The code seems pretty clean, it's well abstracted, and it's neat and readable by humans. However, it's breaking our first SOLID principle, SRP (refer to chapter 5).

There are other types of validators that do not require either a **`Load()`** method or an **`IsValid()`** method. For example, **`DynamicValidator`** doesn't require an **`IsValid()`** method. I refer to it as a magic validator, as it has performed a few magical acts. For instance, it didn't perform a validation check by itself, but instead used other validators, which means it didn't require an **`IsValid()`** method. Here it is at work in Code Listing 24.

Code Listing 24

```
using System;

namespace LSP_Violation
{
    public class DynamicValidator : IValidator
    {
        public bool IsValid()
        {
            throw new NotImplementedException();
        }

        public void Load()
        {
            throw new NotImplementedException();
        }
    }
}
```

Note: In this case, can you identify other ways to bypass the IsValid() method check without making a conditional decision?

In order to include this new validator, we have to revisit and rewrite our program—something similar to Code Listing 25.

Code Listing 25

```
private static IEnumerable<IValidator> LoadAllValidationRules()
        {
            var validators = new List<IValidator>
            {
                new TypeValidator(),
                new IPValidator(),
                new DateValidator(),
                new DynamicValidator()
            };
            validators.ForEach(v => v.Load());
            return validators;
        }
```

And finally, our main triggering method would look like Code Listing 26.

Code Listing 26

```
private static bool IsValidationRulePassed(IEnumerable<IValidator>
validators)
        {
            bool isValid = false;
            foreach (var v in validators)
            {
                if (v is DynamicValidator)
                    continue;

                isValid = v.IsValid();

                if (!isValid)
                    return false;

            }
            return false;
        }
```

In this code, we are skipping the **`IsValid()`** check for **`DynamicValidator`**. We found a way by simply skipping that particular validator so that we can avoid certain issues. However, what if we have more than 10 validators with the same kind of behavior? Do we need to perform the same check?

In that case, far too many conditions will need to be defined.

Let's think about fixes for this problem. In one instance, someone might suggest making some logic in the **`IsValid()`** for **`DynamicValidator`** class. That will work, but it's not a good practice.

Tip: Avoid giving a method or code a name that doesn't match what it does.

Let's think of something that does not violate any principles and will produce expected results.

Making fancy stuff by type-checking isn't important or required in order to perform such operations. If you're doing that, you are surely somehow violating LSP.

A proper solution for this issue is to follow the Interface Segregation Principle (we will discuss ISP in detail in the coming chapter). In order to write a proper solution, we can think about our two methods, **`Load()`** and **`IsValid()`**.

`Load()` merely loads all validation settings and rules.

`IsValid()` actually performs validation checks.

In the above path, we have to make changes as follows.

Code Listing 27

```
namespace LSP_Follow
{
    public interface IValidator
    {
       bool IsValid();
    }
}
```

So, we must split it into two interfaces, **`IValidatorCheck`** and **`IValidatorLoader`**, as we see in Code Listing 28**.**

Code Listing 28

```
public interface IValidatorLoader
{
    void Load();
}
```

Next, we can implement the required interface for our **`Validator`** class in Code Listing 29.

Code Listing 29

```
public class DynamicValidator : IValidatorLoader
{
    public void Load()
    {
        throw new NotImplementedException();
    }
}
```

Nice, we did it. Let's rewrite our main code in Code Listing 30.

Code Listing 30

```
  private static IEnumerable<IValidatorLoader> LoadAllValidationRules()
        {
            var validators = new List<IValidatorLoader>
            {
                new TypeValidator(),
                new IPValidator(),
                new DateValidator(),
                new DynamicValidator()
            };
            validators.ForEach(v => v.Load());
            return validators;
        }
```

Code Listing 30 names only types of **`IValidatorLoader`** (or types required only to load settings or validation rules).

Finally, with Code Listing 31 we need to perform validation checks for only those validators that actually require it.

Code Listing 31

```
private static bool IsValidationRulePassed(IEnumerable<IValidatorCheck>
validators)
        {
            bool isValid = false;
            foreach (var v in validators)
            {
                isValid = v.IsValid();

                if (!isValid)
                    return false;

            }
            return false; ;
        }
```

The preceding implementation actually shows ISP, which, according to Wikipedia, means:

"No client should be forced to depend on methods that it does not use.” The definition continues, stating that large interfaces should be broken into small and more specific interfaces.

That will be the topic of our next chapter. But first, let’s revisit the main point of this chapter:

The Liskov Substitution Principle tells us how to handle real-time problems.

Chapter 7 Interface Segregation Principle

Let's again look at a definition from Wikipedia:

"No client should be forced to depend on methods that it does not use. ISP splits interfaces that are very large into smaller and more specific ones."

I take this to mean: "As a client, why should I implement nine methods of an interface when I need only three methods?" Fewer methods make a developer's life easier.

