OBJECT-ORIENTED PROGRAMMING IN C#

SUCCINCTLY

BY **SANDER ROSSEL**

SUCCINCTLY EBOOK SERIES

Syncfusion®

Object-Oriented Programming in C# Succinctly

By
Sander Rossel

Foreword by Daniel Jebaraj

Syncfusion®
Deliver innovation with ease®

Important licensing information. Please read.

Technical Reviewer: James McCaffrey

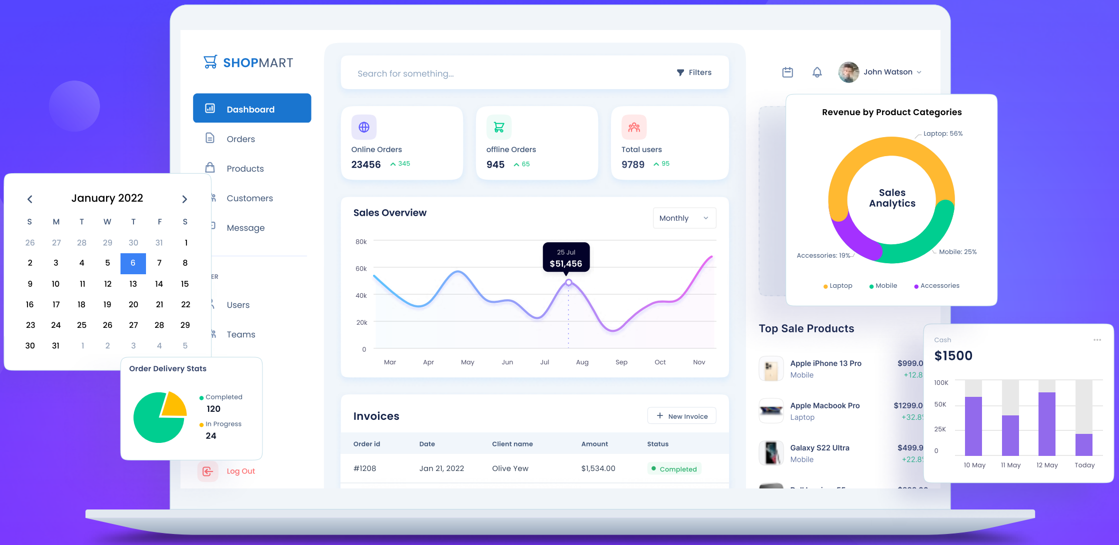
Copy Editor: Darren West, content producer, Syncfusion, Inc.

Acquisitions Coordinator: Hillary Bowling, online marketing manager, Syncfusion, Inc.

Proofreader: Darren West, content producer, Syncfusion, Inc.

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

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 as 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.

About the Author

Sander Rossel is a professional developer with over five years of working experience in .NET (VB and C#, WinForms, MVC, Entity Framework), JavaScript, and SQL Server.

He has an interest in various technologies including, but not limited to, functional programming, NoSQL, and software design.

He seeks to educate others on his blog, Sander's Bits – Writing the code you need, and on his CodeProject profile.

In his spare time he likes to play games, watch a movie, and listen to music. He currently lives with his cat, Nika, in the Netherlands.

Introduction to OOP

Object-oriented programming, or OOP for short, has been around since the 60's and is now the de facto standard programming paradigm. The programming language Simula first adopted OOP concepts in the 60's. Later, these concepts were taken further by the first pure OOP language, Smalltalk. OOP started to really pick up steam in the 90's with languages such as Java. C# and .NET came in 2000.

OOP is a powerful concept that solves many problems found in software development. OOP is not the holy grail of programming, but, as we will see throughout this book, it can help in writing code that is easy to read, easy to maintain, easy to update, and easy to expand.

The concepts in this book are not unique to C#. Other object-oriented languages, such as Java, C++ and Python, share principles that are discussed throughout this book.

The included code samples were all tested in the free Community Edition of Visual Studio 2015 and were created in .NET 4.5 (though most will also run in earlier versions of .NET).

Throughout this book I'm assuming basic knowledge of C#. If you know how to write a class and declare a variable, you're pretty much good to go.

Why OOP?

OOP is all about architecting your software. Let's compare software to a house. Would you want a bunch of builders to just start building your house without predefined rules and agreements? I bet you wouldn't! Your house would be a mess. Still, that's what often happens in software and, as expected, a lot of software turns out a mess! Returning to our house metaphor, let's say your lamps were built right into the ceiling. Whenever a lamp broke you'd need a new ceiling! Luckily, that's not the case. Yet in software, when a small detail needs to change, we often find ourselves rewriting huge pieces of code. And in many cases functionality that had nothing to do with the change breaks anyway. By abstracting away certain functionality we can just worry about the detail and we're sure other parts of the system won't break. On top of that we can reuse code so that if functionality needs to change we're sure it's changed everywhere where we use it. How that's done will become clear throughout the book.

Terminology

Before we dive into OOP it's important that we get some terminology straight.

Often, I see the terms *class* and *object* used interchangeably. Let's get that out of the way, as they are two different things. A class is a blueprint for an object. It contains the methods and properties that will eventually define the behavior of an object. An object is the actual *instance* of a class, created at runtime using the **`new`** keyword.

In the following code listing we'll see an example of a **`Person class`**. It doesn't do anything, it just sits there.

Code Listing 1: A Class

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string GetFullName()
    {
        return FirstName + " " + LastName;
    }
}
```

Now the usage of this class could look as follows:

Code Listing 2: Object Instantiation and Usage

```
Person personObject = new Person();
personObject.FirstName = "Sander";
personObject.LastName = "Rossel";
string fullName = personObject.GetFullName();
```

In the previous code listing, we can see how an object is instantiated from a class by calling its **`constructor`** (using the **`new`** keyword).

Let's go a bit further. Each object has one or more *types*. A type is defined by the class that was used to instantiate an object. For example, our **`personObject`** has the type **`Person`**. One type all objects in C# share is the type **`Object`** (or, including its namespace, **`System.Object`**). This is possible because C# supports *Inheritance*, but we'll get into that later. We have a couple of ways to check the type of an object, for example, we can use **`GetType`**, **`typeof`** or the **`is`** operator. For now, let's move on.

Last I'd like to mention packages. A *package* is a common name for a set of code that is compiled together. In Microsoft land a package is often called a DLL. In .NET a DLL is often known as an *Assembly*. In Visual Studio each project is compiled into a DLL or an exe (executable) file.

Chapter 1 The Three Pillars of OOP

Object-oriented programming has three characteristics that define the paradigm, the so-called “three pillars of OOP.” In this chapter, we’ll look at each one of them in detail. They all play an important role in the design of your applications. That’s not to say that your systems will have a great design by simply applying these concepts. In fact, there is no single correct way of applying these concepts, and applying them correctly can be difficult.

Inheritance

Inheritance is an important part of any object-oriented language. Classes can inherit from each other, which means the inheriting class gets all of the behavior of the inherited class, also known as the base class. Let’s look at the Person example I used earlier.

Code Listing 3: Inheritance Example

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string GetFullName()
    {
        return FirstName + " " + LastName;
    }
}

public class Employee : Person
{
    public decimal Salary { get; set; }
}
```

The trick is in the **`Employee : Person`** part. That part basically says “**`Employee`** inherits from **`Person`**”. And remember everything inherits from **`Object`**. In this case **`Employee`** inherits from **`Person`** and **`Person`** (because it’s not explicitly inheriting anything) inherits from **`Object`**. Now let’s look at how we can use Employee.

Code Listing 4: Subclass usage

```
Employee employee = new Employee();
employee.FirstName = "Sander";
employee.LastName = "Rossel";
string fullName = employee.GetFullName();
employee.Salary = 1000000; // I wish! :-)
```

That’s pretty awesome! We got everything from **`Person`** just by inheriting from it! In this case we can call **`Person`** a *base class* or *superclass* and **`Employee`** a *subclass*. Another common way of saying it is that **`Employee`** *extends* **`Person`**.

There's a lot more though! Let's say we'd like to write some common behavior in some base class, but we'd like subclasses to be able to extend or override that behavior. Let's say we'd like subclasses to change the behavior of **`GetFullName`** in the previous example. We can do this using the **`virtual`** keyword in the base class and **`override`** in the subclass.

Code Listing 5: Method overriding

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public virtual string GetFullName()
    {
        return FirstName + " " + LastName;
    }
}

public class Employee : Person
{
    public decimal Salary { get; set; }
    public override string GetFullName()
    {
        string originalValue = base.GetFullName();
        return LastName + ", " + FirstName;
    }
}
```

As you can see we can override, or re-define, **`GetFullName`** because it was marked **`virtual`** in the base class. We can then call the original method using the **`base`** keyword (which points to the implementation of the base class) and work with that, or we can return something completely different. Calling the original base method is completely optional, but keep in mind that some classes may break if you don't.

Now here's an interesting thought: **`Employee`** has the type **`Employee`**, but it also has the type **`Person`**. That means that in any code where we need a **`Person`** we can actually also use an **`Employee`**. When we do this we can't, of course, access any **`Employee`** specific members, such as **`Salary`**. So here's a little question: what will the following code print?

Code Listing 6: What will the code print?

```
Person person = new Employee();
person.FirstName = "Sander";
person.LastName = "Rossel";
string fullName = person.GetFullName();
Console.WriteLine(fullName);
// Press any key to quit.
Console.ReadKey();
```

If you answered "Rossel, Sander" (rather than "Sander Rossel") you were right!

What else can we do? We can force a subclass to inherit certain members. When we do this we must mark a method, and with that the entire class, as **`abstract`**. An abstract class can't be instantiated and must be inherited (with all abstract members overridden).

Code Listing 7: An Abstract Class

```
public abstract class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public abstract string GetFullName();
}

public class Employee : Person
{
    public decimal Salary { get; set; }
    public override string GetFullName()
    {
        return LastName + ", " + FirstName;
    }
}
```

Of course we can't make a call to **`base.GetFullName()`** anymore, as it has no implementation. And while overriding **`GetFullName`** was optional before, it is mandatory now. Other than that the usage of **`Employee`** stays exactly the same.

On the other end of the spectrum, we can explicitly state that a class or method may not be inherited or overridden. We can do this using the **`sealed`** keyword.

Code Listing 8: A Sealed Class

```
public sealed class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string GetFullName()
    {
        return FirstName + " " + LastName;
    }
}
```

Now that **`Person`** is **`sealed`** no one can inherit from it. That means we can't create an **`Employee`** class and use **`Person`** as a base class.

Methods can only be sealed in subclasses. After all, if you don't want people to override your method, simply don't mark it **`virtual`**. However, if you do have a virtual method and a subclass wants to prevent further overriding behavior it's possible to mark it as **`sealed`**.

Code Listing 9: A sealed method

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public virtual string GetFullName()
    {
        return FirstName + " " + LastName;
    }
}
```

```
public class Employee : Person
{
    public decimal Salary { get; set; }
    public sealed override string GetFullName()
    {
        return LastName + ", " + FirstName;
    }
}
```

No subclass of **`Employee`** can now override **`GetFullName`**.

Why would you ever use **`sealed`** on your classes or methods? First, there is a small performance gain because the .NET runtime doesn't have to take overridden methods into account. The gain is negligible though, so that's not really a good reason. A better reason is perhaps because a class implements some security checks that really shouldn't be extended in any way.

Note: Some languages, like C++, know multiple inheritance. That means a subclass can inherit from more than one base class. In C# this is not possible; each subclass can inherit from, at most, one base class.

Inheritance vs. Composition

While inheritance is very important in OOP, it can be a real pain, too, especially when you get huge inheritance trees with branches to all sides. There is an alternative to inheritance: *composition*. Inheritance implies an "is-a" relationship between classes. An **`Employee`** *is a* **`Person`**. Using composition, we can define a "has-a" relationship.
Let's take a car. A car is built up from many components, like an engine. When we model a car do we inherit **`Car`** from **`Engine`**? That would be problematic, because a car also has doors, chairs and a backseat.

Code Listing 10: No Go

```
public class Engine
{
    // ...
}

public class Car : Engine
{
    // ...
}
```

How would this look when using composition?

Code Listing 11: Composition

```
public class Engine
{
    // ...
}

public class Car
{
    private Engine engine = new Engine();
    // ...
}
```

`Car` can now use the **`Engine`**, but it is not an **`Engine`**!

We could've used this approach with **`Person`** and **`Employee`** as well, but how would we set **`FirstName`** and **`LastName`**? We could make **`Person`** public, but we are now breaking a principle called encapsulation (as discussed in the next chapter). We could mimic **`Person`** by defining a **`FirstName`** and **`LastName`** property, but we now have to change the public interface of **`Employee`** every time the public interface of **`Person`** changes. Additionally, **`Employee`** will not be of type **`Person`** anymore, so the type of **`Employee`** changes and it will not be interchangeable with **`Person`** anymore.

A solution will be presented in Chapter 2: Interfaces.

Encapsulation

Encapsulation is the process of hiding the internal workings of our classes. That means we specify a public specification, used by consumers of our class, while the actual work is hidden away. The advantage is that a class can change how it works without needing to change its consumers.

In C# we have four *access modifiers* keywords which enable five ways of controlling code visibility:

- **`private`**—only visible to the containing class.
- **`protected`**—only visible to the containing class and inheritors.
- **`internal`**—only visible to classes in the same assembly.
- **`protected internal`**—only visible to the same assembly or in derived classes, even in other assemblies.
- **`public`**—visible to everyone.

Let's say we're building some class that runs queries on a database. Obviously we need some method of **`RunQuery`** that's visible to every consumer of our class. The method for accessing the database could be different for every database, so perhaps we're leaving that open for inheritors. Additionally, we use some helper class that's only visible to our project. Last, we need to store some private state, which may not be altered from outside our class as it could leave it in an invalid state.

