SOFTWARE ARCHITECTURE: META AND SOLID PRINCIPLES IN C#

Learn how to develop maintainable software systems applying Meta and SOLID Principles

EngineerSpock

Software Architecture: Meta and SOLID Principles in C#

Learn how to develop maintainable software systems applying Meta and SOLID Principles.

EngineerSpock

This book is for sale at http://leanpub.com/solid-principles

This version was published on 2018-05-10

Tweet This Book!

Please help EngineerSpock by spreading the word about this book on Twitter!

The suggested hashtag for this book is #EngineerSpock.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#EngineerSpock

Dedicated to Marina and Anna: the wings of my soul.

Contents

Few Words About Me

I’m thankful enough for that I love what I do.

I began my career as a postgraduate student participating in Microsoft ImagineCup contest.

I’ve been working with .NET platform since 2003. I’ve been professionally architecting and implementing software for nearly 10 years, primarily based on the .NET platform. I’m passionate about building rich and powerful applications using modern technologies. I’m a certified specialist in Windows Applications and Service Communication Applications by Microsoft. I’m one of the coordinators of the MskDotNet User Group in Moscow.

“If it’s work, we try to do less. If it’s art, we try to do more.” - Seth Godin.

What I can say is that software is my art.

A Bit of Advertisment

I'm the author of more than 10 programming video courses. More than 20k students are enrolled into them. This book is heavily based on one of my courses. I decided to write a book because different people prefer different formats. One prefer videos, another prefer books. Follow this link to learn more[1] about my video courses.

[1] http://www.engineerspock.com/courses-overview/#courses

SOLID Principles

You can download the source code following this link.[2]

[2]https://1drv.ms/u/s!AqtQeAOHZEjQscdr7US3jd16K7CokQ

Introduction

Before investigating what the SOLID principles are and how to apply them properly, we need to understand why do we need them at all?

SOLID principles are all about designing software.

But what is design?

How to define the design of software?

Robert C. Martin in his book "Agile Principles, Patterns, and Practices in C#" stated that

> **the design of software is source code itself**.

Martin Fowler wrote in his blog that

> **architecture is the shape which code takes**.

Why is this so?

Engineers produce documents, blueprints which specify how to build a product. The only thing which truly specifies how software works is **the source code**. You may say that source code is the product, but the product is the running program, not the source code itself.

Look at this from the following perspective. What is the input to a factory of circuit boards? Special electronic diagrams. In the case of software development, we have a factory which is called "compiler". We feed it by the source code. A compiler builds software using the source code. And this leads to interesting thoughts regarding costs model of building software and, say, houses.

When we build a house, it's much cheaper to create a good design upfront rather than rebuild an entire house if we don't like the result. It is just not feasible to rebuild a house each time we don't like the result.

The opposite we can say about software development. We can build binaries in a minute, roughly speaking. Big upfront design in case of software development is much more expensive than to draw a sketch, build the software and then tweak it along the way.

Software development process is somewhat unique because conducting a big upfront design we actually can't guarantee that we take into account all the possible requirements. *The software is the fluid substance and requirements tend to change very quickly.* Because of such a fluid, uncertain nature of software requirements we need to constantly keep the design as clean as we can. Saying clean, I mean as maintainable as we can.

To determine whether the source code is clean or not we need to mark out the signs of design rotting. Developers refer to such signs as **design smells**.

We can mark out five major design smells:

- rigidity,
- fragility,
- immobility,
- viscosity and
- needless complexity.

Let's define them one by one.

The software is **rigid** if the cost of making a single change is very high. If something very simple takes many hours to implement, then the software is rigid. To overcome this problem, you might need to redesign the software. The primary source of rigidity is the high coupling between modules. When modules are highly coupled, then it's very hard to introduce any changes.

The software is **fragile** when small changes in one module cause bugs appearance in other, sometimes even unrelated modules. This smell is also often caused by poorly designed relationships between modules. To overcome fragility, you need to isolate dependencies.

The software is **immobile** when it's components can't be reused in other systems. Most of the time we should be able to extract a component without too many efforts and adapt for using in another system. And what a surprise, this smell is most likely caused by a tight coupling between components. To overcome this smell, you need to decouple components well.

The software is **viscose** when adding a single feature evokes dealing with tons of aspects including, say, transport layer security, marshaling and such things. In practice, you also can detect this smell by observing hard to perform check-ins, check-outs, and merges. And most likely, the primary reason for this smell is a tight coupling between components.