You might notice that this is similar to the High Cohesion Principle of General Responsibility Assignment Software Patterns (GRASP). Although GRASP goes beyond the scope of this e-book, we should be aware that these patterns provide a guide for assigning responsibility to collaborating objects.

Here is a list of the GRASP patterns:

- Creator
- Information Expert
- Low Coupling
- Controller
- High Cohesion
- Indirection
- Polymorphism
- Protected Variables
- Pure Fabrication

 ***Note: 'Cohesion' refers to how the operations of any element are functionally related.***

 ***Note: Refer to: https://en.wikipedia.org/wiki/Cohesion_(computer_science) to learn more about Cohesion.***

In previous chapters, we've seen how to solve our validators issue. For the proper solution, we adopt ISP. Can we say that if someone is violating LSP, they gain an understanding of ISP?

In this chapter we will discuss a few more scenarios in which we can adapt ISP to solve our real-world problems.

Here's what Robert C. Martin said about ISP:

"Make fine-grained interfaces that are client specific."

From this, we can say that we should not mess classes in a pollution of interfaces. Our implementing class should obey the interface that possesses the required functionality.

Let me rephrase—ISP says we should not force clients to use interfaces they don't need to use.

Let's go back and revisit the problem in which one of our validators didn't want to use a specific method, but we forced it to use our **`IsValid()`** method.

Code Listing 32

```
public interface IValidator
{
  void Load();
  bool Isvalid();
}
```

Here in Code Listing 32 we are providing the interface **`IValidator`** for the client. Note that if any class implements this interface, it must implement all the methods.

If one class doesn't need either the **`Load()`** or the **`IsValid()`** method, this interface is forcing clients to implement unwanted methods, as Code Listing 33 demonstrates.

Code Listing 33

```
public class DynamicValidator : IValidator
{
    public bool IsValid()
    {
       //I do not want this, why should I care about this?
       throw new NotImplementedException();
    }

    public void Load()
    {
        //Some stuff
    }
}
```

`DynamicValidator` is not meant to implement the **`IsValid()`** method. If a client implements **`IValidator`**, that client is forced to deal with the **`IsValid()`** method, which is not good. If you are doing this, your client won't be happy.

In other words, we can say that our interface **`IValidator`** is fat and not cohesive. It might have only two methods, but most clients will feel uneasy interacting with these methods.

Note: An interface is called a fat interface or a bloated interface when it incorporates too many operations. Refer to: https://en.wikipedia.org/wiki/Interface_bloat for more details.

Many developers get confused while they're reading about LSP and ISP. Here is a concise statement of the difference between them:

“The LSP is about subtyping and inheritance. The ISP is about business logic to clients communication.”

With that in mind, we’ll look into the implementation logic for ISP. In Code Listing 34, we have an abstract class validator.

Code Listing 34

```
public abstract class Validator
{
    public abstract void Load();
    public abstract bool IsValid();
}
```

Some clients will implement this class in order to implement client-specific business logics or what they need as per their own requirements. Code Listing 35 shows an example.

Code Listing 35

```
public class SchemaValidator : Validator
{
    public override bool IsValid()
    {
        throw new NotImplementedException();
    }

    public override void Load()
    {
        throw new NotImplementedException();
    }
}
```

And we have a **`DynamicValidator`** as well, as seen in Code Listing 36.

Code Listing 36

```
public class DynamicValidator : Validator
{
    public override bool IsValid()
    {
        throw new NotImplementedException();
    }

    public override void Load()
    {
        throw new NotImplementedException();
    }
}
```

Note: This is not complete code. Refer to Bitbucket for the complete source code.

In Code Listing 36's scenario, all clients have to bind or must stick with the internal implementation, which means they have to override the methods declared in the abstract class.