Code Listing 12: Access Modifiers

```
public class QueryRunner
{

    private IDbConnection connection;

    public void RunQuery(string query)
    {
        Helper helper = new Helper();
        if (helper.Validate(query))
        {
            OpenConnection();
            // Run the query...
            CloseConnection();
        }
    }

    protected void OpenConnection()
    {
        // ...
    }

    protected void CloseConnection()
    {
        // ...
    }

}

internal class Helper
{
    internal bool Validate(string query)
    {
        // ...
        return true;
    }
}
```

If we were to compile this into an assembly and access it from another project we'd only be able to see the **`QueryRunner`** class. If we'd create an instance of the **`QueryRunner`** we could only call the **`RunQuery`** method. If we were to inherit **`QueryRunner`** we could also access **`OpenConnection`** and **`CloseConnection`**. The **`Helper`** class and the **`connection`** field will be forever hidden from us though.

I should mention that classes can contain nested private classes, classes that are only visible to the containing class. Private classes can access private members of their containing classes. Likewise, an object can access private members of other objects of the same type.

Code Listing 13: A private nested class

```
public class SomeClass
{

    private string someField;
    public void SomeMethod(SomeClass otherInstance)
    {
        otherInstance.someField = "Some value";
    }

    private class InnerClass
    {

        public void SomeMethod(SomeClass param)
        {
            param.someField = "Some value";
        }

    }

}
```

When omitting an access modifier a default is assigned (**`internal`** for classes and **`private`** for everything else). I'm a big fan of explicitly adding access modifiers though.

A last remark, before moving on, is that subclasses cannot have an accessibility greater than their base class. So if some class has the **`internal`** access modifier an inheriting class cannot be made **`public`** (but it could be **`private`**).

Polymorphism

We've seen inheritance and that we can alter the behavior of a type through inheritance. Our **`Person`** class had a **`GetFullName`** method which was altered in the subclass **`Employee`**. We've also seen that whenever, at run-time, an object of type **`Person`** is expected we can throw in any subclass of **`Person`**, like **`Employee`**. This is called polymorphism.

In the following example the **`PrintFullName`** method takes an object of type **`Person`**, but it prints "Rossel, Sander" because the parameter that's passed into the method is actually of subtype **`Employee`**, which overrides the functionality of **`GetFullName`**.

Code Listing 14: Polymorphism

```
class Program
{
    static void Main(string[] args)
    {
        Person p = new Employee();
        p.FirstName = "Sander";
        p.LastName = "Rossel";
        PrintFullName(p);
        // Press any key to quit.
        Console.ReadKey();
    }

    public static void PrintFullName(Person p)
    {
        Console.WriteLine(p.GetFullName());
    }
}

public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public virtual string GetFullName()
    {
        return FirstName + " " + LastName;
    }
}

public class Employee : Person
{
    public decimal Salary { get; set; }
    public sealed override string GetFullName()
    {
        return LastName + ", " + FirstName;
    }
}
```

We’re going to see a lot more of this in the next chapter.

The Takeaway

The Three Pillars of OOP are the foundation of object-oriented programming. They haven’t been implemented for nothing and they do solve real problems. It’s crucial that you know these features by heart and practice them in your daily code. Think about encapsulation every time you create a class or method. Use inheritance when necessary, but don’t forget it brings extra complexity to the table as well. Be very wary of polymorphism, know which code will run when you inherit classes and override methods. Even if you don’t practice it you’ll come across code that does. Throughout this book we’ll see these be used extensively.

Chapter 2 Interfaces

In the previous chapter, we saw that the class **`Employee`** inherits from **`Person`**. Do you see any potential issues with this approach? What if a person is an employee, but also a stamp collector? Obviously, a person can also be a stamp collector but not an employee (lots of people are "between jobs"). If all we had was inheritance we might write the following code:

Code Listing 15: An inheritance chain

```
public class Person
{
    // ...
}

public class Employee : Person
{
    // ...
}

public class StampCollector : Person
{
    // ...
}

public class EmployeeStampCollector : Employee
{
    // ...
}
```

I don't know about you, but it just doesn't feel right to me. The problem we have is that the only thing the **`EmployeeStampCollector`** and the **`StampCollector`** have in common is the **`Person`** base class, but they're still both stamp collectors! Writing a method that takes a stamp collector as input is now impossible! This problem could be solved using multiple inheritance, but C# doesn't have that (and many developers say it is better that way).

So how do we solve this problem? We use an *Interface*. An Interface is something like an abstract class with only abstract methods. It really doesn't do more than defining a type. The good part though, is that we can inherit, or 'implement', multiple interfaces in a single class!

Let's look at an example of a random interface:

Code Listing 16: An interface

```
public interface ISomeInterface
{
    string SomeProperty { get; set; }
    string SomeMethod();
    void SomethingElse();
}
```

Notice that none of the fields have an access modifier. If a class implements an interface everything on that interface is publicly accessible. This is what the actual implementation would look like:

Code Listing 17: Interface implementation

```
public class SomeClass : ISomeInterface
{
    public string SomeProperty { get; set; }

    public string SomeMethod()
    {
        // ...
    }

    public void SomethingElse()
    {
        // ...
    }
}
```

So here **`SomeClass`** has the types **`object`**, **`SomeClass,`** and **`ISomeInterface`**. The “I” prefix on **`ISomeInterface`**, or any interface, is common practice in C#, but not necessary.

When using inheritance and interface implementation on a single class, you first specify the base class and then use a comma-separated list of interfaces. How would this look for our **`Person`**, **`Employee`** and **`StampCollector`** example?

Code Listing 18: Interfaces

```
public interface IEmployee
{
    // ...
}

public interface IStampCollector
{
    // ...
}

public class Person
{
    // ...
}

public class Employee : Person, IEmployee
{
    // ...
}

public class StampCollector : Person, IStampCollector
{
    // ...
}

public class EmployeeStampCollector : Employee, IStampCollector
```

```
{
    // ...
}
```

Now both **StampCollector** and **EmployeeStampCollector** are of type **Person** and of type **IStampCollector**. **EmployeeStampCollector** is also of type **Employee** (and **IEmployee**) because it inherits from **Employee**, which implements **IEmployee**.

Because a class can implement multiple interfaces it is possible to have an interface inherit from multiple interfaces. A class must then simply implement all interfaces that are inherited by the interface. For obvious reasons, an interface can't inherit a class.

Code Listing 19: Interface inheritance

```
public interface IPerson { }
public interface IEmployee : IPerson
{ }
public interface IStampCollector : IPerson
{ }
public interface IEmployeeStampCollector : IEmployee, IStampCollector
{ }
```

Now that's a very theoretical example, but let's consider a real world example. Let's say our application needs to log some information. We may want to log to a database in a production environment, but we'd also like to log to the console for debugging purposes. Additionally, in case the database isn't available, we'd like to log to the Windows Event logs.

Code Listing 20: Loggers

```
class Program
{
    static void Main(string[] args)
    {
        List<ILogger> loggers = new List<ILogger>();
        loggers.Add(new ConsoleLogger());
        loggers.Add(new WindowsLogLogger());
        loggers.Add(new DatabaseLogger());

        foreach (ILogger logger in loggers)
        {
            logger.LogError("Some error occurred.");
            logger.LogInfo("All's well!");
        }
        Console.ReadKey();
    }
}

public interface ILogger
{
    void LogError(string error);
    void LogInfo(string info);
}

public class ConsoleLogger : ILogger
{
    public void LogError(string error)
    {
        Console.WriteLine("Error: " + error);
    }

    public void LogInfo(string info)
    {
        Console.WriteLine("Info: " + info);
    }
}

public class WindowsEventLogLogger : ILogger
{
    public void LogError(string error)
    {
        Console.WriteLine("Logging error to Windows Event log: " + error);
    }

    public void LogInfo(string info)
    {
        Console.WriteLine("Logging info to Windows Event log: " + info);
    }
}

public class DatabaseLogger : ILogger
{
    public void LogError(string error)
    {
        Console.WriteLine("Logging error to database: " + error);
    }

    public void LogInfo(string info)
    {
        Console.WriteLine("Logging info to database: " + info);
    }
}
```

That's pretty nifty! And as you can imagine, we can add or remove loggers as we please.

By the way, having both an interface and a base class (which implements the interface) is perfectly fine. Maybe you have an **`ILogger`**, a **`DbLogger`** (which implements **`ILogger`** and defines some common behavior for logging to a database) and then have a **`SqlServerLogger`**, an **`OracleLogger`**, a **`MySqlLogger`**, etc. (which all inherit from **`DbLogger`**).

Explicitly Implementing Interfaces

I've mentioned that when you implement an interface, all members of that interface are public by default. That's logical behavior; after all, an interface is a sort of contract that promises other code that it will have some methods and properties defined (because it's of a certain type). It is, however, possible to hide interface members. To do that, you can explicitly implement an interface. When you explicitly implement an interface member, the only way to invoke that member is by using the class as a type of the interface.

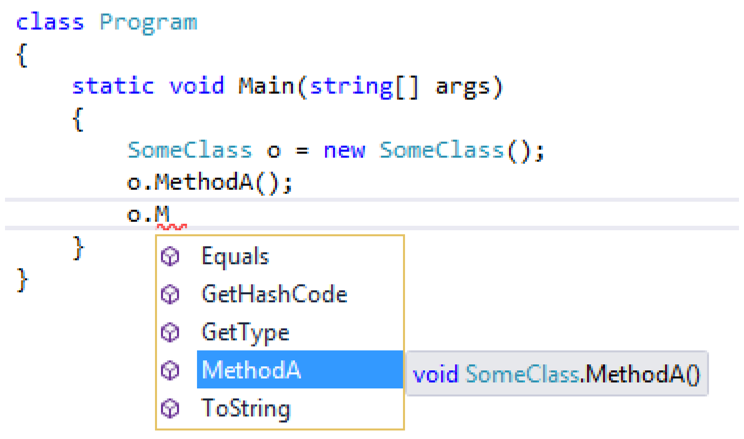
Code Listing 21: Explicitly Implemented Interface

```
public interface ISomeInterface
{
    void MethodA();
    void MethodB();
}

public class SomeClass : ISomeInterface
{
    public void MethodA()
    {
        // Even SomeClass can't invoke MethodB without a cast.
        ISomeInterface me = (ISomeInterface)this;
        me.MethodB();
    }

    // Explicitly implemented interface member.
    // Not visible in SomeClass.
    void ISomeInterface.MethodB()
    {
        throw new NotImplementedException();
    }
}
```

Users of **`SomeClass`** can't invoke **`MethodB`** either.



```
class Program
{
    static void Main(string[] args)
    {
        SomeClass o = new SomeClass();
        o.MethodA();
        o.M
    }
}
```

Figure 1: MethodB is not accessible.

Unless they use it as, or cast it to, **`ISomeInterface`**.

Code Listing 22: Invoking an Explicitly Implemented Member

```
ISomeInterface obj = new SomeClass();
obj.MethodA();
obj.MethodB();
```

Inheritance vs. Interface

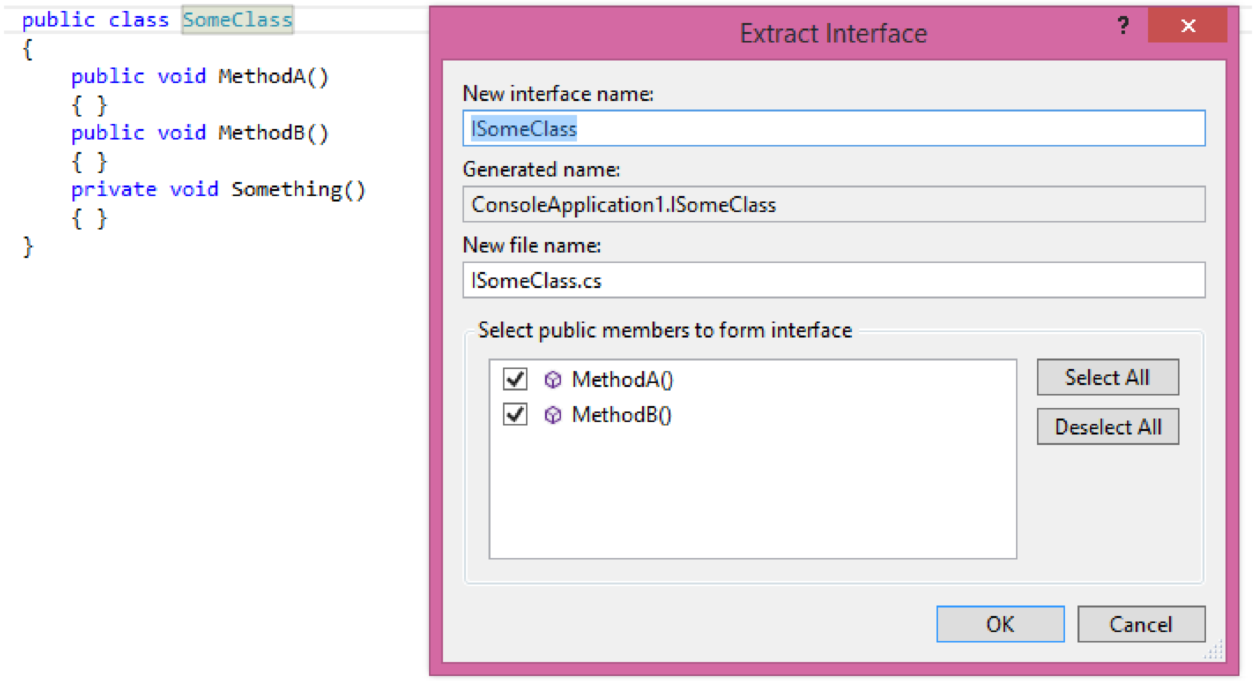
One discussion you will find frequently online, or might get engaged in at work, is that of inheritance vs. using an interface. Remember that you can only inherit from one class. That means if you use inheritance, like **`Person`** -> **`Employee`**, your **`Employee`** is stuck with the **`Person`** base class. Alternatively, you could define an **`IPerson`** and implement it separately in **`Person`** and **`Employee`**. One guideline you'll often see is that inheritance defines an "is-a" relationship (an **`Employee`** *is a* **`Person`**) while interface implementation defines a "can do" or "has-a" relationship (a **`Person`** *has a* stamp collection). As you see, it's not entirely fool proof, as a stamp collector *is* also a **`Person`**, but still inheritance can lead us into trouble.