The software is **needlessly complex** when developers all the time are trying to forecast the future, anticipating upcoming changes, introducing excessive points of extension. Developers should concentrate on the current requirements, they should construct the supple architecture which can bend to be able to meet new requirements. Adding not needed points of extension yet is a bad practice which leads to a needless complexity.

Did you notice that almost all the smells are related to bad dependencies management?

The major difference between object-oriented and not object-oriented languages is that OO-languages are capable of harnessing the power of dynamic dispatch. By using virtual functions and interfaces, we can invert the dependencies. In OO-languages when we make a call we don't know which object will handle that call. This is because of polymorphism or dynamic dispatch.

So, the key to achieving a good architecture is to manage the dependencies well.

And here we reach the point when we're ready to talk about SOLID principles.

SOLID principles are five principles which can be referred to as dependency management principles. They are all about relationships and operations between objects.

In this volume, we are going to dive deeply into the SOLID principles. The SOLID acronym was firstly introduced by Robert Martin aka Uncle Bob. SOLID implies five principles of software development:

- SRP – Single Responsibility Principle,
- OCP – Open/Closed Principle,
- LSP – Liskov Substitution Principle,
- ISP – Interface Segregation Principle,
- DIP – Dependency Inversion Principle.

All these principles are based on the classic work of Bertrand Meyer. We are going to discuss all the principles one by one looking at examples of poor design and how to fix it applying an appropriate principle. We will also notice the difference between Mayer's and Martin's definitions of some principles what is also important for deep understanding. What I also want to emphasize is that *SOLID principles are not bound to any technology*. You can apply them in any programming language, in Java for example.

Another point is that *SOLID is not a goal by itself.* The thing is that Fully SOLID code is an oxymoron. It's impossible to write software which conforms SOLID in every single line.

It's impossible to measure the "SOLIDness" of the code base. SOLID are the five principles which help us to create better software. They involve a great deal of philosophy, and you might think that this is bad. Well, you can treat it as a flaw, but actually it's not. It's very important to feel this philosophy of designing software to become a better developer. And in this book, you'll see all the tradeoffs developers face applying SOLID principles; you'll feel the philosophy of designing software.

Ok, you've learned the common design smells and that we apply SOLID principles to remove that smells.

In the next lecture, we will dive into SOLID principles starting with the outline of the SRP chapter.

SRP - Single Responsibility Principle

Outline

In this chapter, we're going to learn what Single Responsibility Principle (SRP in short) is about and how to apply this principle to achieve more maintainable code. In this chapter, you will:

- learn what SRP means in its essence;
- learn at what levels SRP can be applied;
- look at a dozen of examples of SRP violation;
- learn how to adhere the SRP.

Problem Statement

Single Responsibility Principles or SRP in short states that:

> *every object should have a single responsibility, and that responsibility should be entirely encapsulated by the class.*

This is the definition taken from Wikipedia. It's very hard to understand what does it mean. For example, the first two questions which come to my mind are "how to define those responsibilities and how to calculate the number of responsibilities of a certain class?". Such definitions are not very helpful from the practical perspective. Robert Martin aka "Uncle Bob" clarified this definition by saying that

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

Sounds much clearer, though it still needs further clarification.

The responsibilities of a class can be treated as *axes of change*. It might be simpler to understand it as axes of changing requirements. For example, if a class implements logging and caching on its own, then it has at least two responsibilities.

We also can view the responsibilities of a class from the perspective of its users. Any API has its users. These *users are the source of changes.* Understanding that, we can calculate how many axes of change a class has. For example, if a class deals with a database, then we have one axe since DB architects are those users who can request changes. If that class deals with salary reports in addition, then we have two axes since accountants are those who can request changes to reports.

> The more responsibilities a class has, the more likely it's going to be changed!

So, applying SRP, we want to separate different concerns. A class should do one thing, and do it well!

By the way, the same principle can be applied at the level of modules. Modules should have only one logical responsibility.

So, SRP can be applied at different levels:

- at the function level;
- object level;
- and at the module level.

Sometimes we find SRP violations at function's level. We refactor out different responsibilities to classes, and then we separate classes by their responsibilities into different modules or assemblies if you wish. We discussed the SRP, but you might wonder how the concrete problems look like in real life caused by SRP violation.

Imagine the problems which are caused by SRP violation.

```
public class PaymentProcessor
{
    public void Charge(decimal amount)
    {
        //initialize bank terminal
        //send a "ChargeCard" request to the terminal
    }

    public string CreateReport()
    {
        //format a report
        return string.Empty;
    }

    public void PrintReport()
    {
        //initialize printer's driver
        //send a printing command
    }

    public void SavePayment()
    {
        //saving to DB
    }
}
```

Here we have a class. Look at it. How many responsibilities does it have? I see at least four dependencies. Let's count them. The very first method called `Charge` is responsible for *interacting with a bank terminal*; it should know how to initialize a device properly and send a corresponding command to it.