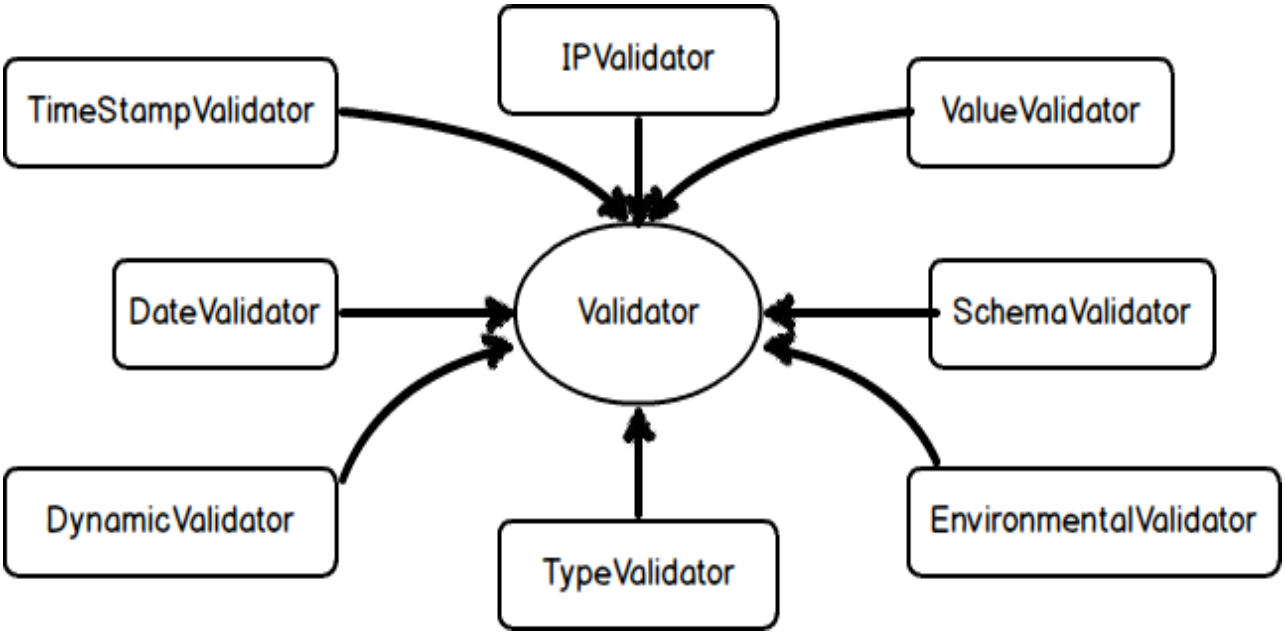


Figure 9: 1-server, n-clients scenario

In Figure 9, we have a scenario in which 1-server is contacted by n-number of clients. We have certain clients that are a special type of client, such as **`DynamicValidator`** and **`EnvironmentalValidator`** (these require either method).

You can also think in terms of vice-versa. Suppose you have an interface, **`IValidator`**, that has only the **`IsValid()`** method, and various clients are using this. Some special type of validators require one additional method along with **`IsValid()`**. This new method would be **`Notify()`**. Here is the challenge—if we add a new method to an existing interface, we are forcing our happy clients to implement new methods as well.

Imagine you are using an API from a third party, you implement everything, then you deploy to production. The next day, when you come into the office, you are suddenly informed that the third party changed its API and added a new method to the interface you are currently using. Whichever programmers changed that API, you hope their cell phone falls in a toilet!

So you can see how painful it would be if you put unnecessary methods in the interfaces and forced a client to use them. Yet in this scenario, the client must use them because there is no other option available.

Imagine a case in which your implementations team is very happy with the external clients who are using your interfaces, but suddenly your manager says that a top revenue-generating client has requested a few changes be made in your existing interface. They want to add two new methods and delete or deprecate one old method (as they are not going to use it anymore). What would you do?

As a typical developer, you would start digging into the code, or you would discuss the situation with your manager and come up with some good solutions. There are a lot of headaches in real-world applications while you are working with highly scalable applications and your clients are pushing you to provide them new features continuously.

Let's come back to our original issue of interface segregation. We have the solution, but there are a few more tweaks I would like to discuss.

What is our requirement?

As clients, we must implement only those things we require. As module writers, we should not force our client to implement anything they might not require.

In our real-time example, we must provide a new method to both our old and new clients, so let's try to make it ISP in Code Listing 37.

Code Listing 37

```
public interface IValidator
{
    bool Isvalid();
}

public interface IvalidatorLoader : IValidator
{
    void Load();
}
```

In order to meet our requirements, we have created a new interface **`IValidatorLoader`**, which implements interface **`IValidator`** itself. Why? Because our new clients need new methods, but they don't say whether or not they still require the existing method, which means they will get both the old and new methods, **`IsValid()`** and **`Load()`**, as seen in Code Listing 38.

Code Listing 38

```
public class Validator : IValidator
{
    public bool Isvalid()
    {
        //Perform some awesome stuff here.
        return true;
    }
}
```

Our old validator class will remain unchanged—it will implement **IValidator** interface and there will be no change in business logic, either.

Code Listing 39

```
public class SpecialValidator : IValidator, IvalidatorLoader
{
    public bool Isvalid()
    {
        //Why should I do new things–I have this method already in place.
        Validator validator = new Validator();
        return validator.Isvalid();
    }
    public void Load()
    {
        //Perform good stuff here to load special validator.
    }
}
```

Code Listing 39 shows our **SpecialValidator** class, which is introduced in order to meet the requirements of our new clients. The new class implements both **IValidator** and **IValidatorLoader**.

There are no changes in business logic for our **IsValid()** method—we are simply wiring up the existing method within definition. There is no need to write and make duplicate code.

Note that we have introduced a new **Load()** method. It should have some real business logic, and here we write all business logics related to our new **Load()** method.

We are done! Let's see how our clients will be making a call as depicted in Code Listing 40.