I like to use inheritance when I have some functionality that is shared across multiple, if not all, subtypes of that class, like with the **`DbLogger`** example.

And remember, composition may also be a viable option!

The Takeaway

An interface is a means to create additional types without the need for multiple inheritance. They are extremely useful. I've talked to people who say every class needs a matching interface. It's such an important feature that Visual Studio even has the option to create an interface based on a class. When your cursor is in a class, you can find it under Edit -> Refactor -> Extract Interface in the top menu.



```
public class SomeClass
{
    public void MethodA()
    { }
    public void MethodB()
    { }
    private void Something()
    { }
}
```

Extract Interface

New interface name: ISomeClass

Generated name: ConsoleApplication1.ISomeClass

New file name: ISomeClass.cs

Select public members to form interface

☑ MethodA()
☑ MethodB()

Figure 2: Extract Interface Dialog

Learn to use them well, as we'll use them a lot throughout this book.

Chapter 3 SOLID

We've seen the building blocks that make up an object-oriented language, but how can we write our classes so that they make sense? When to inherit, what to put inside a class, how many interfaces do we need? These are questions that can be answered in part by the so-called SOLID principles. SOLID is an acronym and each letter stands for a principle.

- Single Responsibility Principle (SRP)
- Open Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

In this chapter we're going to look at all of them. Knowing these principles can help you write modular code that is easy to maintain and expand. The principles are described by Robert C. Martin in his 2006 book Agile Principles, Patterns, and Practices in C#[1].

Single Responsibility Principle (SRP)

The Single Responsibility Principle, or SRP, states that a class should be responsible for a single functionality. A functionality can be considered a reason to change. For example, a class that logs errors to a database could change because the database communication should change, or because the format of the log output needs to change. So more generally, a class should have only one reason to change. That's a little tricky; after all, each line of code could be changed at one time or another. And indeed, I have met programmers who believed a class should contain only a single method. That's really not what SRP is about though!

Let's look at a concrete example. Let's say you're building a database module. Now you've got a class that can establish a connection to the database, get and send data, and finally close the connection. If we defined an interface it could look as follows:

Code Listing 23: A Possible Violation of SRP

```
public interface IDatabase
{
    void Connect(string connectionString);
    void Close();
    object GetData();
    void SendData(object data);
}
```

[1] R.C. Martin, M. Martin - Agile Principles, Patterns and Practices in C#

So this could really be the interface to your class, right? Unfortunately, this class is doing two things. The class deals with opening and closing connections and with data communication. Here comes the tricky part, and this may or may not be a problem. The question you should ask yourself is this: will the application change in a way that either connections or data communications will change, but not both? This is important, because when the time comes and you realize you need to change one of the two functionalities, the functionality that doesn't need to change will be in the way. The more functionality that's in the way, the more chances you have of breaking those parts of the application.

When applying SRP, your classes will be a lot smaller and more maintainable, but you'll have more classes to deal with. Which would you rather change, a class with 200 lines of code that does multiple things, or a class with 100 lines of code that does only one thing?

As you see, there is no wrong or right in SRP. Unfortunately, it's a bit of a gut feeling. If you feel a class is to bloated, try identifying different responsibilities and give each their own class.

Open Closed Principle (OCP)

Without a doubt, one of the hardest principles to grasp and get right is the Open Closed Principle, or OCP. "Open Closed" means that a class should be open for extension, but closed for modification. That means consumers of your class should be able to customize its usage, but without actually changing its code. And unless your coworkers are working in the same solution consumers of your class, even they probably can't change the source code of your class.

Let's say you've built that database class we just talked about. In a sense it's already extensible, since everyone can inherit from it and add methods and properties. Doing so will almost certainly result in breaking SRP though, since the core functionality of the class is already defined in the base class and we can't change that. You could make every method in the base class **`virtual`**, and I've seen that happen, but that's not really good practice as consumers may now break your class. For example, if they **`override Connect`** and just leave it empty, no connection will be made and each subsequent method call to your class will fail. Besides, even if the method is overridden correctly the base class will now just be sitting there doing nothing, which doesn't feel right.

I propose a different solution, one of many. Let's break up the previously proposed interface into two separate interfaces. You'll probably need some more methods now, but let's stick to simplicity for now.

Code Listing 24: SRP for OCP

```
public interface IConnectionManager
{
    void Connect(string connectionString);
    void Close();
}

public interface IDataManager
{
    object GetData(IConnectionManager connManager);
    void SendData(IConnectionManager connManager, object data);
}
```

Now suppose I'd like to change the way I connect to the database. I can implement my own IConnectionManager and pass it to my IDataManager. Without touching the original code, I am now able to extend the functionality of the classes without actually breaking the already existing class! Likewise, if I'd like to change the way I'm getting or sending data I can simply implement my own IDataManager and use it with the already existing implementation of IConnectionManager.

It gets better, though. Every class that depends on IConnectionManager and/or IDataManager is now extendable in that it can use any database you can ever imagine! Well, theoretically.

Later we'll look at Design Patterns, which can help you in designing classes that conform to OCP.

Liskov Substitution Principle (LSP)

The Liskov Substitution Principle, LSP for short, is about inheritance. It states that subclasses and base classes must be substitutable. That means that any method that consumes an object of any base type must not know if it's using a base class or any subclass. Here's an example of what you wouldn't do (although I must admit I've written code just like this in the past):

Code Listing 25: Violation of LSP

```
bool TestConnection(IConnectionManager connMngr)
{
    if (connMngr is SqlServerConnectionManager)
    {
        // Do something...
    }
    else if (connMngr is OracleConnectionManager)
    {
        // Do something else...
    }
    else
    {
        // ...
    }
}
```

`TestConnection` is a function that receives an object of type **`IConnectionManager`**, which could be any connection manager. In this case, the method checks for the type of **`connMngr`** and acts accordingly. That means that for every new implementation of **`IConnectionManager,`** you need to change this method. Furthermore, you've just violated OCP as this function is now unable to test new connection types. There's plenty of ways to fix this, like using a Strategy Pattern, but we'll get to that in a later chapter. In the above method it should always be good enough to invoke the **`Connect()`** or **`Close()`** method, no matter what **`IConnectionManager`** you get.

A more subtle violation of LSP is when subclasses behave differently than their base classes. The classic example is that of a **`Square`** class that inherits from a **`Rectangle`** class. A square is a rectangle (with equal width and height), so this inheritance looks plausible. Unfortunately, in some scenarios a square may behave differently than a rectangle and may break users of the **`Rectangle`** class and get a **`Square`** instead.

Code Listing 26: Breaks with Square

```
void TestSquareMeters(Rectangle rect)
{
    rect.Height = 2;
    rect.Width = 3;
    Assert(rect.Height * rect.Width == 6);
}
```

Interface Segregation Principle (ISP)

The Interface Segregation Principle, or ISP, is kind of like the Single Responsibility Principle for Interfaces. Strictly speaking, we're not talking about the abstract-class-like Interfaces, but your class' public methods. In C# practice, though, we're doing ISP mostly through Interfaces. Let me clear that up. Let's consider our database example for a second. We had the following interface:

Code Listing 27: An IDatabase Interface

```
public interface IDatabase
{
    void Connect(string connectionString);
    void Close();
    object GetData();
    void SendData(object data);
}
```

As I mentioned in the bit about SRP, this may or may not be SRP-proof, depending on your requirements. Let's say it is okay for our example, but let's take another approach, that of inheritance. More specifically, I'm going to create a base class for connection management and inherit that for our data management.

Code Listing 28: DatabaseManager Using Inheritance

```
public class ConnectionManager
{
    public void Connect(string connectionString)
    {
        // ...
    }

    public void Close()
    {
        // ...
    }
}
```

```
public class DataManager : ConnectionManager
{
    public virtual object GetData()
    {
        // ...
    }

    public virtual void SendData(object data)
    {
        // ...
    }
}

public class DatabaseManager : DataManager
{
    // Needs ConnectionManager...
}
```

So our **`DatabaseManager`** is inheriting **`ConnectionManager`**, through **`DataManager`**, to make use of, of course, its base implementation, but also to make use of multipurpose functions such as **`TestConnection`** defined earlier. Can we be sure that all our data managers need connection managers though? Sure, the **`DatabaseManager`** needs it, but what about our **`FileDataManager`**?

Code Listing 29: FileDataManager

```
public class FileDataManager : DataManager
{
    // Doesn't need ConnectionManager...
}
```

Now our **`FileDataManager`** has a *fat interface*. It depends on classes it doesn't need! Notice that this would not have happened if we just created two Interfaces and implemented those instead. We could even use composition to re-use the **`ConnectionManager`** in other **`Data(base)Managers`**. The downside to this is that **`FileDataManager`** now depends upon **`ConnectionManager,`** even though it doesn't need it. Any change to **`ConnectionManager`** may now break **`FileDataManager`** though! Such couplings between classes should be avoided where possible.

Dependency Inversion Principle (DIP)

The Dependency Inversion Principle, DIP for short, simply states that you should depend on abstractions instead of on concrete types. This is one principle you're going to see a lot but which is pretty hard to put into practice (correctly). Let's consider our **`TestConnection`** method again. We could create one for our **`SqlDatabaseManager,`** as in the following example:

Code Listing 30: A Violation of DIP

```
class Program
{
    //...
    bool TestConnection(SqlConnectionManager connMngr)
    {
        // ...
    }

}

public interface IConnectionManager
{
    void Close();
    void Connect(string connectionString);
}

public class SqlConnectionManager : IConnectionManager
{
    // ...
}
```

It should be obvious that **`TestConnection`** can now only be used for **`SqlConnectionManager`**. We have created two problems. First, **`TestConnection`**, and **`Program`** with it, now depend on **`SqlConnectionManager`**. Second, we'll have to write a **`TestConnection`** for each **`ConnectionManager`** we're going to write. We're saying that **`SqlConnectionManager`** is a concrete type of the abstract type **`IConnectionManager`**. So let's fix that so that we're relying on abstract types compliant to DIP.

Code Listing 31: Fixed for DIP

```
bool TestConnection(IConnectionManager connMngr)
{
    // ...
}
```

You may think that's a pretty lame solution, and you may even say this will break your existing code because **`TestConnection`** relies on methods that aren't part of the **`IConnectionManager Interface`**. If that's the case, you should ask yourself three things. Is **`TestConnection`** implemented correctly, and does it really only test connections? Is my **`SqlConnectionManager`** really an **`IConnectionManager`**? Finally, is **`IConnectionManager`** defined correctly?

Now why is this principle so important? It has everything to do with expanding and changing existing systems. If you depend on abstractions, it's easier to switch implementations. In theory you could switch your database (or at least the connection manager) and **`TestConnection`** would still behave as expected. We've seen loggers earlier, depend on **`ILogger`** and you can switch from file logging to database logging without problems!

Dependency Injection (DI)

As I said you're going to see DIP a lot. Perhaps you've heard of Dependency Injection (DI) before? There's plenty of "DI Frameworks" that help you put DIP into practice. The previous example was as simple as changing our method signature to using an Interface instead of a concrete Type. But any method still gets a concrete type at runtime, so where does that come from? We'll have to break this "all abstract type" principle somewhere! And that's where DI comes in.

For the next example I'm going to use Unity, a DI library developed by Microsoft and now maintained by other people. You can get it using the NuGet Package Manager in Visual Studio. Just go to Tools > NuGet Package Manager > Manage NuGet Packages for Solution and search for "Unity." Install it in your (saved) project and you're good to go.

A "DI Container," which Unity provides, is a sort of repository where abstract types are mapped to concrete types. Any code can then ask that repository for an instance of an abstract type and the repository will return whatever concrete type you've mapped to it. Your entire code base, save for the part(s) where you map your types, can now depend upon abstractions. Sounds pretty neat, so let's have a look at some code!

I've slightly modified the **`IConnectionManager`** example from earlier:

Code Listing 32: IConnectionManagers

```
public interface IConnectionManager
{
    void Close();
    void Connect();
}

public class SqlConnectionManager : IConnectionManager
{
    public void Close()
    {
        Console.WriteLine("Closed SQL Server connection...");
    }

    public void Connect()
    {
        Console.WriteLine("Connected to SQL Server!");
    }
}

public class OracleConnectionManager : IConnectionManager
{
    public void Close()
    {
        Console.WriteLine("Closed Oracle connection...");
    }

    public void Connect()
    {
        Console.WriteLine("Connected to Oracle!");
    }
}
```

Here's the definition of **`TestConnection`**:

Code Listing 33: Definition for TestConnection

```
static bool TestConnection(IConnectionManager connMngr)
{
    connMngr.Connect();
    connMngr.Close();
    return true;
}
```

Using Unity, we can register either **`SqlConnectionManager`** or **`OracleConnectionManager`** with **`IConnectionManager`**:

Code Listing 34: Registering a Type with Unity

```
class Program
{
    static void Main(string[] args)
    {
        // Create a new DI Container...
        IUnityContainer container = new UnityContainer();
        // ...And register a type!
        container.RegisterType<IConnectionManager, SqlConnectionManager>();
    }
}
```

And elsewhere in the code we can now request an **`IConnectionManager`** and call **`TestConnection`**:

Code Listing 35: Resolving a Type with Unity

```
IConnectionManager connMngr = container.Resolve<IConnectionManager>();
bool success = TestConnection(connMngr);
// Do something with the result...
```

You should now see “Connected to SQL Server!” and “Closed SQL Server connection…” in your console window. Now change the call to **`RegisterType`** so it registers as **`OracleConnectionManager`**. The **`TestConnection`** will now print Oracle instead of SQL Server. Imagine: by changing this one line of code you could change an entire application to use Oracle instead of SQL Server!