The next method called `CreateReport` knows how to *create a report about the payment transaction*; it knows how to format all the data properly.

Reports are usually so complex that actually here can hide two responsibilities. One separate concern could be the process of *creating a report* and the second is the *formatting*.

If the process of creating an object is too complex that very often we treat it as a separate concern and we tend to isolate such responsibility by extracting a factory class which knows all the details about reports creation.

The third method called `PrintReport` knows how to *deal with printer's driver* and how to send commands to it.

And finally, the `SavePayment` method is responsible for *interacting with a database* to save all the data about a payment transaction.

We often have a hidden responsibility in such cases. We can call that responsibility – "**Orchestration**". We often need some high-level object which knows how to integrate all the responsibilities together.

Now imagine that we need to change the payment processing and saving to a database simultaneously. Two different programmers are responsible for these two different parts of the system. Unfortunately, these two different parts reside in the same class. As a result, these two programmers need to change the same class, and in the end, they will need to merge their changes to finish a check-in.

Aside from this problem, such classes which accumulate many responsibilities are hard to understand. Such classes become a big ball of mud. One day, no one will understand what the hell is going on in that class.

⚠ **Classes with too many responsibilities are hard to understand!**

Another problem is that sometimes we need to introduce changes into only one responsibility, let' say we need to change the payment processing in the example I showed you a minute ago.

Because several responsibilities reside in the single object, classes which represent them will also be recompiled and redeployed.

Nowadays, this problem in the majority of cases may seem like not a problem at all. But for example, if we develop in C++ then it can cause some issues in the end. The thing is that such careless attitude to dependencies management leads to a long compilation time. C++ is much harder to compile, so this problem is very important in such cases.

So, when SRP is violated, responsibilities start to collate with each other, what means that they become coupled. What we need to strive to do is to gather all the same responsibilities together and separate from each other those which are different. When we gather the same responsibilities, we try to achieve the so-called **high cohesion**. When we separate different responsibilities, we try to achieve low coupling.

At the level of functions, cohesion means the following:

> a set of functions, an interface, is considered cohesive when each function is closely related to another.

Coupling indicates how dependent modules are on the inner workings of each other. Tightly coupled modules rely extensively on the specific state of each other, sharing variables and many types. Loosely coupled modules are fairly independent: they have a few well-defined APIs and share a limited amount of or no data at all.

Ok, let's look at the problem of SRP violation on another example.

Demonstration of the Problem

Let's consider an example of SRP violation. In this example, we deal with an application which allows users to buy tickets on suburban trains. Pretend, that the system we develop works on the website and on the point of Service terminals.

```
public class PaymentModel
{
    private decimal _cashAccepted;

    public void BuyTicket(TicketDetails ticket, PaymentDetails payment,
                          Action onPayChangeToMobilePhone)
    {
        if (payment.Method == PaymentMethod.CreditCard)
        {
            ChargeCard(ticket, payment);
        }
        else
        {
            AcceptCash(ticket);
            DispenseChange(ticket, onPayChangeToMobilePhone);
        }
    }

    private void ChargeCard(TicketDetails ticket, PaymentDetails payment)
    {
        var gateway = new ProcessingCenterGateway();
        gateway.Charge(ticket.Price, payment);
    }

    private void AcceptCash(TicketDetails ticket)
    {
        var r = new Random();
                int price = (int) ticket.Price
        _cashAccepted = r.Next(price, price + 1000);
    }

    private void DispenseChange(TicketDetails ticket,
                Action onPayChangeToMobilePhone)
    {
        if (_cashAccepted > ticket.Price &&
            !TryToDispense(_cashAccepted - ticket.Price))
            onPayChangeToMobilePhone?.Invoke();
    }

    private bool TryToDispense(decimal changeAmount)
    {
        return false; //or true :)
    }
}

internal class ProcessingCenterGateway
{
    public void Charge(decimal ticketPrice, PaymentDetails paymentDetails)
    {
        //charging process
    }
}
```

Let's also say that we develop an MVVM-based application, and somewhere in the code base, we have the PaymentModel which provides the ability to buy a ticket.

If you look at the API of the PaymentModel, you'll see that it exposes the `BuyTicket` method with three arguments of the following types:

1. TicketDetails which contains details like the station of departure and arrival, the price and so on.
2. PaymentDetails contains the paying method; a user can pay by cash or by a credit card, it also can contain any other relevant information.
3. The action is the parameter which is used in the following case: a user tries to pay by cash in the POS-terminal, he overpays, POS-terminal tries to dispense the change but it doesn't have enough money to dispense all the change, so we need to return the remainder to the mobile phone of a user. The client's code provides the handler of such situation by passing it via that parameter of type Action. This is a real case from my practice.