Code Listing 40

```
class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Old clients, who do not require a new
method");

            IValidator validator = new Validator();
            var isvalid = validator.Isvalid();

            NotifyValidationStuff(isvalid);

            IvalidatorLoader validatorLoader = new SpecialValidator();
            NotifyValidationStuff(validatorLoader.Isvalid());
```

```
            //This is for example only, in real time one should have some
business logic on when to load validators.
            validatorLoader.Load();
            Console.WriteLine("New client can get the taste of Load()
method as well");

            Console.ReadLine();

        }

        private static void NotifyValidationStuff(bool isvalid)
        {
            Console.WriteLine("Validations are {0}.", isvalid ? "passing"
: "failing");
        }
    }
```

This is what we wanted. Code Listing 40's code is self-explanatory. We keep our old clients happy by not complicating their lives, and we provide good things for our new clients, giving them access to both the new and old methods.

Chapter 8 Dependency Inversion Principle

The Dependency Inversion Principle is related to decoupling. Here is Robert C. Martin on DIP:

“High-level modules should not depend on low-level modules. Both should depend on abstractions.”

In thinking about this, let's have a look at our **`EmailNotification`** example once again and consider why our code should decide where to send email (SMTP server or a printer) at the very beginning. Why should it not automatically perform the preferred action?

We’ll examine those details later in this chapter, but first let me address the confusion many developers have about DIP and Inversion of Control. I’ve received many emails and messages on my social pages asking about their differences. Simply put, the Dependency Inversion Principle is a software design principle and Inversion of Control is a software design pattern.

Again, let’s think about the original validator real-time scenario we’ve discussed in previous chapters. We are now adding a new element, writing the validation results by choosing **`ValidatorWriters`**.

Note: Different validators have different writers.

Code Listing 41

```
public class DataValidator : IDataValidator
    {
        private IValidatorWriter writer;

        public void Validator(ValidatorWriter validatorWriter)
        {
            //Write stuff to validate data.
            string validationResults = ValidateData();

            //Write validation results.
            WriteValidationResults(validatorWriter, validationResults);
        }

        private void WriteValidationResults(ValidatorWriter
validatorWriter, string validationResults)
        {
            if (validatorWriter == ValidatorWriter.FileValidatorWriter)
                writer = new FileValidatorWriter();
            else if (validatorWriter ==
ValidatorWriter.TypeValidatorWriter)
                writer = new TypeValidatorWriter();
```

```
            else if (validatorWriter ==
ValidatorWriter.DataValidatorWriter)
                writer = new DataValidatorWriter();
            else if (validatorWriter ==
ValidatorWriter.IPValidatorWriter)
                writer = new IPValidatorWriter();
            else if (validatorWriter ==
ValidatorWriter.SchemaValidatorWriter)
                writer = new SchemaValidatorWriter();
            else if (validatorWriter ==
ValidatorWriter.CodeValidatorWriter)
                writer = new CodeValidatorWriter();
            else
                throw new ArgumentException("No matched validator
found.", validatorWriter.ToString());
            writer.Write(validationResults);
        }

        private string ValidateData()
        {
            throw new NotImplementedException();
        }
    }
```

So, we have a **DataValidator** class that implements an interface **IDataValidator** and performs the specific operations.

Code Listing 42

```
public interface IDataValidator
    {
        void Validator(ValidatorWriter validatorWriter);
    }

public interface IValidatorWriter
    {
        void Write(string strText);
    }
```

The **IValidatorWriter** interface actually provides the method to write all validation results.

Code Listing 43

```
public class CodeValidatorWriter : IValidatorWriter
{
    public void Write(string strText)
    {
       throw new NotImplementedException();
```

```
    }
}

public class SchemaValidatorWriter : IValidatorWriter
{
    public void Write(string strText)
    {
       throw new NotImplementedException();
    }
}

public class TypeValidatorWriter : IValidatorWriter
{
   public void Write(string strText)
   {
      throw new NotImplementedException();
   }
}
```

Here is our implementation of the **`IValidatorWriter`** interface. Be sure to note **`SchemaValidatorWriter`** and **`TypeValidatorWriter`**—both have different ways of writing validation rules, which means there are different implementations of the **`Write()`** method for each one.

But what principle is violated here?

There is no doubt that our code is violating SRP. Here we are actually deciding which object should be created as per the enum **`ValidatorWriter`**.

In Code Listing 44, we should create an object of class **`FileValidatorWriter`** but not **`TypeValidatorWriter`**.

Code Listing 44

```
public enum ValidatorWriter
{
   TypeValidatorWriter = 1,
   DataValidatorWriter = 2,
   IPValidatorWriter = 3,
   SchemaValidatorWriter = 4,
   CodeValidatorWriter = 5,
   FileValidatorWriter = 6
}
```

While we are making a decision here, we are in fact ignoring the actual **`DataValidator`** class concept, which is meant to validate the data and not to write validation results. But we are making the decision on the basis of the incoming type of **`ValidatorWriter`**, then performing **`Write()`** after the creation of an object of specific class, which is wrong and violates SRP.