Now suppose you have some dynamic user interface and the controls that are shown on screen depend on some settings and/or configuration. Let’s say we have some HTML form defined in the database, and we wish to generate the HTML server side. I’m going to let you figure this one out on yourself, but I’m pretty sure you’ll get it.

Code Listing 36: HTML Generator

```
class Program
{
    static void Main(string[] args)
    {
        IUnityContainer container = new UnityContainer();
        container.RegisterType<IHtmlCreator, HtmlInputTextCreator>("text");
        container.RegisterType<IHtmlCreator, HtmlInputCheckboxCreator>("checkbox");

        var someDbFields = new[]
        {
            new
            {
                Text = "Please enter your name:",
                Type = "text"
            },
            new
            {
                Text = "Do you wish to receive offers?",
                Type = "checkbox"
            }
        };

        // Generate the HTML...
        StringBuilder builder = new StringBuilder();
        foreach (var dbField in someDbFields)
        {
            builder.AppendLine($"<p>{dbField.Text}</p>");
            IHtmlCreator html = container.Resolve<IHtmlCreator>(dbField.Type);
            builder.AppendLine(html.CreateHtml());
        }
        Console.WriteLine(builder.ToString());

        Console.ReadKey();
    }
}

public interface IHtmlCreator
{
    string CreateHtml();
}

public class HtmlInputTextCreator : IHtmlCreator
{
    public string CreateHtml()
    {
        return "<input type=\"text\" />";
    }
}

public class HtmlInputCheckboxCreator : IHtmlCreator
{
    public string CreateHtml()
    {
        return "<input type=\"checkbox\" />";
    }
}
```

See how that just saved you an unwieldy **`switch`**/**`if`** statement? And the best part is that you can add new **`IHtmlCreators`** and types without changing the code that generates the HTML! So you've just created OCP and DIP compliant code (except for the <p> tags, but you can figure that out)!

This isn't a tutorial on using Unity, but it does show the basic principle of DIP and DI containers. The sample in this section demonstrated what is called Interface Injection, but it is also possible to create objects using a specified *constructor*, called Constructor Injection, and automatically set objects on properties, called Property, or Setter, Injection. Other DI Frameworks exist, such as Ninject and Spring.NET. In this example we've written code to setup our container, but other frameworks also work with (XML) config files. All of them work with the same principle though, and depend upon abstractions.

Inversion of Control (IoC)

Often mentioned together with DI is Inversion of Control, or IoC. You may even see the terms being used interchangeably. The basic principle of IoC is that “higher” code depends upon abstractions. We have already seen this in action with **`TestConnection`**. Here’s another fun fact: DI is actually a form of IoC (not necessarily the other way around though)!

The Takeaway

Generally speaking, the SOLID Principles can lead to good, maintainable, extensible, and testable software design at the expense of additional complexity. However, they do not come naturally to a lot of developers. Let’s be honest, it’s hard to think ahead, so if you need a SqlConnectionManager now, why bother with some IConnectionManager that may need to be modified later anyway? It’s a lot easier to simply create a SqlConnectionManager and use it throughout your code. However, you cannot predict the future and, even though you think you’ll never switch to something other than SQL Server now, the future may prove otherwise. When you’ll have to switch to something else, you will wish you had practiced the Dependency Inversion Principle. Likewise, when your API that had always been internal suddenly has to go public, you will wish you had practiced the Open Closed Principle. And that stuff does happen. I’ve heard “that will never happen” a few times too many, so really, think ahead and practice SOLID!

Chapter 4 Design Patterns

Have you ever had that problem that you know you've solved before somewhere else? Many people have. Design Patterns are general solutions to specific problems that many people face. Design Patterns were first systematically organized for the C++ language in the book *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma et al. in 1995[2].

In general there are three kinds of Design Patterns: Creational, Structural, and Behavioral. Not only do they solve recurring problems, a formalized solution to a problem means that we don't re-invent the wheel and that we can communicate more clearly about our code. Any programmer will (hopefully) recognize your use of pattern X and then know how to change your code accordingly.

In this chapter, I'm going to discuss a few (of many) Design Patterns. I'll include a Unified Modeling Language (UML) class diagram of each pattern. UML is outside the scope of this book, so I can't explain how the diagrams should be interpreted. I've added them for reference and they're the "official" UML diagrams, so the class and method names don't match those in the examples. You can use them for visual explanation, but if you don't get them don't worry, it's all explained in the text.

The patterns I'm discussing are mentioned by the Design Patterns book as good starting patterns. They're pretty easy to get into and you'll probably use them a lot as well.

Creational Patterns

Creational Patterns deal with, as the name implies, the creation of objects. With Creational Patterns we can abstract away the process of object instantiation. As we've seen before, we can use composition rather than inheritance. Creating a set of objects that work together as a group can be beneficial, but rather than hard-coding which concrete types to use we can abstract this process like we've seen with DI. In the next section we're going to look at a few Creational Patterns. It's not a full list, but it shows the power they can bring.

[2] Gamma, Helm, Johnson, Vlissides - Design Patterns: Elements of Reusable Object-Oriented Software

Abstract Factory

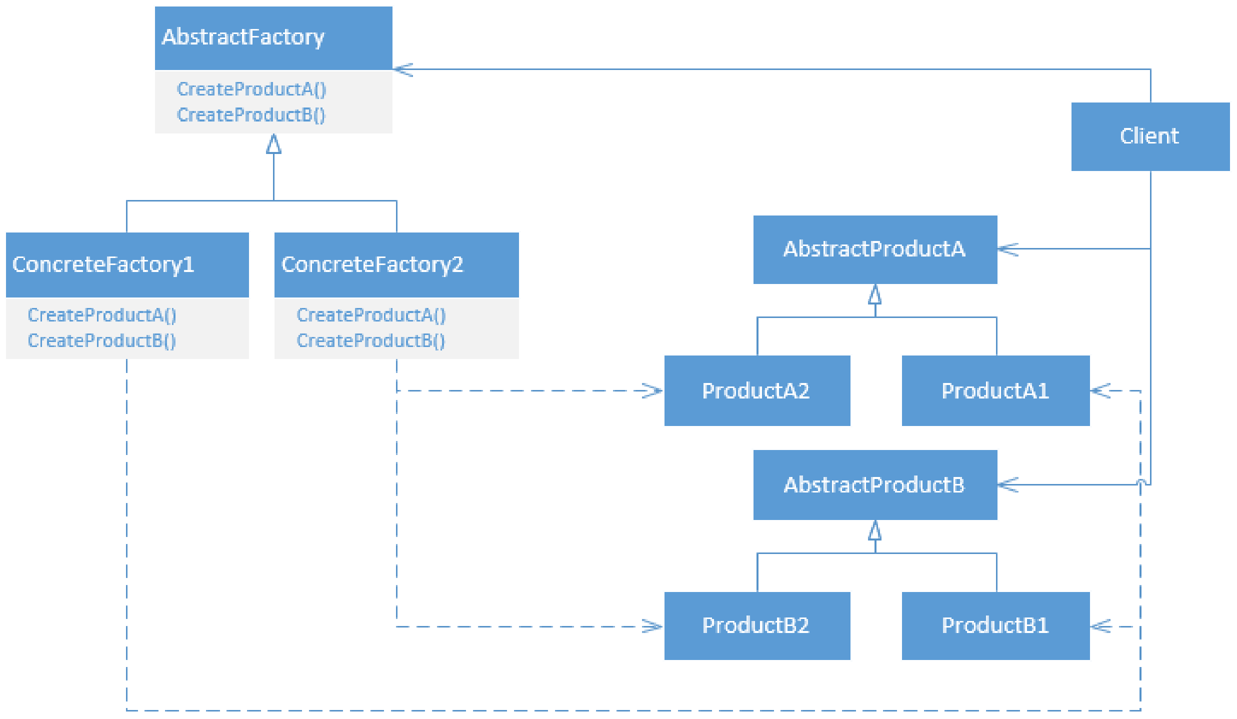


Figure 3: Class Diagram of the Abstract Factory Pattern

One of the easiest, yet most useful, Creational Design Patterns is the Abstract Factory. The Abstract Factory defines an interface for the creation of closely related objects. Imagine, again, that we're building a dynamic user interface. We need textboxes, buttons, checkboxes, radio buttons, etc. So let's say we have the following Interface, analogous to the example I've used before, but in WinForms:

Code Listing 37: An Abstract Factory

```
public interface IControlFactory
{
    Control CreateTextBox();
    Control CreateCheckBox();
}
```

And let's say it's used as follows (which creates a pretty much unusable user interface):

Code Listing 38: Usage of the Abstract Factory

```
private void CreateGui(IControlFactory factory)
{
    Control textBox = factory.CreateTextBox();
    textBox.Location = new Point(10, 10);
    Controls.Add(textBox);
    Control checkBox = factory.CreateCheckBox();
    checkBox.Location = new Point(10, 50);
    Controls.Add(checkBox);
}
```

We can now create concrete types for this Interface.

Code Listing 39: Concrete Factories

```
public class RedControlFactory : IControlFactory
{
    public Control CreateTextBox()
    {
        return new TextBox { BackColor = Color.Red };
    }

    public Control CreateCheckBox()
    {
        return new CheckBox { BackColor = Color.Red };
    }
}

public class GreenControlFactory : IControlFactory
{
    public Control CreateTextBox()
    {
        return new TextBox { BackColor = Color.Green };
    }

    public Control CreateCheckBox()
    {
        return new CheckBox { BackColor = Color.Green };
    }
}
```

Now our application can create either a red or a green GUI using the specified concrete type.

Code Listing 40: Using the Concrete Factory

```
private void Form1_Load(object sender, EventArgs e)
{
    CreateGui(new RedControlFactory());
}
```

The result is stunning:

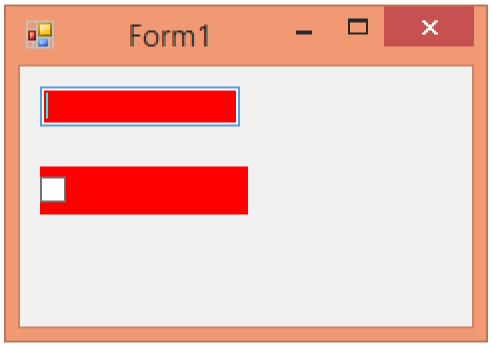


Figure 4: Form Using the RedControlFactory

And of course, changing **`RedControlFactory`** to **`GreenControlFactory`** changes the rendered Form from red to green.

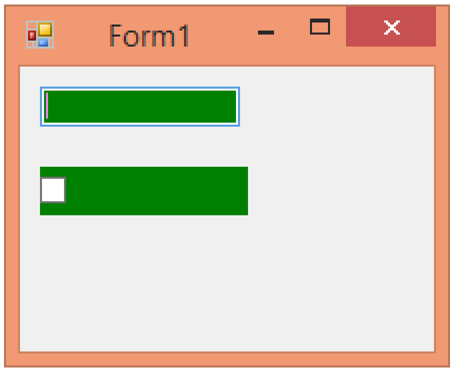


Figure 5: Form Using the GreenControlFactory

Notice how easy it would be to add a **`BlueControlFactory`** without breaking any existing code? Or even better, we can swap our ugly WinForms controls with custom controls or with third party controls.

In .NET you can find the Abstract Factory Pattern when working with databases. The **`DbProviderFactory`** base class is such an Abstract Factory, although it's typically not necessary to implement it yourself.