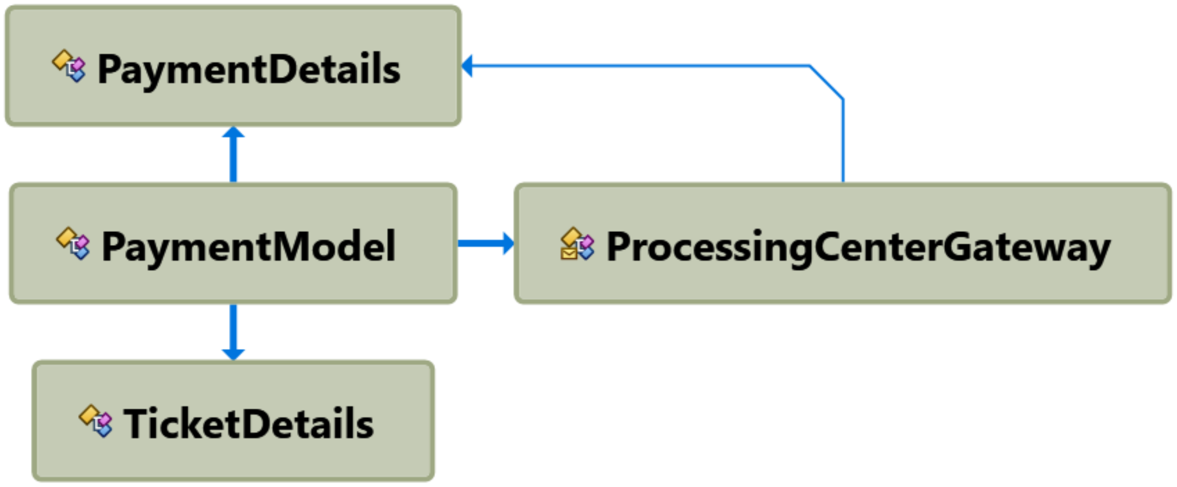
In the body of the method we check if a user requested to pay via a credit card and in such case, we call the `ChargeCard` method passing the ticket and payment parameters.

The `ChargeCard` method creates some gateway through which we perform the Charge operation. If a user wants to pay by cash, we call the `AcceptCash` and then `DispenseChange` methods. In the `AcceptCash` method we would initialize devices which accept money, but here we just have a simple stub for the sake of simplicity. The `DispenseChange` method checks if a user overpaid and in such case, initiates the process of dispensing the change.

I omitted another responsibility – printing the ticket on the paper in case of purchasing a ticket in a POS-terminal. This is for the sake of simplicity, but you can imagine that in real life this method could be even more complex. This example is just fine for our investigation of the problem.

What's wrong with the `BuyTicket` method? It clearly violates the Single Responsibility Principle. The two different mechanisms, paying by a credit card and by cash are coupled.

Let's look at the diagram of dependencies.



The diagram of dependencies

You can see here that the PaymentModel is the meat, it is the central part of the design which is responsible for everything. It depends on PaymentDetails, ProcessingCenterGateway, and TicketDetails. In other words, it knows everything.

We have already discussed the possible unfortunate consequences of such a design. Let's separate the feature of paying by a credit card from the feature of paying by cash.

Refactoring to a Better Design

Let's perform a refactoring. At first, I want to abstract away the process of buying a ticket. So, I'll create an abstract class `PaymentModel` with an abstract method `BuyTicket`.

```
public abstract class PaymentModel
{
    protected TicketDetails _ticketDetails;

    protected PaymentModel(TicketDetails ticketDetails)
    {
        _ticketDetails = ticketDetails;
    }

    public abstract void BuyTicket();
}
```

This class also contains the `TicketDetails` as a protected field since we need `TicketDetails` to be visible from within all the inheritors. Adding `TicketDetails` here is not necessary, but if we want to implement some shared business logic which uses `TicketDetails`, then this could be beneficial.

Again, this is a tradeoff. Everything depends on the certain requirements of a concrete application. Let's introduce an interface for abstracting away the Bank Gateways. I'll call it `ICanPayViaCredit-Card`.

```
public interface ICanPayViaCreditCard
{
    void ChargeCard(TicketDetails ticketDetails,
                        PaymentDetails paymentDetails);
}
```

This interface defines only one method right now, `ChargeCard`.