So, what would be the best solution?

Let's review. Note that the **DataValidator** class is currently responsible for deciding which writer should be created to write for specific validation results. So we found the solution—let's withdraw these rights from the **DataValidator** class and introduce something new that will decide which class object should be created to write for specific validation results.

And, because we are delegating the responsibility to a new class, we cannot force our existing class to play more operations.

We can envision an implementation in which the client makes the call and tells us what kind of writer is required for this validator. In Code Listing 45, we rewrite our **DataValidator** class.

Code Listing 45

```
using System;

namespace DIP_Follow
{
    public class DataValidator : IDataValidator
    {
        private readonly IValidatorWriter _writer;

        public DataValidator(IValidatorWriter writer)
        {
            _writer = writer;
        }

        public void Validator(ValidatorWriter validatorWriter)
        {
            //Write stuff to validate data.
            string validationResults = ValidateData();

            //Write validationResults.
            _writer.Write(validationResults);
        }

        private string ValidateData()
        {
            throw new NotImplementedException();
        }
    }
}
```

In Code Listing 45, we wrote clean code—it's concise, and we removed all those dirty **if..else if..else** blocks., Our **DataValidator** class knew which kind of writer we should use to write validation results. Our client should ship this with their call, something like this: **var obj = new DataValidator(new SpecificWriter());**.

Code Listing 46 is our complete call from the client.

Code Listing 46

```
using System;

namespace DIP_Follow
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Start writing validation results.");

            IDataValidator dataValidator = new DataValidator(new
SchemaValidatorWriter());

            Console.ReadLine();
        }
    }
}
```

In the preceding code examples, in places where our code violated DIP, we were explicitly checking the type of writer, but here in Code Listing 46 we do not need to handle that scenario in our code.

Chapter 9 Conclusion

Here in Chapter 10, I will conclude our look at SOLID, but if you want to discuss these topics further, feel free to contact me. Otherwise, make a pull request to the repository, or send me your solutions and I will push them to the *SOLID Succinctly* repository.

Let's recap what we have examined so far.

The importance of SOLID

We started with a look at the importance of SOLID principles in our code and programs.

We discussed:

- **Why SOLID?**—We concluded that SOLID principles provide us with a way to write neat, clean code base that is readable by humans.
- **Checklist**—We concluded with a checklist of points to help determine whether or not our code is dirty code:
    - Is the code implementing design patterns?
    - Is the code tightly coupled?
    - Is the code testable?
    - Is the code human readable?
    - Is the code duplicated?
    - Is the code too lengthy to understand?
- **Should I care about SOLID?**—We concluded that if we want to make our code readable, robust, and clean, we must care about SOLID.
- **Starting to learn SOLID**—We looked at things that commonly puzzle people when they start to learn SOLID principles.
- **Single Responsibility Principle**—We can define Single Responsibility Principle with: "A class should be designed for a single responsibility. The responsibility of this class should be completely encapsulated by the class."
- **Open-Closed Principle**—We can use Wikipedia's definition for OCP: "Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."

- **Liskov Substitution Principle**—We concluded that this principle can be summarized as "a parent should be easily replaced by the child object without affecting the correctness of the program."

- **Interface Segregation Principle**—We can describe this with: "No client should be forced to implement methods it does not use, and large interfaces should be broken into small and more specific interfaces." We also dipped into the High Cohesion Principle of GRASP, which refers to how operations of elements are functionally related.

- **Dependency Inversion Principle**—We noted this is related to decoupling and that DIP states: "High-level modules should not depend on low-level modules. Both should depend on abstractions."

Please refer to the Bitbucket source code repository for exercises.

Likely questions about SOLID

1. Can we implement SOLID principles in an existing project, and if yes, how so?

Yes, we can implement SOLID principles in our existing projects. Keep in mind that SOLID principles are not tools; these are only guidelines that tell us how to write SOLID code.

Here are few things I find useful for cleaning or refactoring an existing code. Thanks to Stack Overflow and Aaron Daniels for http://stackoverflow.com/questions/783974/how-to-implement-solid-principles-into-an-existing-project.

Single Responsibility Principle

As we now know, a single class should have a single responsibility. So, while you're implementing SOLID in your existing project, please take a deeper look into your classes to see if they obey or can be modified to follow SRP. To do so, I generally follow these steps:

1. Check what your class is written for.
2. See if your class is doing multiple things at once.
3. Think about whether or not there are any scenarios or reasons that could force your class to change.
4. Start refactoring.
5. Make small chunks of your class functionalities. You can break one class into many classes in order to serve SRP.

Note: The above points are not rules of thumb or principles for refactoring, these come from my own practices—you can compile steps that work for you.

Open-Closed Principle

We know that our classes should not allow changes but instead should be open for extension.

In order to obey OCP, check your existing code and try to keep your members and methods virtual or abstract, if feasible. If you want to extend any functionality at some point, make sure you can do so easily later on.