Code Listing 41: Abstract Factory in .NET

```
public class SqlProviderFactory : DbProviderFactory
{
    public override DbConnectionStringBuilder CreateConnectionStringBuilder()
    {
        return new SqlConnectionStringBuilder();
    }
    public override DbConnection CreateConnection()
    {
        return new SqlConnection();
    }
    public override DbCommand CreateCommand()
    {
        return new SqlCommand();
    }
    // ...
}
```

Factory Method

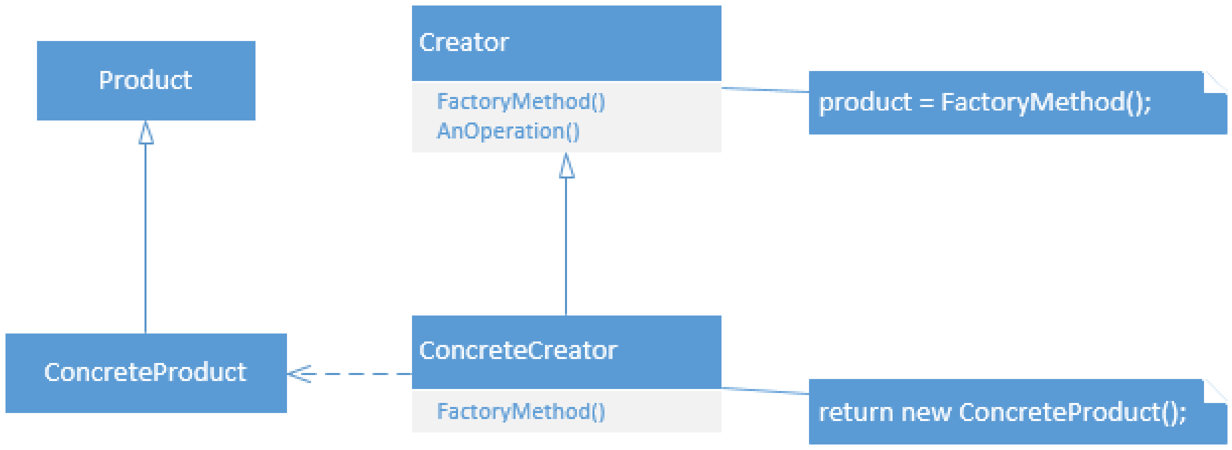


Figure 6: Class Dagram of the Factory Method Pattern

Sometimes it does not make sense to create an Abstract Factory. What if there are no closely related objects, like **`TextBox`**, **`CheckBox`**, **`Button`**, etc.? Having an Abstract Factory to create a single object (which may contain other objects, of course), while possible, feels weird (although can be useful). Suppose we're building some class that generates our UI. The **`UIGenerator`** could make use of an **`IControlFactory`** to construct the UI. Furthermore we may have multiple **`UIGenerators`** to build UIs for different kinds of users.

With the Factory Method Pattern we can delegate the creation of objects to subclasses. This is especially useful when a base class can't anticipate which subclass it'll need to perform its functionality.

The next example is, again, in WinForms.

So here is the **`BaseUIGenerator`** (which will serve as a base class for all **`UIGenerators`**).

Code Listing 42: A Base Class for UIGenerators

```
public abstract class BaseUIGenerator
{

    public void GenerateUI(Control container)
    {
        IControlFactory factory = CreateControlFactory();
        Control textBox = factory.CreateTextBox();
        textBox.Location = new Point(10, 10);
        container.Controls.Add(textBox);
        Control checkBox = factory.CreateCheckBox();
        checkBox.Location = new Point(10, 50);
        container.Controls.Add(checkBox);
    }

    protected abstract IControlFactory CreateControlFactory();
}
```

We can now easily create new **`UIGenerators`**.

Code Listing 43: Concrete UIBuilders

```
public class RedUIGenerator : BaseUIGenerator
{
    protected override IControlFactory CreateControlFactory()
    {
        return new RedControlFactory();
    }
}

public class GreenUIGenerator : BaseUIGenerator
{
    protected override IControlFactory CreateControlFactory()
    {
        return new GreenControlFactory();
    }
}
```

The creation of the **`IControlFactory`** is now delegated to subclasses of **`BaseUIGenerator`**. Now you can use a generator to generate the UI.

Code Listing 44: Usage of the Generator Class

```
private void Form1_Load(object sender, EventArgs e)
{
    BaseUIGenerator generator = new RedUIGenerator();
    builder.GenerateUI(this);
}
```

This will build the form as in the previous example:

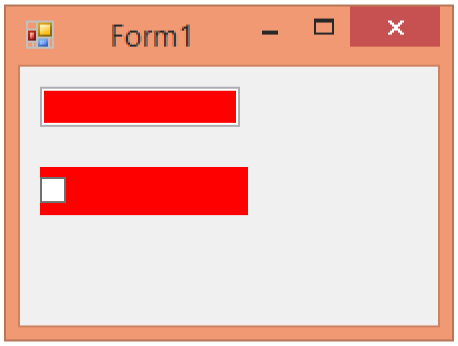


Figure 7: Form Using the RedUIGenerator

It's also possible to use something else as a container, for example a **`Panel`**.

Code Listing 45: Using the Generator with Panels

```
private void Form1_Load(object sender, EventArgs e)
{
    BaseUIGenerator red = new RedUIGenerator();
    red.GenerateUI(panel1);
    BaseUIGenerator green = new GreenUIGenerator();
    green.GenerateUI(panel2);
}
```

The result is that each **`Panel`** now has the same **`Controls`**, but in a different color.

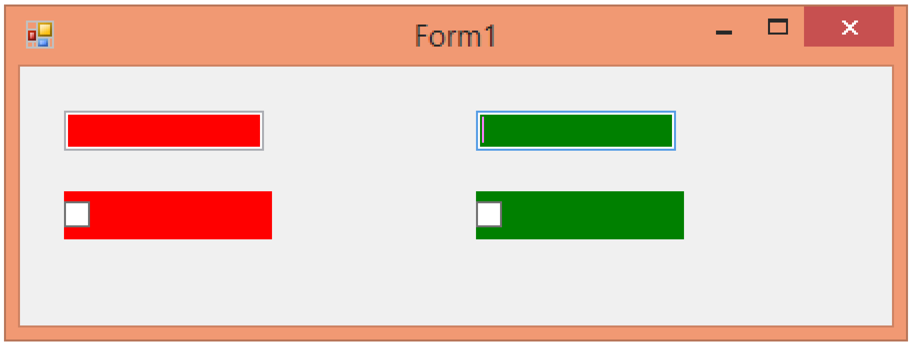


Figure 8: Panels with Different UIGenerators

This example also demonstrates the power of the Abstract Factory and Factory Method patterns.

The .NET Framework uses the Factory Method Pattern too, such as when working with database connections.

Code Listing 46: Factory Method in .NET

```
using (SqlConnection connection = new SqlConnection("connString"))
using (DbCommand cmd = connection.CreateCommand()) // Factory Method
{
    DbParameter param = cmd.CreateParameter(); // Factory Method
    //...
}
```

Singleton

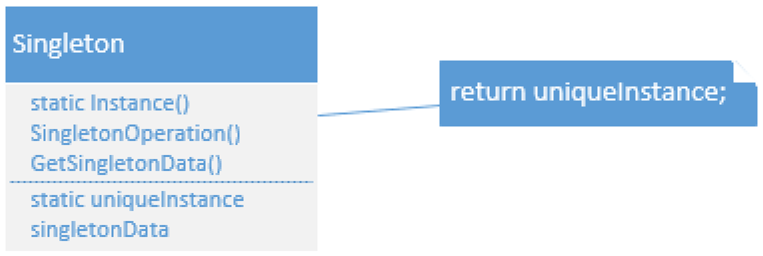


Figure 9: Class Diagram of the Singleton Pattern

The Singleton Pattern is pretty fun. It's easy, but often implemented incorrectly. It's loved, and it's hated. So why do I want to include it here? First, the Singleton limits the amount of created objects of a specific type to exactly one, which is pretty neat and can be necessary (examples of Singletons are your print spooler, the file system, etc.). Also, since it's often used incorrectly, I'd like to educate a little and hope to clear up when to use this. Last, since it's so often incorrectly implemented, we're going to do it correctly now, once and for all.

At the base of the Singleton is a `private` constructor. You read that right. When the constructor of a class is private no other class can create instances of this class. That may come in handy, as the class can now create instances of itself and keep track of those instances. That's a bit counter-intuitive, but with a static field completely possible.

Code Listing 47: The Start of a Singleton

```
public class SomeSingleton
{
    private static SomeSingleton instance;

    private SomeSingleton()
    { }
}
```

At this point no other class can create or reach an instance of this class. Next, we'll need another static method (or property) to give access to the instance of this class. We'll also need to create that instance.

Code Listing 48: The Complete Singleton

```
public class SomeSingleton
{
    private static SomeSingleton instance;

    private SomeSingleton()
    { }

    public static SomeSingleton Instance
    {
        get
        {
            if (instance == null)
            {
                instance = new SomeSingleton();
            }
            return instance;
        }
    }
}
```

And now the Singleton is complete. Indeed, this is the implementation we’ll often find in code bases (you’ll also often find Singletons with public constructors a.k.a. not Singletons at all).

So what is wrong with this Singleton? Nothing, if your application is completely single-threaded. When you’re going multi-threaded, this implementation can still result in multiple instances. This happens when two threads try to get the instance at the same time. Both threads will see that instance is null because neither thread has actually created the new instance yet, so we’ll need to lock this code for multiple threads. And as a small optimization, we won’t lock after the instance is created. Here is the full thread-safe Singleton:

Code Listing 49: The Thread-Safe Singleton

```
public class SomeSingleton
{
    private static SomeSingleton instance;
    private static object syncRoot = new object();

    private SomeSingleton()
    { }

    public static SomeSingleton Instance
    {
        get
        {
            if (instance == null)
            {
                lock(syncRoot)
                {
                    if (instance == null)
                    {
                        instance = new SomeSingleton();
                    }
                }
            }
            return instance;
        }
    }

    public void SomeMethod()
    {
        // Just an instance method like any other.
    }
}
```

The double null check may look weird, but remember that when two threads access Instance at the same time, both will see that instance is null, and one thread will be locked while the other creates the instance. After the locked thread is released, it should check for null again or it will create a second instance.

Other methods on the Singleton can now simply be public instance methods and will behave like any other. Just keep in mind that the state of a Singleton object is shared throughout the entire application because it's static!

So what's the problem with Singletons? They're used far too often. I've seen Singletons that, after a year, needed a second instance somewhere in the application! As it turned out, this Singleton wasn't a Singleton at all!

Second, Singletons can't be subclassed. Since the constructor is not accessible by any part of the application except the Singleton itself, not even subclasses can access it (meaning they won't be able to construct themselves).

Last, Singletons are hard to test. Suppose you're testing a part of the application that internally uses a Singleton. For a long time there was no way for you to mock the Singleton. It's now possible using newer testing frameworks that use code weaving and Aspect Oriented Programming (AOP), but it still doesn't feel completely right.

So when should you use a Singleton? Basically when you're absolutely sure your application will never need a second instance of an object (or when another instance may even be harmful for the system). It may also come in handy to provide one-point access to shared resources (like the print spooler). Logging is often noted as an acceptable use of the Singleton Pattern.

While many objects in .NET logically have only one instance (like **`Application`**), I am not aware of any Singleton implementations in .NET (that's not to say you can create instances of Application, the constructor is just internal instead of private). If a rather huge framework like .NET doesn't need Singletons, you should think very carefully if you need them or if you can solve your problem differently.

Structural Patterns

Structural Patterns are about making objects work together and structuring your code accordingly. Where Creational Patterns are about the creation of objects, structural patterns are about making objects work together, but without these objects knowing about each other's internals.

Adapter

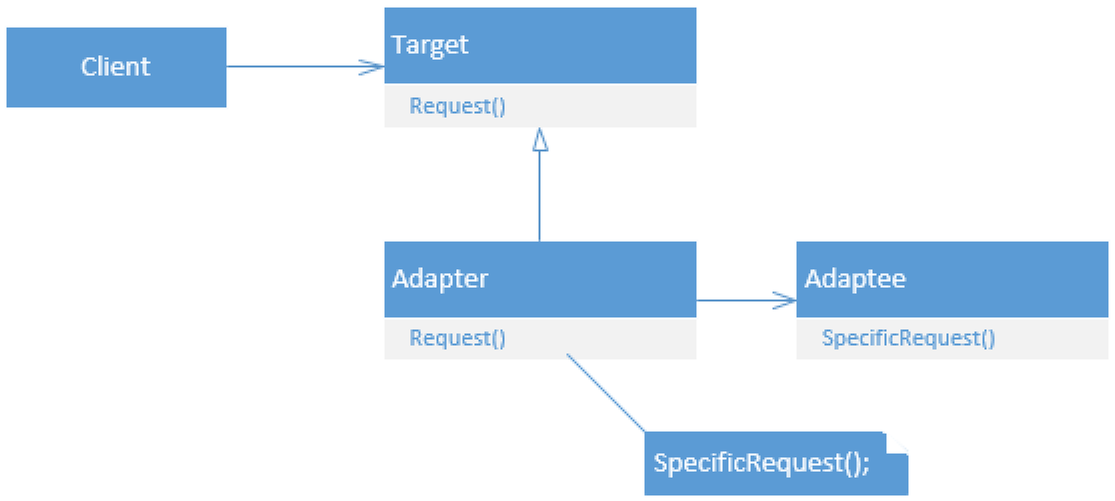


Figure 10: Class Diagram of the Adapter Pattern

An easy and well known Structural Pattern is the Adapter Pattern, or Wrapper. With the Adapter Pattern, we can make two classes work together even though they have incompatible interfaces. Suppose we're creating a data retrieval system, but the client hasn't decided on Oracle or SQL Server yet, or they might switch in the future. And let's say both databases come with some API with different classes and methods (of course, both already have compatible .NET API's). The following code is simplified, but does show the use of the Adapter:

Code Listing 50: External database API's

```
public class SqlServerApi
{
    public DataTable GetData()
    {
        return new DataTable();
    }

    public void SendData(DataTable data)
    {
        // ...
    }
}

public class OracleApi
{
    public object GetQuery()
    {
        return new object();
    }

    public void ExecQuery(object data)
    {
        // ...
    }
}
```

The problem here is that if we choose one API, we can't easily replace it with the other. **`SqlServerApi`** works with **`DataTables`** while Oracle works with plain objects, the methods have different names, and on and on. In the real world you'll find both interfaces use different classes, have many different methods (that the other doesn't have), and more. So let's create an interface that can tie these two incompatible API's together. The problem is what we'll do with the **`DataTables`**. Our interface can work with **`DataTables,`** and then we should make that work for Oracle, or we can work with **`objects`** and convert it to **`DataTables`** for SQL Server. A third option is to create a custom object which can be converted to both a plain **`object`** and a **`DataTable`**. For simplicity we'll work with **`object`**.