And again, here is a tradeoff. We can avoid creating a separate interface. But if we know for sure, or expect with a high probability that shortly we will implement other Bank Gateways or we already have different implementations of the BankGateway, then it's meaningful to introduce an interface to decouple different implementation of Bank Gateways from each other.

Let's also create a Bank Gateway which implements the `ICanPayViaCreditCard` interface:

```
public class BankGateway : ICanPayViaCreditCard
{
    public void ChargeCard(TicketDetails ticketDetails,
                PaymentDetails paymentDetails)
    {
        //charge
    }
}
```

The first inheritor of the PaymentModel I'm going to implement is the `OnlinePayment`.

```
public class OnlinePayment : PaymentModel
{
    private readonly PaymentDetails _payment;
    private readonly ICanPayViaCreditCard _bankGateway;

    public OnlinePayment(TicketDetails ticketDetails,
                        PaymentDetails payment) : base(ticketDetails)
    {
        _payment = payment;
        _bankGateway = new BankGateway();
    }

    public override void BuyTicket()
    {
        _bankGateway.ChargeCard(base._ticketDetails, _payment);
    }
}
```

In the constructor, it accepts `TicketDetails` and `PaymentDetails` which are required only in case of payment by a credit card. In the constructor's body, we set the payment property and create the `BankGateway` which implements the `ICanPayViaCreditCard` interface.

Another way is to stick with the all the implementations crammed into a single class which accepts a parameter which would indicate which gateway to use, but that most likely would lead to overcomplicated code.

Developers often refer to such code as to a big ball of mud. Take a note, that we explicitly create a concrete implementation of the `ICanPayViaCreditCard` interface what means that we can't use here other implementations if they exist. This is a topic which we will discuss in the Dependency Inversion Principle chapter. So, wait a little bit, and you'll get the answer.

Another responsibility I want to abstract away with an interface is the responsibility of money accepting and dispensing. So, I'll create another interface for that and I'll call it `ICanOperateWithCash`.

```
public interface ICanOperateWithCash
{
    void AcceptCash();
    void DispenseChange();
}
```

Here we have only two methods for accepting money and dispensing the change. Ok, let's implement another inheritor of the `PaymentModel`. It will be responsible for payments via pos terminals.

```
public class PosTerminalPayment : PaymentModel, ICanOperateWithCash
{
    private readonly Action _onPayChangeToMobilePhone;
    private decimal _cashAccepted;

    public PosTerminalPayment(TicketDetails ticketDetails,
                Action onPayChangeToMobilePhone) : base(ticketDetails)
    {
        _onPayChangeToMobilePhone = onPayChangeToMobilePhone;
    }

    public override void BuyTicket()
    {
        AcceptCash();
        DispenseChange();
    }

    public void AcceptCash()
    {
        Random r = new Random();
                int price = (int) _ticketDetails.Price;
        _cashAccepted = r.Next(price, price + 1000);
    }

    public void DispenseChange()
    {
        if (_cashAccepted > _ticketDetails.Price && !TryToDispense())
        {
            _onPayChangeToMobilePhone?.Invoke();
        }
    }

    private bool TryToDispense()
    {
        //issue a command for dispensing
        //now just a stub
        return false;
    }
}
```

Here we have the `PosTerminalPayment` class. In the constructor, it accepts the `TicketDetails` and a callback to notify the caller about the necessity to return the remainder of change to the mobile phone of a user. This class implements the `ICanOperateWithCash` interface which as I already said exposes two methods, `AcceptCash` and `DispenseChange`.

This implementation is different from what we've done in the `OnlinePayment` class. `OnlinePayment` accepts in the constructor an interface for operating with a bank, while `PosTerminalPayment` itself inherits from the `ICanOperateWithCash` interface.

I just want to show you different ways and tradeoffs. Inheritance via classes and interfaces provides to us the opportunity to build a very flexible design.