Liskov Substitution Principle

In my view, in order to implement this principle, you should check and implement inheritance in the way that best suits your existing code.

Interface Segregation Principle

To reiterate the definition:

“No client should be forced to implement methods it does not use, and large interfaces should be broken into small and more specific interfaces.”

Get into a deep check of your existing code and identify the interfaces in which you are providing too many responsibilities to classes (that are implementing these interfaces). Now, try to refactor these interfaces by diving into various interfaces. I suggest going to the code examples discussed in our chapter on the Interface Segregation Principle.

Dependency Inversion Principle

In order to implement this pattern, think about a scenario in which we are trying to use the functionality but without giving full details. I suggest you go through our discussion on DIP.

2. I am aware that Repository Pattern violates SOLID principles, but which principles does it violate and how so?

Yes, Repository Pattern violates SOLID principles. In order to discover which principles and how so, we need to revisit the Repository Pattern (it’s beyond the scope of this e-book, so I am using a few simple code examples).

Consider the following repository in Code Listing 47 (I am not considering generic repository).

Code Listing 47

```
public interface IServerDataRepository
{
    void Add(ServerData data);
    bool Save(ServerData data);
    bool Update(ServerData data);
    ServerData GetBy(int id);
    ServerData FindSpecialRecById(int splid);
    bool DeleteById(int id);
    IEnumerable<ServerData> GetServerData();
}
```

Here, we have interface **`IServerDataRepository`**, and as is evident by its name, it’s a repository of **`ServerData`** and contains the following methods:

- **`Add`**–Inserts new **`ServerData`** records.
- **`Save`**–Persists **`ServerData`** records and returns true/false.

- **Update**–Updates the existing **ServerData** record and returns true/false.
- **GetBy**–Fetches **ServerData** record by **ServerData** ID.
- **FindSpecialRecById**–Fetches **ServerData** record by **ServerData** special ID (as can be predicted from its name, this seems to be a special method).
- **DeleteById**–Removes **ServerData** record by ID and returns true/false. (It isn't clear from the method name if this method deletes records physically or just marks records as deleted).
- **GetServerData**–Returns a collection of type **ServerData**. (It looks like a heavy operation because it fetches all records. If you have 30 million records in a database, is this a good method?)

In Code Listing 48, we have two classes, **ServerData** and **SpecialServerData**, that implement the interface **IServerDataRepository**.

Code Listing 48

```
using System.Collections.Generic;

public class ServerData : IServerDataRepository
{
    public void Add(ServerData data)
    {
        //Code implementation
    }
    public bool Save(ServerData data)
    {
        //Code implementation
    }
    public bool Update(ServerData data)
    {
        //Code implementation
    }
    public ServerData GetBy(int id)
    {
        //Code implementation
    }
    public ServerData FindSpecialRecById(int splid)
    {
        //Code implementation
    }
    public bool DeleteById(int id)
    {
        //Code implementation
    }
    public IEnumerable<ServerData> GetServerData()
```

```
    {
        //Code implementation
    }
}
```

Look closely at Code Listing 48 and find which SOLID principle the code is breaking. Yes, it's breaking SRP. Our class is doing many things—it's meant to update records, delete records, fetch records, and more—but it should have only a single responsibility. Imagine the implementation of each method. It's dirty code in which we are making a mess of our code in a single place. It reminds me of my student days of coding, but now it's time to make our code mature. That's how we need to think as we address this class—simply that it is violating the Single Responsibility Principle.

Code Listing 49

```
using System.Collections.Generic;

public class SpecialServerData : IServerDataRepository
{
    //We are supposed to implement only this method.
    public ServerData FindSpecialRecById(int splid)
    {
        //Code implementation
    }

    //Unfortunately we have to implement them all.
    public void Add(ServerData data)
    {
        //Code implementation
    }
    public bool Save(ServerData data)
    {
        //Code implementation
    }
    public bool Update(ServerData data)
    {
        //Code implementation
    }
    public ServerData GetBy(int id)
    {
        //Code implementation
    }

    public bool DeleteById(int id)
    {
        //Code implementation
    }
    public IEnumerable<ServerData> GetServerData()
```

```
    {
        //Code implementation
    }
}
```

In the Code Listing 48 implementation, we require only one method implementation, but we have to implement all the methods, so we are violating ISP.

As an exercise, look at Code Listing 49 and try to find more points where a SOLID principle is being violated.

3. As a solution architect, what priorities do you keep in mind while implementing SOLID principles in your new solution?

There are many ways to answer this important question. From my vantage point as a solution architect, the first thing I must provide is the solution of a real-time design or a problem, etc. The solution can depend on or be independent from language. For example, a solution should work similarly for Java and C# code.