Code Listing 51: IDbApiAdapter

```
public interface IDbApiAdapter
{
    object GetData();
    void SendData(object data);
}
```

Now we can implement this for both SQL Server and for Oracle.

Code Listing 52: IDbApiAdapter Implementations

```
public class SqlServerApiAdapter : IDbApiAdapter
{
    private SqlServerApi api = new SqlServerApi();

    public object GetData()
    {
        DataTable table = api.GetData();
        object o = ConvertToObject(table);
        return o;
    }

    private object ConvertToObject(DataTable table)
    {
        // ...
        return null; // Placeholder.
    }

    public void SendData(object data)
    {
        DataTable table = ConvertToDataTable(data);
        api.SendData(table);
    }

    private DataTable ConvertToDataTable(object obj)
    {
        // ...
        return new DataTable();
    }
}

public class OracleApiAdapter : IDbApiAdapter
{
    private OracleApi api = new OracleApi();
    public object GetData()
    {
        return api.GetQuery();
    }

    public void SendData(object data)
    {
        api.ExecQuery(data);
    }
}
```

As you can see, converting from and to **`DataTables`** is now an “implementation detail.” We don’t really care how the **`SqlServerApiAdapter`** does it (probably using Reflection), as long as the public API is correct. You can now work with the **`IDbApiAdapter`** and leave the creation of a specific adapter to another part of the system (using a Factory or Dependency Injection).

Code Listing 53: Usage of IDbApiAdapter

```
static void Main(string[] args)
{
    IDbApiAdapter adapter = DbApiFactory();
    object o = adapter.GetData();
    adapter.SendData(new object());
}

static IDbApiAdapter DbApiFactory()
{
    return new SqlServerApiAdapter();
}
```

The Adapter Pattern is used in .NET when working with legacy COM objects. When referencing a COM object, .NET creates a wrapper (or adapter) that can be used within .NET to call the corresponding COM methods.

Composite

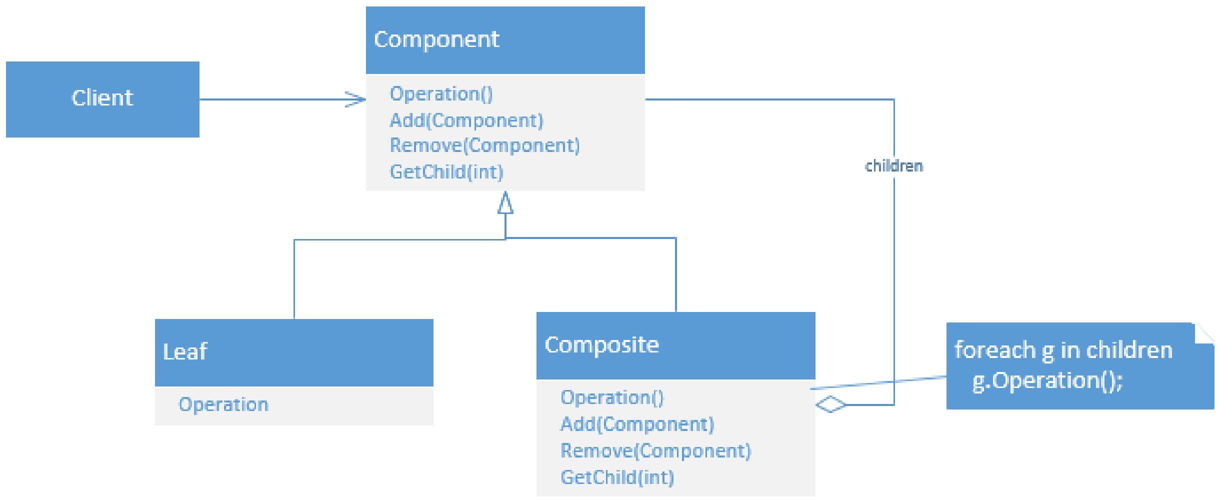


Figure 11: Class Diagram of the Composite Pattern

Composite is also one of the easier patterns. Let's say you have a collection of something, a **`List<Something>`** if you like. And maybe all these "somethings" have a collection of those same things too, child somethings. In that case we'll have a tree of somethings. Now something has some operation, and when that operation is performed the same operation should be performed on all child somethings. Sounds abstract, but let's say that something is a **`Control`** (like **`TextBox`**, **`CheckBox`**, **`DataGrid`**, etc.). So we have a **`Control`** which has child **`Controls`** and each child **`Control`** can have child **`Controls`** as well. And the operation is rendering the **`Control`**. So when we render our topmost **`Control`** all child **`Controls`** and children of those children should render too.

The next example may seem a bit farfetched, but it's not easy to create an example of the Composite Pattern, so bear with me. In the example, we're going to print sentences symbol by symbol. So our operation will be **`Print`**. We need to be able to print a single symbol, like "c" or "#" so let's create a class that can do just that.

Code Listing 54: IPrintable for our Composite

```
public interface IPrintable
{
    void Print();
}

public class Symbol : IPrintable
{
    private char symbol;

    public Symbol(char c)
    {
        symbol = c;
    }

    public void Print()
    {
        Console.Write(symbol);
    }
}
```

With that, we can create an **`IPrintable`** containing any single symbol. Let's create some additional **`IPrintables`** for our convenience (and for the example).

Code Listing 55: Additional IPrintable Implementations

```
public class LineBreak : IPrintable
{
    public void Print()
    {
        Console.WriteLine();
    }
}

public class Space : IPrintable
{
    public void Print()
    {
        Console.Write(" ");
    }
}
```

Now we need a composite class that can contain multiple symbols (or **`IPrintables`**), a **`Writing`**.

Code Listing 56: The (Composite) Writing Class

```
public class Writing : IPrintable
{

    private List<IPrintable> printables = new List<IPrintable>();

    public void Add(IPrintable printable)
    {
        printables.Add(printable);
    }

    public void Remove(IPrintable printable)
    {
        printables.Remove(printable);
    }

    public IPrintable GetPrintable(int index)
    {
        return printables[index];
    }

    public void Print()
    {
        foreach (IPrintable printable in printables)
        {
            printable.Print();
        }
    }
}
```

And that's it. We can now use these classes to compose words and sentences and print them on screen. In a simple Console application, the usage looks as follows:

Code Listing 57: Composite Usage

```
// Create the writing 'hello'.
Writing hello = new Writing();
hello.Add(new Symbol('h'));
hello.Add(new Symbol('e'));
hello.Add(new Symbol('l'));
hello.Add(new Symbol('l'));
hello.Add(new Symbol('o'));
// Add a line break.
LineBreak br = new LineBreak();
hello.Add(br);

// Print hello.
hello.Print();
// Remove the line break, so we can re-use hello.
hello.Remove(br);

// Create the writing 'bye'.
Writing bye = new Writing();
bye.Add(new Symbol('b'));
bye.Add(new Symbol('y'));
bye.Add(new Symbol('e'));

// Create a writing consisting of the writings
// hello and bye separated by a space.
Writing helloBye = new Writing();
helloBye.Add(hello);
helloBye.Add(new Space());
helloBye.Add(bye);

// Print all the printables.
helloBye.Print();

Console.ReadKey();
```

In .NET, this pattern is used, as mentioned, in WinForms when rendering **`Controls`** (and probably in WPF and ASP.NET as well).

Decorator

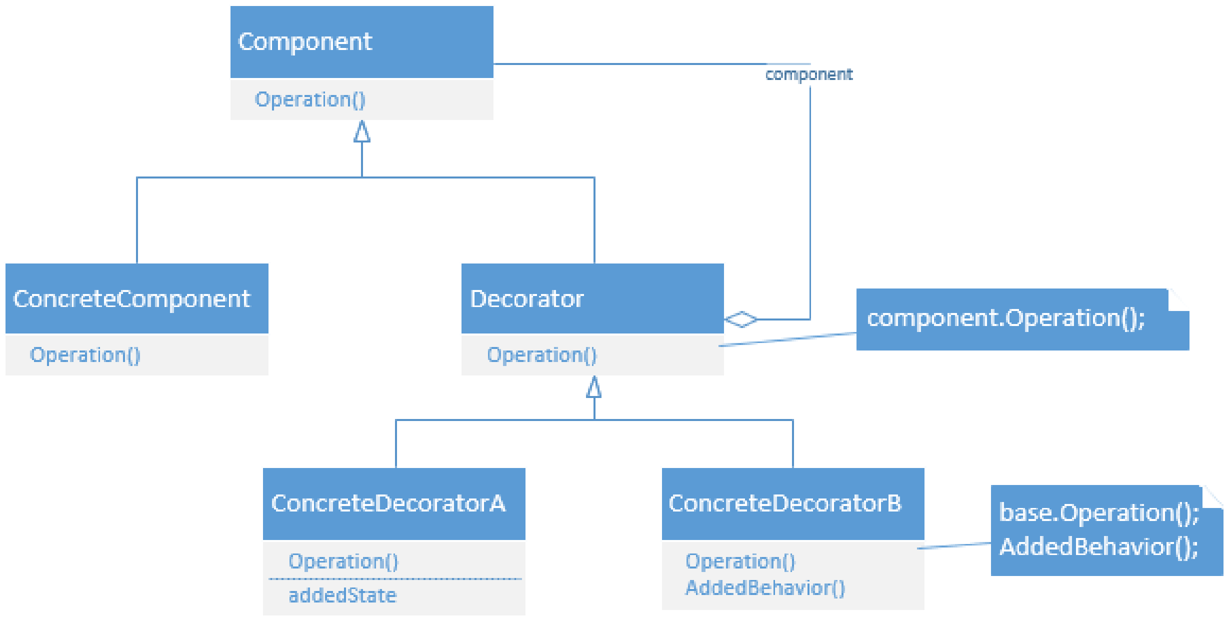


Figure 12: Class Diagram of the Decorator Pattern

The Decorator Pattern can be used to add functionality to an object dynamically. It's especially useful when one object can have multiple functionalities that aren't always used at the same time, like different representations of the same data. In that respect it offers a flexible alternative to inheritance.

Let's say we have some text message and we want to print and/or email and/or fax (yes, people still do that) and/or send it to an external system. Additionally, the message must always be saved to the database. First we'll need our **`Message`** class that contains the message we want to process and we can implement the default process (after all, we always need to send the message to the database).

Code Listing 58: The Message Class

```
public interface IMessage
{
    string Msg { get; set; }
    void Process();
}

public class Message : IMessage
{
    public string Msg { get; set; }
    public void Process()
    {
        Console.WriteLine(String.Format("Saved '{0}' to database.", Msg));
    }
}
```

We can now create a base decorator. Notice that we always need another **`IMessage`** passed to the constructor (or we'll get a **`NullReference`**, you may want to add a check there). The **`Msg`** property is simply delegated to the root **`IMessage`**.

Code Listing 59: A Base Decorator

```
public abstract class BaseMessageDecorator : IMessage
{
    private IMessage innerMessage;

    public BaseMessageDecorator(IMessage decorator)
    {
        innerMessage = decorator;
    }

    public virtual void Process()
    {
        innerMessage.Process();
    }

    public string Msg
    {
        get
        {
            return innerMessage.Msg;
        }

        set
        {
            innerMessage.Msg = value;
        }
    }
}
```

Now that we have our base decorator, we can create some sub decorators.

Code Listing 60: Sub Decorators

```
public class EmailDecorator : BaseMessageDecorator
{
    public EmailDecorator(IMessage decorator)
        : base(decorator)
    { }

    public override void Process()
    {
        base.Process();
        Console.WriteLine(String.Format("Sent '{0}' as email.", Msg));
    }
}

public class FaxDecorator : BaseMessageDecorator
{
    public FaxDecorator(IMessage decorator)
        : base(decorator)
    { }
    public override void Process()
    {
        base.Process();
        Console.WriteLine(String.Format("Sent '{0}' as fax.", Msg));
    }
}

public class ExternalSystemDecorator : BaseMessageDecorator
{
    public ExternalSystemDecorator(IMessage decorator)
        : base(decorator)
    { }

    public override void Process()
    {
        base.Process();
        Console.WriteLine(String.Format("Sent '{0}' to external system.", Msg));
    }
}
```

We can now process our message in many different ways; we can just save it to the database, we can save it and email or fax or send it to an external system; we can save, email and fax; save, email, and send it to the external system; or save, fax, and send it to the external system. The beauty is that we can easily add new functionality without breaking the already existing functionality.

In a simple Console application, the usage could be as follows:

Code Listing 61: Usage of the Decorators

```
IMessage msg = new Message();
msg.Msg = "Hello";

IMessage decorator = new EmailDecorator(
    new FaxDecorator(
        new ExternalSystemDecorator(msg)));
decorator.Process();

Console.WriteLine();
decorator = new EmailDecorator(msg);
decorator.Msg = "Bye";
decorator.Process();

Console.ReadKey();
```

In .NET we can find the Decorator Pattern when using streams. Many streams, like **`MemoryStream`**, **`BufferedStream`**, and **`FileStream`** offer the same base functionality (provided by the **`Stream`** base class), but they work differently internally. So the different streams decorate the **`Stream`** class. In the usage of these classes we can often find a Strategy Pattern (explained later).

Behavioral Patterns

Behavioral Patterns are about changing runtime behaviors of algorithms and classes. If Creational Patterns are about the creation of objects, and Structural Patterns about how the objects work together, then Behavioral Patterns are more about changing the implementation of those objects. With the three kinds of Patterns, you can build highly scalable and maintainable software that is neatly organized into small pieces of code.