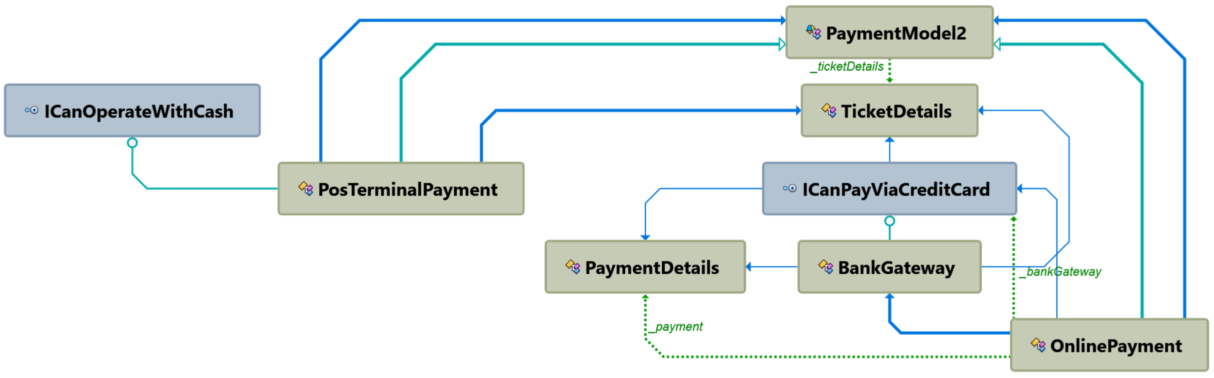
It might be better to accept the `ICanOperateWithCash` in the constructor in this case and frankly I think that it would be better most likely, but I want to stress that we can face a situation when we want to make the `PosTerminalPayment` itself substitutable by another implementation of the `ICanOperateWithCash` interface.

When we decouple a responsibility using an interface and require that interface in a constructor of the class which takes the role of an orchestrator, we rely on a composition rather than on inheritance. Composition doesn't impose such restrictions as a direct inheritance does.

Remember that inheritance couples the parent and child classes very tight.

Child classes take a great burden of satisfying the parent's API and parent's invariants. We will be talking about invariants in the LSP chapter in more details.

Look at the diagram again.



The diagram of dependencies

Now, we have two different classes which orchestrate all the things; these are `PosTerminalPayment` and `OnlinePayment` classes. Each of them takes a concrete responsibility.

`PosTerminalPayment` doesn't know anything about Bank Gateways, `OnlinePayment` doesn't know anything about how to accept and dispense cash. PaymentModel is the top of the hierarchy, and it doesn't depend on anything except `TicketDetails`.

Now we have two points of extension, `ICanOperateWithCash`, and `ICanPayViaCreditCard`.

You can argue that now we have more classes and the design now even more complex than it was before the refactoring. And I can both agree and disagree with you.

Everything depends on the requirements of an application.

If you see that a class or a method is too fat and it is hard to understand, then it would be better to refactor it applying SOLID principles.

Sometimes we can apply only one of the SOLID principles. But more often we apply more than one principle to perform a refactoring.

Ok, you've learned how to apply SRP to perform a refactoring. Now, I want you to look at more simple examples of the SRP violation. You'll see common smells which indicate that the SRP is violated.

More Example of Violation the SRP

Here we have a method named `GetReport`. In the body, we can see that it gathers some statistical data, then creates formatted strings and put them together.

```
class Violation1
{
    //example with business logic and formatting
    public string GetReport()
    {
        int clientsNumber = GetNumberOfClients();
        decimal totalIncome = GetTotalIncome();
        int satisfiedClients = GetSatisfiedClients();
        int unsatisfiedClients = GetUnsatisfiedClients();

        string clientsStr = $"Total number of Clients = {clientsNumber}";
        string incomeStr = $"Total Income = {totalIncome}";
        string satisfiedClientsStr =
                                $"Number of satisfied Clients = {satisfiedClients}";
        string unsatisfiedClientsStr =
                                $"Number of sad Clients = {unsatisfiedClients}";

        return clientsStr + Environment.NewLine +
               incomeStr + Environment.NewLine +
               satisfiedClientsStr + Environment.NewLine +
               unsatisfiedClientsStr + Environment.NewLine;
    }
}
```

Does this method violate SRP?

This method clearly violates the SRP. It has two responsibilities. The first one – *gathering the statistical data* and the second one is *formatting*. Junior programmers very often mix the formatting responsibility with business logic. Of course, sometimes, if we're sure for 99% that formatting will never be changed and it doesn't interfere with the other code, we can leave such code as it is. But more likely it would be better to at least extract formatting into a separate method.

Tip

It's almost always better to extract formatting into a separate method.

Let's look at another case.

```
public class Violation2
{
    public void FindAlarmDevice()
    {
        var driver = new AlarmDriver();
        string port = driver.Find();
        if (!string.IsNullOrWhiteSpace(port))
        {
            SystemState.AlarmCanBeUsed = false;
        }
        SystemState.AlarmCanBeUsed = true;
    }
}
```

Here we have the `FindAlarmDevice`. It scans through serial ports trying to find the device, and if it finds it, it sets a global state by setting the AlarmCanBeUsed property.

Does this method violate the SRP?