Here are a few recommended points I keep in mind while writing new programs:

- Write a class which deals only with one operation at a time—if my class is meant to show data, it should not do save data, etc. (SRP).
- Write classes in a manner so that there will be no reason to later change the functionality of the class (OCP).
- Abstract code so that the client is able to use a derived object instead of an instance of a class defining the parent type (LSP).
- Create smaller interfaces with minimum methods during segregation (ISP).
- Keep loose coupling in mind while writing software/an application,(DIP).

4. Are SOLID principles good for writing code that is easily testable?

Yes. SOLID principles are indeed good for writing a code that can be easily tested. Consider the five principles and try to map how easy they would be to test:

Single Responsibility Principle

In a case in which a class is responsible for saving data, the implementation of the **Save** method should be abstracted somewhere in a business class to implement business logics.

Therefore, using SRP, we can easily test our class and business logics by writing only two test cases—one to test **Save** method and one to test business logics.

Open-Closed Principle

Refer to Code Listing 15, in which **`ValidateData`** class implements the **`IValidator`** interface and validates the **`ServerData`** with **`SourceData`**.

Using OCP, we can write less code to test, and we created mock data and test validations in our Code Listing 15 example.

Liskov Substitution Principle

In Code Listing 27, we used the **`IValidatorLoader`** interface, and we can use the **`IsValid()`** method while using any validators such as **`TypeValidator`**, **`IPValidator`**, etc., without knowing which validator class we are using.

By following LSP, we can write tests for mocked objects, test fake data, and verify it.

Interface Segregation Principle

With the use of ISP, in which we have smaller methods, we can more easily use tests to mock our methods.

Dependency Inversion Principle

Finally, by following DIP, we are writing good code, which makes it easily testable code. As with using ISP, we can easily mock an interface when we use DIP.

Test your understanding

1. Which SOLID Principle is violated in Code Listing 50?

Code Listing 50

```
using System;

public abstract class Notify
{
    public abstract void NotifyClient();
}

class OnPremisesClient : Notify
{
    public override void NotifyClient()
    {
        Console.WriteLine("You're getting these notifications because you
opted....");
    }
}

class CloudClient : Notify
```

```
{
    public override void NotifyClient()
    {
        Console.WriteLine("You're getting these notifications because you
opted....");

        if (IsOnPremisesToo)
            NotifyClientAsOnPremisesClient();
    }

    public void NotifyClientAsOnPremisesClient()
    {
        Console.WriteLine("Awesome! You are also using On premises
services...");
    }

    public bool IsOnPremisesToo { get; set; }
}
```

The code in Code Listing 51 implements the preceding classes and code.

Code Listing 51

```
class Program
{
      static void Main(string[] args)
      {
          var premisesClient = new OnPremisesClient();
          var cloudClient = new CloudClient();

          ProcessNotifications(new List<Notify> { premisesClient,
cloudClient });
      }

      private static void ProcessNotifications(List<Notify> list)
      {
           throw new NotImplementedException();
      }

      static void HandleItems(IEnumerable<Notify> notifications)
      {
          foreach (var notification in notifications)
          {
             if (notification is CloudClient)
             {
                 var cloudClient = notification as CloudClient;
                 cloudClient.IsOnPremisesToo = true;
             }
```

```
            notification.NotifyClient();
        }
    }
}
```

The code snippets in both Code Listings 50 and 51 are violating SOLID principles. Let's discuss them one by one:

Single Responsibility Principle

The **`Notify`** class uses **`Console.WriteLine`** and, as per class, the main purpose of this notification is not clear.

Class **`Notification`** should have one and only one method. The corrected code that follows SRP would look like Code Listing 52.

Code Listing 52

```
public void SendAll()
{
   foreach (var notificationProvider in _providerList)
   {
      notificationProvider.Process();
   }
}
```

Liskov Substitute Principle

We should use some kind of base class, making sure to use Is-substitute for the relationship instead of Is-A (which is currently being used in the code).

The corrected code that follows LSP would look like Code Listing 53.

Code Listing 53

```
public class OnPremiseProvider : INotify
    {
        public void Process()
        {
```

```
            //This should be used more than just a message or
notification.
            //Assuming there are few business rules for this particular
provider
            //process those rules and change code accordingly.
            Console.WriteLine("You're getting these notifications because
you opted for OnPremise Notifications....");
        }
    }
```

We have interface INotify and now, whenever we make a call from client, we use INotify instead of OnPremisesProvider, which will look something like Code Listing 54.

Code Listing 54

```
INotify onPremiseProvider = new OnPremiseProvider();
```

Dependency Inversion Principle

Code Listing 54 clearly violating DIP, as it's dependent on **`Notify`** class. Instead of client, we should make changes to pass messages from the notifier itself.

With the implementation of **`SendAll`** method, our code is no longer dependent on client implementations.

With the new code changes, I have also refactored the code. Code Listing 55 shows the code when obeying SOLID Principles.