Observer

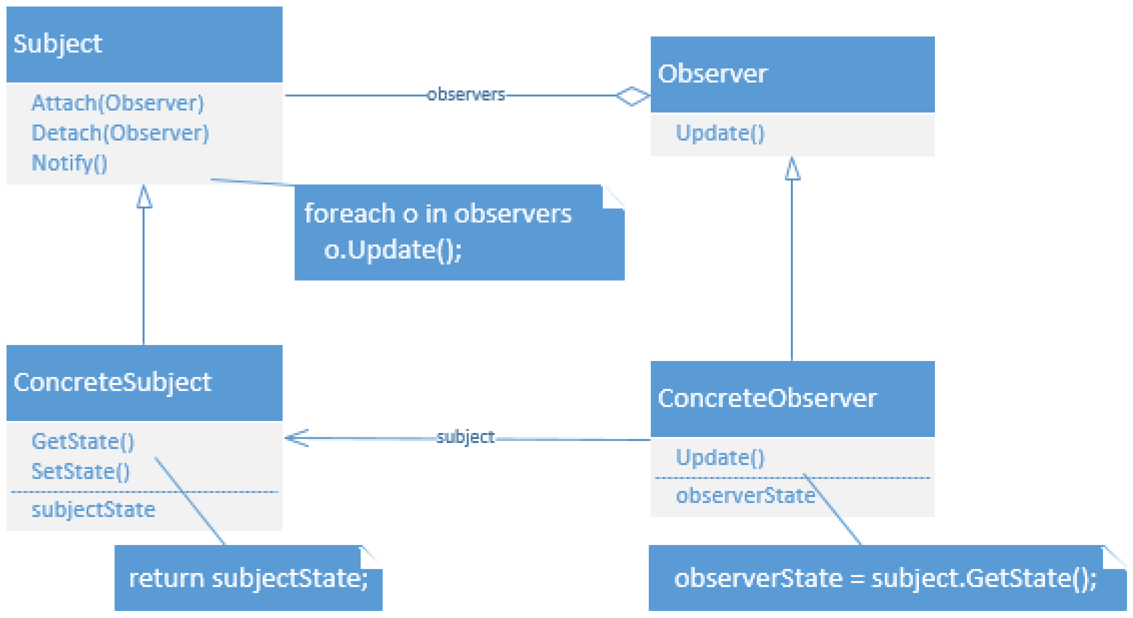


Figure 13: Class Diagram of the Observer Pattern

With the Observer Pattern it's possible for a class, a so-called observable, to notify other classes, observers, of changes. .NET events are implemented using this pattern.

A few quick notes regarding the Observer Pattern in .NET. First of all, because .NET already has events, it shouldn't be necessary to implement this pattern yourself. Let's assume for a moment that you still want to. You're probably going to start out with two interfaces: **`IObservable`** and **`IObserver`**. Then you find out .NET already has those interfaces (in a generic variant). Do not use them. I've seen the .NET interfaces being used in the Observer Pattern, but those interfaces are meant to be used primarily with Reactive Extensions (Rx). I can't explain Rx here, but the interfaces make little sense for the "traditional" Observer Pattern that I'm going to discuss here.

So let's start out by defining the **`IObservable`** and **`IObserver`** interfaces.

Code Listing 62: Observer Pattern Interfaces

```
public interface IObservable
{
    void Attach(IObserver observer);
    void Detach(IObserver observer);
    void Notify();
}

public interface IObserver
{
    void Update(IObservable observable);
}
```

Now let's implement the **`IObservable`**. We're going to create a product stock, and every time the stock changes we're going to notify observers (also called listeners). The code is a bit lengthy, mostly due to the properties at the bottom. Notice that these properties have nothing to do with the Observer Pattern, they're simply necessary to identify the stock.

Code Listing 63: IObservable Implementation

```
public class Stock : IObservable
{
    private List<IObserver> observers = new List<IObserver>();
    private string productName;
    private int noOfItemsInStock;

    public Stock(string name)
    {
        productName = name;
    }

    public void Attach(IObserver observer)
    {
        observers.Add(observer);
    }

    public void Detach(IObserver observer)
    {
        observers.Remove(observer);
    }

    public void Notify()
    {
        foreach (IObserver observer in observers)
        {
            observer.Update(this);
        }
    }

    public string ProductName
    {
        get
        {
            return productName;
        }
    }

    public int NoOfItemsInStock
    {
        get
        {
            return noOfItemsInStock;
        }
        set
        {
            noOfItemsInStock = value;
            Notify();
        }
    }
}
```

And now we can create some observers.

Code Listing 64: Observers

```
public class Seller : IObserver
{
    public void Update(IObservable observable)
    {
        Console.WriteLine("Seller was notified about the stock change.");
    }
}

public class Buyer : IObserver
{
    public void Update(IObservable observable)
    {
        Console.WriteLine("Buyer was notified about the stock change.");
    }
}
```

The usage in a simple Console application is now pretty easy.

Code Listing 65: Usage of the Observer

```
Stock stock = new Stock("Cheese");
stock.NoOfItemsInStock = 10;

// Register observers.
Seller seller = new Seller();
stock.Attach(seller);
Buyer buyer = new Buyer();
stock.Attach(buyer);

stock.NoOfItemsInStock = 5;
Console.ReadKey();
```

You'll find a big caveat with this approach though. The buyer and seller get notified, but they don't know about what. It's just some **`IObservable`**. There are a few options here. First we can simply cast the **`IObservable`** to **`Stock`**. For example, the **`Seller`** class would now look as follows:

```
public class Seller : IObserver
{
    public void Update(IObservable observable)
    {
        Stock stock = (Stock)observable;
        Console.WriteLine(String.Format("Seller was notified about the stock change of 
{0} to {1} items.",
            stock.ProductName, stock.NoOfItemsInStock));
    }
}
```

This approach is taken in a WinForm event where every **`Control`** event has the signature of **`(object sender, EventArgs e)`**. The **`sender`** can be any type of **`Control`** and should be cast if you'd like to do anything useful with the sender (the initiator of the event).

A second approach is to make your **`IObservable`** less generic (or more specific). In that case, **`IObservable`** would get the **`ProductName`** and **`NoOfItemsInStock`** properties (and you might want to rename it to **`IStock`**).

Code Listing 66: A More Specific IObservable

```
public interface IObservable
{
    void Attach(IObserver observer);
    void Detach(IObserver observer);
    void Notify();
    string ProductName { get; }
    int NoOfItemsInStock { get; set; }
}

public class Seller : IObserver
{
    public void Update(IObservable observable)
    {
        Console.WriteLine(String.Format("Seller was notified about the stock change of 
{0} to {1} items.",
            observable.ProductName, observable.NoOfItemsInStock));
    }
}
```

Of course, the problem with this approach is that you can't reuse **`IObservable`** for other classes.

So last, but not least, you might want to use Generics. This should solve all your problems. Remember that **`IObserver<T>`** and **`IObservable<T>`** already exist for Reactive Extensions, so we'll have to use another name. I've used **`INotifier`** and **`IListener`**. Not much changes, but the implementation is slightly more difficult.

Code Listing 67: Generic IObservable

```
public interface INotifier<T>
{
    void Attach(IListener<T> observer);
    void Detach(IListener<T> observer);
    void Notify();
}

public interface IListener<T>
```

```
{
    void Update(T observable);
}

public class Stock : INotifier<Stock>
{
    private List<IListener<Stock>> observers = new List<IListener<Stock>>();
    private string productName;
    private int noOfItemsInStock;

    public Stock(string name)
    {
        productName = name;
    }

    public void Attach(IListener<Stock> observer)
    {
        observers.Add(observer);
    }

    public void Detach(IListener<Stock> observer)
    {
        observers.Remove(observer);
    }

    public void Notify()
    {
        foreach (IListener<Stock> observer in observers)
        {
            observer.Update(this);
        }
    }

    public string ProductName
    {
        get
        {
            return productName;
        }
    }

    public int NoOfItemsInStock
    {
        get
        {
            return noOfItemsInStock;
        }
        set
        {
            noOfItemsInStock = value;
            Notify();
        }
    }
}

public class Seller : IListener<Stock>
{
    public void Update(Stock observable)
    {
        Console.WriteLine(String.Format("Seller was notified about the stock change of
{0} to {1} items.",
            observable.ProductName, observable.NoOfItemsInStock));
    }
}
```

Strategy

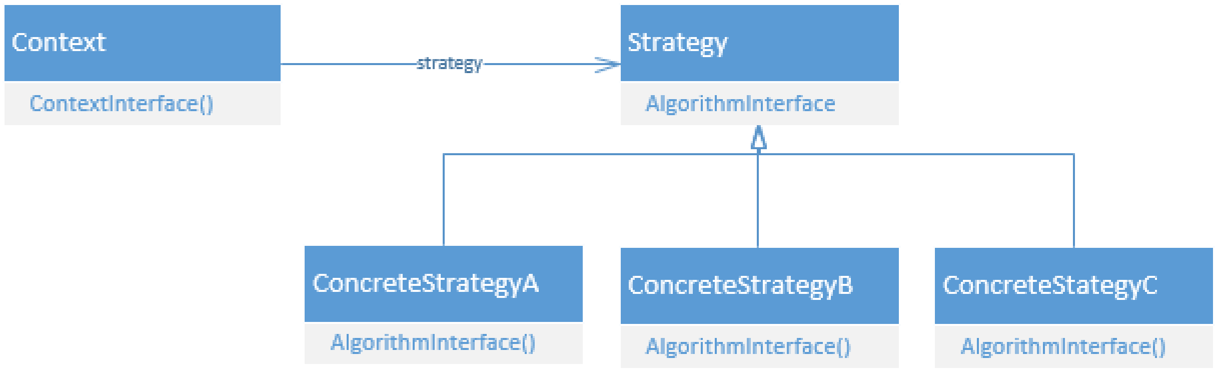


Figure 14: Class Diagram of the Strategy Pattern

The Strategy Pattern allows you to encapsulate a family of algorithms and use them interchangeably. You can think of many applications for the Strategy Pattern, for example a video game where units on a battlefield all behave the same in many aspects, such as moving and rendering, but differ by attack strategy. One unit may be offensive while another is defensive. The offensive or defensive algorithm can be injected into each unit.

Another example is the sorting of a list. Depending on the size of the list you might want to use another sorting algorithm, but you don't want to hard code each algorithm into the list class.

So let's make a little game. It's a simple Console application and we'll just be pressing any button which will move our units and which may result in an enemy encounter (keep pressing buttons or press escape to quit). Each unit reacts differently to an enemy encounter. So first let's define a generic **`Unit`** class.

Code Listing 68: The Strategy Pattern

```
public interface IUnitBehavior
{
    void ReactToOpponent();
}

public class Unit
{
    private string name;
    private IUnitBehavior behavior;

    public Unit(string name, IUnitBehavior behavior)
    {
        this.behavior = behavior;
        this.name = name;
    }

    public void Render()
    {
        Console.WriteLine(@"\o/ ");
        Console.WriteLine(@" O ");
        Console.WriteLine(@"/ \");
    }

    public void Move()
    {
        Console.WriteLine(String.Format("{0} moves...", name));
    }

    public void ReactToOpponent()
    {
        behavior.ReactToOpponent();
    }

}
```

So as you can see the **`Render`** and **`Move`** methods are the same for each **`Unit`**. **`ReactToOpponent`**, however, is injected through the constructor. The **`IUnitBehavior`** interface defines the ultimate strategy to use. So let's define some strategies.

Code Listing 69: Some Strategies

```
public class WarriorBehavior : IUnitBehavior
{
    public void ReactToOpponent()
    {
        Console.WriteLine("\"ATTACK!!!\"");
    }
}

public class DefenderBehavior : IUnitBehavior
{
    public void ReactToOpponent()
    {
        Console.WriteLine("\"Hold the line!\"");
    }
}
```

And now let's glue the **`Unit`** and **`IUnitBehavior`** together.

Code Listing 70: The Final Game

```
Unit warrior = new Unit("Warrior", new WarriorBehavior());
Unit defender = new Unit("Defender", new DefenderBehavior());

List<Unit> units = new List<Unit>();
units.Add(warrior);
units.Add(defender);
```

```
foreach (Unit unit in units)
{
    unit.Render();
    Console.WriteLine();
}

Console.WriteLine("Your troops are on an important mission!");
Console.WriteLine("Press Escape to quit or any other key to continue.");
Console.WriteLine();
Random rnd = new Random();
while (Console.ReadKey().Key != ConsoleKey.Escape)
{
    foreach (Unit unit in units)
    {
        unit.Move();
    }
    Console.WriteLine();

    // Returns 0, 1, or 2.
    if (rnd.Next(3) == 0)
    {
        Console.WriteLine("The enemy attacks!");
        foreach (Unit unit in units)
        {
            unit.ReactToOpponent();
        }
    }
    else
    {
        Console.WriteLine("Nothing happened...");
    }
    Console.WriteLine();
}
```

That's quite a bit of code, but it's really the first two lines that matter. We're defining two **`Units`**, but with another **`IUnitBehavior`**. When an enemy attacks (chance 1 in 3) you'll see that each **`Unit`** behaves according to its strategy. It also becomes easy to add another strategy.

Code Listing 71: Adding a New Strategy

```
// ...
Unit priest = new Unit("Priest", new PriestBehavior());
// ...
units.Add(priest);
// ...

public class PriestBehavior : IUnitBehavior
{
    public void ReactToOpponent()
    {
        Console.WriteLine("\"Heal our troops!\"");
    }
}
```

The .NET Framework makes use of the Strategy Pattern as well, such as when calling **`List<T>.Sort()`**. You can pass in an optional **`IComparer<T>`**. Let's check that out as it's another good example. We're going to sort people by their first and last names.