This method clearly violates the SRP. It mixes *the policy or business logic* with *mechanics*. Remember that high-level policy should be decoupled from the low-level details. We could refactor it by introducing a special class which is responsible for interacting with global state. That class could receive notifications via events or any other appropriate way.

Tip

Remember that high-level policy should be decoupled from the low-level details.

Let's look at the next case:

```
class Violation3
{
    public void DrawRectangle()
    {
        Rectangle rect = GetRectangle();
        Color color = Colors.Red;
        rect.Fill(color);
    }

    private static Rectangle GetRectangle()
    {
        //return new Ractangle();
    }
}
```

Here we have the method named `DrawRectangle`. It calculates where to draw a rectangle, then picks a color and draws the rectangle filling it with that color.

Does this method violates the SRP?

This case is trickier than previous ones. The previous examples demonstrate very popular smells of violating the SRP. It's harder to determine the responsibilities. The number of actual responsibilities always depend on the users. Are there actors or users who might be interested in managing the color? If there are such users, then it might be better to separate the responsibility of picking a color.

A separate interesting question is whether the creation of an object is itself a responsibility or not? We're not going to answer this question in this chapter. We will keep it for the chapter about DIP.

In the end, I also want to list some common SRP violations:

1. Mixing logic and infrastructure, when business logic is mixed with views (see Model-View-ViewModel, Model-View-Controller, Model-View-Presenter patterns) or a persistence layer.
2. Class or a module serves different layers: calculates income and sends e-mails, parses XML and analyze the results.

Before going further, I just want to share with you one case. Once, I saw a function which was about *five thousand lines long*. It consisted of switch-cases and if-statements. It relied on them massively. That was a real hell and an extreme example of SRP violation. I wanted to share with you that case just for fun; the majority of developers don't believe me when I say that I've seen such a function. No, such functions exist, and developers who write such functions also exist as you may guess.

Related Patterns

Sometimes, we apply the SRP to refactor a part of a system, and we end up with many small classes. Clients of such an API may struggle with understanding the whole part of the system because they need to understand the relationships between all that small classes. It's not that bad as it sounds. As I've already said, there is always a tradeoff between different solutions.

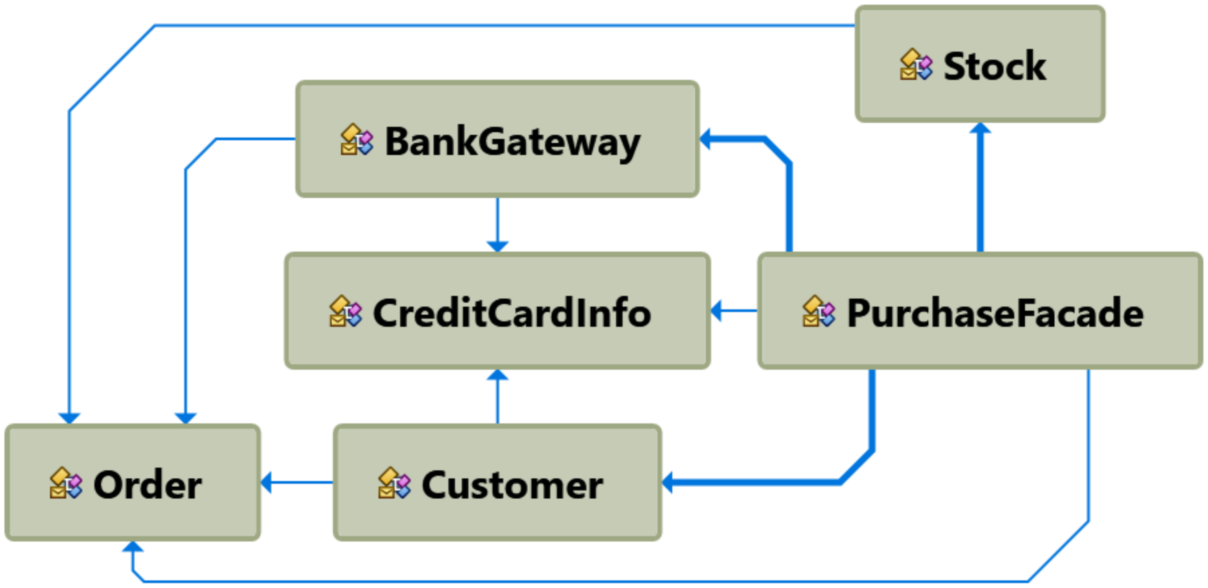
In such cases, the **Façade pattern may come to the rescue**. It wraps all those little classes just delegating all the responsibilities to them. It doesn't make the Façade a violator of the SRP. The only responsibility of such a façade is to bring the functionality required by a client together. It may seem like it violates the SRP since we will see the responsibilities of different layers reflected in the API of the Façade. But as I said, aggregating the functionality which allows solving a single client's problem is not bad. It simplifies the interaction process with the system from the client's perspective, and that's all. Nothing more.