Code Listing 55

```
public class Notification
    {
        private readonly IEnumerable<IProcess> _providerList;

        public Notification(IEnumerable<IProcess> providerList)
        {
            _providerList = providerList;
        }
```

```
        public void SendAll()
        {
            foreach (var notificationProvider in _providerList)
            {
                notificationProvider.Process();
            }
        }
    }

    public interface IProcess
    {
        void Process();
    }

    public class OnPremiseProvider : IProcess
    {
        public void Process()
        {
            //This should be used more than just a message or
notification.
            //Assuming there are few business rules for this particular
provider
            //process those rules and change code accordingly.
            Console.WriteLine("You're getting these notifications because
you opted for OnPremise Notifications....");
        }
    }

    public class CloudNotifier : IProcess
    {
        public void Process()
```

```
        {
            //This should be used more than just a message or
notification.
            //Assuming there are few business rules for this particular
provider
            //process those rules and change code accordingly.
            Console.WriteLine("You're getting these notifications because
you opted for Cloud Notifications....");
        }
    }

    public class GalaxyNotifier : IProcess
    {
        public void Process()
        {
            //This should be used more than just a message or
notification.
            //Assuming there are few business rules for this particular
provider
            //process those rules and change code accordingly.
            Console.WriteLine("You're getting these notifications because
you opted for Galaxy Notifications....");
        }
    }
```

In the new code, we have renamed **`NotifyClient`** method to **`Process`** because this will process our notifications for client. Every notifier implements the **`IProcess`** interface. Now, at client side there is no need to take care of notifications per provider, and our **`Notification`** class will automatically take care of these notifications. So, our client code would look Code Listing 56.

Code Listing 56

```
class Program
    {
```

```
        static void Main(string[] args)
        {
            var notification = new Notification(new List<IProcess>
            {
                new CloudNotifier(),
                new OnPremiseProvider(),
                new GalaxyNotifier()
            });

            notification.SendAll();

            Console.ReadLine();
        }
    }
```

There are many ways to implement the code, but the main consideration is that we follow SOLID principles.

Note: Complete source code is available on the Bitbucket repository under Chapter 10. You can also see the working code using this fiddle: https://dotnetfiddle.net/vXg3Lg .

2. Implement the SOLID principle you found for Q.3 code.

In order to implement this yourself, first figure out the violated SOLID principle, then rewrite the entire code. I would love to see your new code—you can share by creating a pull request on the source code repository at Bitbucket, or you can write to the email address I've given in the About the Author section.

Test questions

Here is a list of comprehension questions.

1. What are SOLID principles? Explain with examples.

2. Explain the importance of the Single Responsibility Principle, using an example.

3. What SOLID principles are you violating while implementing Repository Pattern?

4. What is meant by the principle of package and namespace coupling?

5. What is meant by the principle of package and namespace cohesion?

6. As a .NET developer, how do you find SOLID principles in asp.net or MVC projects (code/folder structure in Visual Studio)?

7. What is the importance of refactoring while you're working on legacy projects and while you're working on new projects?

8. Create a small utility to grab configurations key/values from a config file (these files might have an extension of .config, .txt, .ini, or .json) and have a key value pair in the format of **`<key="" value="" otherinfo="" />`** by using SOLID principles.

9. Share your experience from your first SOLID refactoring.

10. Give an example for OCP versus LSP.

Note: LSP is an extension of the Open-Closed Principle.

References

I am thankful to all the great people who helped me to prepare this e-book. Here are some of the references used:

- http://stackoverflow.com/questions/26465627/difference-between-oop-basics-vs-solid
- http://www.dofactory.com/net/design-patterns
- http://www.codeproject.com/Articles/1028439/Getting-Started-with-Microsoft-ASP-NET-WebHook-Pre
- http://www.codeproject.com/Articles/703634/SOLID-architecture-principles-using-simple-Csharp
- http://www.codeproject.com/Articles/833246/Learning-The-S-O-L-I-D-Programming-Principles-Over
- http://www.questpond.com/demo.html#dp
- http://www.codeproject.com/Articles/1017352/Solidifying-your-code-using-SOLID-Programming-Prin
- http://programmers.stackexchange.com/questions/155852/programming-solid-principles
- http://en.wikipedia.org/wiki/SOLID_(object-oriented_design)
  http://en.wikipedia.org/wiki/Open/closed_principle
- http://code.tutsplus.com/tutorials/solid-part-3-liskov-substitution-interface-segregation-principles--net-36710
- http://programmers.stackexchange.com/questions/153410/what-are-the-design-principles-that-promote-testable-code-designing-testable-c
- http://en.wikipedia.org/wiki/Liskov_substitution_principle
- http://en.wikipedia.org/wiki/GRASP_(object-oriented_design)
- http://en.wikipedia.org/wiki/Coupling_(computer_programming)
- http://martinfowler.com/articles/dipInTheWild.html
- https://github.com/garora/somestuff/tree/master/LearningSolid
- http://www.oodesign.com/design-principles.html
- https://en.wikipedia.org/wiki/Coupling_(computer_programming)