So, there are two main reasons for using the Façade pattern:

- Provide for a client a simple API for interaction with a set of complex objects.
- Provide for a client a cleaner API for interaction with poorly designed API.

Sometimes it's much easier to roll out a Façade rather than rewrite a poorly designed part of a system, and this is what the second point is about.

From the SRP's perspective, we're interested in the example which demonstrates the first reason of using the Façade pattern.
Here you can see a diagram which demonstrates an example of the Façade pattern.



example of the Façade pattern

You can see here that the `PurchaseFacade` class depends on all the other classes. It contains or in another word wraps all these classes. The wrapped entities don't have to be classes; they could be interfaces as well.

So, imagine that initially, a client was dependent on all these classes. We introduce the Façade to simplify the interaction process with the system. If a client depends only on the Façade, it doesn't have to directly interact with many classes.

Let's say we have a client which uses our system to sell goods from the store online. So, the client's code looks like this:

```
class Problem
{
    void Purchase()
    {
        int id = 123;
        Customer c = Customer.FindCustomer(id);

        int goodId = 13457;
        Order o = Stock.Find(goodId);
        CreditCardInfo cci = c.GetCreditCardInfo();
        BankGateway gateway = new BankGateway();
        gateway.ChargeCard(cci, o);

        c.AddToStatistics(o);
    }
}
```

It creates a customer object using the static method `FindCustomer` passing in the `customerId`. After that, it creates an `Order` through the static `Find` method of the `Stock` class passing in the `goodId`. Then it gets the info about the customer's credit card. Creates the `BankGateway` and calls the `ChargeCard` passing required parameters.

I bet such interaction process looks like a mockery. Indeed, it's our duty to simplify lives of our clients. So, we can create a façade which hides all these details of processing a purchase. We can create a facade like the following:

```
class PurchaseFacade
{
    public void ProcessPurchase(int customerId, int goodId)
    {
        Customer c = Customer.FindCustomer(customerId);

        Order o = Stock.Find(goodId);
        CreditCardInfo cci = c.GetCreditCardInfo();
        BankGateway gateway = new BankGateway();
        gateway.ChargeCard(cci, o);


        c.AddToStatistics(o);
    }
}
```

I've created here the PurchaseFacade class which exposes a single method. This method requires two integer parameters and hides all the details inside its body. Now, a client doesn't have to know about all the classes and how to interact them, which methods to call. The only thing a client should know is that it can just create a façade and call one method to achieve the goal. That's all.

There are many patterns which are related to SRP. **Decorator** and **Composite** patterns are also very closely related to the SRP.

According to the definition from the Gang of Four book,

> Composite pattern allows to compose objects into tree structures to represent part-whole hierarchies. Composite pattern lets clients treat individual objects and compositions of objects uniformly.

According to the same book, the Decorator pattern

> allows the behavior to be added to an individual object, either statically or dynamically, without affecting the behavior of other objects from the same class. The decorator pattern is often useful for adhering to the Single Responsibility Principle, as it allows functionality to be divided between classes with unique areas of concern.

We're not going to discuss design patterns in this book deeply, so I can just recommend you some sources to look at to get a deeper understanding of design patterns and their relationship to SOLID principles.

Conclusion

Congratulations! In this chapter, you learned the Single Responsibility Principle, what this principle is about and how to adhere this principle to make the code base more maintainable.

So, let's sum up.

I like the definition of the SRP from Uncle Bob, > There should never be more than one reason for a class to change.

- Applying SRP we want to separate different concerns. **A class should do one thing**, and do it well!
- SRP **can be applied at different levels**: at the object level, at function level, and module level.
- Classes which accumulate **many responsibilities are hard to understand**.
- **When SRP is violated**, **responsibilities** start to collate with each other, what means that they **become coupled**.
- At the same time when abusing the SRP, you can end up **with too many small classes**, and methods with business logic spread among them. In such a case, you might want to **apply the Façade pattern** we discussed in the "Related Patterns" lecture.
- Remember, that those **modules which are expected to change frequently should be isolated from the other parts of the system**.

Concerning this problem, I also want to add that in general, I agree that using interfaces with a single implementation can lead to less maintainable code, but sometimes the isolation provided by such design is exactly what we need to isolate a module that changes frequently.

In the next chapter, we're going to learn the Open/Closed Principle, one of the most misunderstood principles.