Code Listing 72: Person Class and IComparers

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class FirstNameSorter : IComparer<Person>
{
    public int Compare(Person x, Person y)
    {
        return x.FirstName.CompareTo(y.FirstName);
    }
}

public class LastNameSorter : IComparer<Person>
{
    public int Compare(Person x, Person y)
    {
        return x.LastName.CompareTo(y.LastName);
    }
}
```

We can now create a list, put some people in it, and sort them by first or last name.

Code Listing 73: Sorting a List with Strategy

```
Person bill = new Person { FirstName = "Bill", LastName = "Gates" };
Person steve = new Person { FirstName = "Steve", LastName = "Ballmer" };
Person satya = new Person { FirstName = "Satya", LastName = "Nadella" };

List<Person> ceos = new List<Person>();
ceos.Add(bill);
ceos.Add(steve);
ceos.Add(satya);

ceos.Sort(new FirstNameSorter());
foreach (Person p in ceos)
{
    Console.WriteLine(String.Format("{0} {1}", p.FirstName, p.LastName));
}
Console.WriteLine();

ceos.Sort(new LastNameSorter());
foreach (Person p in ceos)
{
    Console.WriteLine(String.Format("{0}, {1}", p.LastName, p.FirstName));
}
Console.ReadKey();
```

Template Method

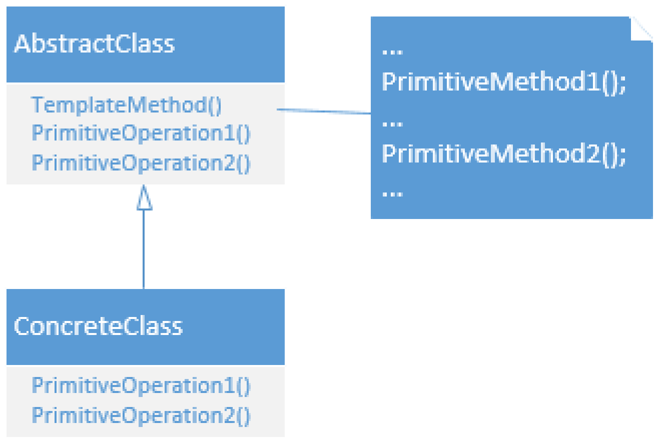


Figure 15: Class Diagram of the Template Method

The Template Method Pattern is really very simple. In fact, it's not much more than simple inheriting and method overriding. It's the inheritance variant of the Strategy Pattern (Inheritance vs. Delegation).

Let's say we have a general pattern of connecting to a database, constructing a query, executing the query, retrieving the results, and finally closing the connection. We can quite literally write down this example in code without worrying about the details.

Code Listing 74: Template Method

```
public abstract class BaseDataRetriever
{
    public DataTable RetrieveData()
    {
        Connect();
        string query = GetQuery();
        DataTable result = ExecQuery(query);
        Close();
        return result;
    }

    protected abstract void Connect();
    protected abstract string GetQuery();
    protected abstract DataTable ExecQuery(string query);
    protected abstract void Close();
}
```

Let's say we need to retrieve the Product table from a SQL Server database. We can inherit the **`BaseDataRetriever`** and fill in the blanks.

Code Listing 75: Inheritance of the Template Class

```
public class ProductRetriever : BaseDataRetriever
{
    private string connString;
    private SqlConnection connection;

    public ProductRetriever(string connString)
    {
        this.connString = connString;
    }

    protected override void Connect()
    {
        connection = new SqlConnection(connString);
    }

    protected override string GetQuery()
    {
        return "SELECT Id, Name, Price FROM Product";
    }

    protected override DataTable ExecQuery(string query)
    {
        using (SqlDataAdapter adapter = new SqlDataAdapter(query, connection))
        {
            DataTable result = new DataTable();
            adapter.Fill(result);
            return result;
        }
    }

    protected override void Close()
    {
        connection.Dispose();
        connection = null;
    }
}
```

Likewise, we can implement it for ODBC, Oracle, or what have you.

It would make sense to add another level of depth, so we can implement this for SQL Server, but not make it query specific. Just rename **`ProductRetriever`** to **`SqlDataRetriever`**, make it abstract, and remove the **`GetQuery`** method. Now the **`ProductRetriever`** would look as follows:

Code Listing 76: ProductRetriever

```
public class ProductRetriever : SqlDataRetriever
{
    public ProductDataRetriever(string connString)
        : base(connString)
    { }

    protected override string GetQuery()
    {
        return "SELECT Id, Name, Price FROM Product";
    }
}
```

And it's now very easy to create a **`CustomerRetriever`**.

Code Listing 77: CustomerRetriever

```
public class CustomerRetriever : SqlDataRetriever
{
    public CustomerRetriever(string connString)
        : base(connString)
    { }

    protected override string GetQuery()
    {
        return "SELECT Id, FirstName, LastName, Address FROM Customer";
    }
}
```

The usage of the **`CustomerRetriever`** class (or any **`BaseDataRetriever`** class) is as straight forward as it gets.

Code Listing 78: Usage of the Template Method

```
CustomerRetriever retriever = new CustomerRetriever("connString");
DataTable table = retriever.RetrieveData();
// Do stuff with the data.
```

In .NET we find the Template Method frequently. Basically, every method that you can override is a Template Method. One such example is the **`Collection<T>`** class, which was meant as a base class for custom collections.

Code Listing 79: A Custom Collection

```
public class UniqueCollection<T> : Collection<T>
{
    // Only insert or set items if
    // the item is not yet in the collection.
    protected override void InsertItem(int index, T item)
    {
        if (!this.Contains(item))
        {
            base.InsertItem(index, item);
        }
    }

    protected override void SetItem(int index, T item)
    {
        if (!this.Contains(item))
        {
            base.SetItem(index, item);
        }
    }
}
```

More Patterns

The patterns discussed so far are “classic” patterns that were systematically described in the book *Design Patterns: Elements of Reusable Object-Oriented Software*. However, there are many more Design Patterns. Many of these are discussed in the book *Patterns of Enterprise Application Architecture* by Martin Fowler[3] (in Java). The patterns discussed in Fowler’s book cover different areas of your code base, such as patterns specific for you domain logic, communication with your data source, concurrency and web session. I will discuss a few of these patterns that are widely known and used.

Martin Fowler uses a lot of UML sequence diagrams. Like UML class diagrams, sequence diagrams are outside the scope of the book, but I will add them for reference and visual explanation.

Lazy Load

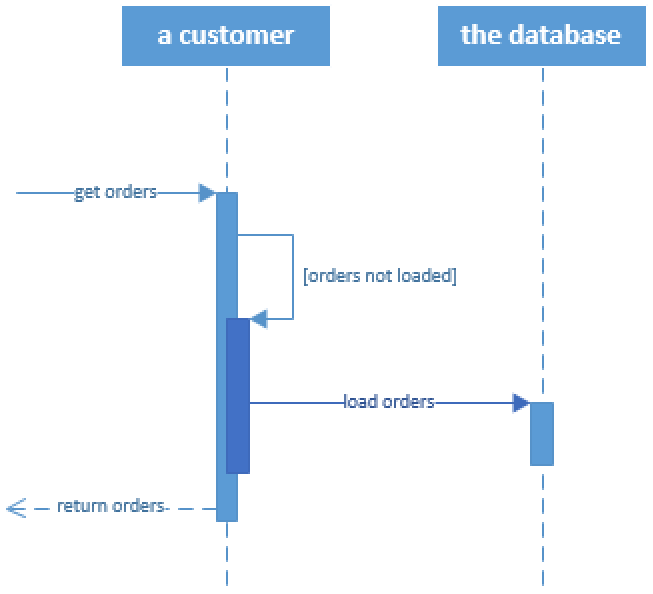


Figure 16: Sequence Diagram of the Lazy Load Pattern

The Lazy Load Pattern is about not getting data until you actually use it. It comes in four flavors: lazy initialization, virtual proxy, value holder, and ghost. We’ll look at lazy initialization only, as it’s the most common and discussing all of them takes up too much space. I also want to discuss this one because it has become more and more relevant and widely used. Functional languages, such as Haskell, have a default lazy loading strategy for everything. Such practices are becoming more common in .NET as well.

[3] Martin Fowler - *Patterns of Enterprise Application Architecture*

We have actually already seen the Lazy Loading Pattern when we implemented the Singleton Pattern. The Singleton instance was not created until the object was actually used. Let's take a look at the (simple) implementation again.

Code Listing 80: Lazy Initialization

```
public class SomeSingleton
{
    private static SomeSingleton instance;

    private SomeSingleton()
    { }

    public static SomeSingleton Instance
    {
        get
        {
            if (instance == null)
            {
                instance = new SomeSingleton();
            }
            return instance;
        }
    }
}
```

If **`SomeSingleton.Instance`** is not referenced anywhere in your code, then **`SomeSingleton`** will never be initialized. This is what's called lazy initialization. It's especially useful when an object takes a long time to load and you don't want to execute the code unless you really have to.

The .NET Framework has a **`Lazy<T>`** class for lazy loading any class. Let's say we have a class that takes five seconds to initialize.

Code Listing 81: A heavy initialization

```
public class Something
{
    public Something()
    {
        // Fake some heavy computation.
        Thread.Sleep(5000);
    }

    public void PrintUse()
    {
        Console.WriteLine("Something is used.");
    }
}
```

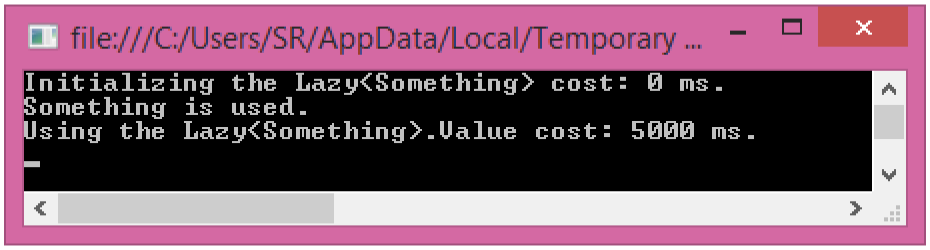
By using a **`Lazy<T>`** wrapper we won't actually need to create the instance until we use it.

Code Listing 82: Usage of Lazy<T>

```
Stopwatch sw = new Stopwatch();
sw.Start();
Lazy<Something> lazy = new Lazy<Something>();
sw.Stop();
Console.WriteLine(String.Format(
    "Initializing the Lazy<Something> cost: {0} ms.",
    sw.ElapsedMilliseconds));

sw.Restart();
lazy.Value.PrintUse();
sw.Stop();
Console.WriteLine(String.Format(
    "Using the Lazy<Something>.Value cost: {0} ms.",
    sw.ElapsedMilliseconds));
Console.ReadKey();
```

The result should not be surprising.



```
Initializing the Lazy<Something> cost: 0 ms.
Something is used.
Using the Lazy<Something>.Value cost: 5000 ms.
```

Figure 17: Result of Using the Lazy<T>

We can use complex initialization, too, using a lambda method.

Code Listing 83: Lazy<T> with Complex Initialization

```
// ...
Lazy<Something> lazy = new Lazy<Something>(() => new Something(new object()));
// ...

public class Something
{
    public Something(object someParam)
    {
        // ...
```

The .NET Framework uses lazy loading extensively with the Entity Framework, an Object Relation Mapper (ORM). The Entity Framework maps database tables to classes within .NET and can update your database using these classes. Now imagine you have a **SalesOrder** table; each **SalesOrder** has **SalesOrderLines**, which is another table. You have classes in C# with the same name and attributes. Additionally, your C# class has a property, **SalesOrderLines,** which is in fact a collection of **SalesOrderLines**. When loading your **SalesOrders,** your **SalesOrderLines** will be empty until you actually request the **SalesOrderLines** of a **SalesOrder**. This is not necessarily faster if you need ALL **SalesOrderLines**, but if you only need some of them, it can be pretty fast and convenient. See the sequence diagram at the beginning of this section for a visualization.

Model View Controller

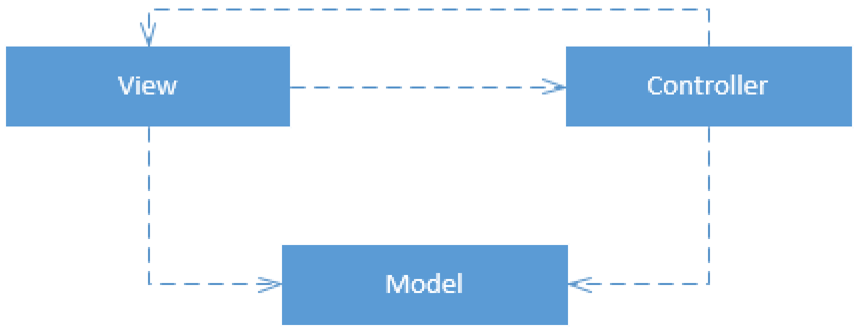


Figure 18: Class Diagram of the MVC Pattern

The Design Patterns book already mentions the Model View Controller pattern, MVC for short, used in Smalltalk-80, but only to say this pattern makes use of the Observer, Composition, and Strategy Design Patterns. There is more to the MVC pattern though. It makes use of three components: the Model, the View, and the Controller [insert Ennio Morricone soundtrack here]. The Model contains some information on the domain, such as a product, a customer, or a sales order. The Model is strictly non-UI. The UI information is in the View object. If this is a web page, the view might contain HTML. Or maybe it contains some UI widgets that are displayed on-screen. The third component, the Controller, glues these two together. It handles requests from the View, retrieves and manipulates data that is then put in a Model, which is returned to the View, which can use the Model for display on-screen.

The entire MVC model is about separation of concerns (SoC). Indeed, using MVC it becomes easy to swap one part of the system in favor of something else. In web development you could swap the Views or have multiple Views to represent the same data.

The Model View Controller pattern is so prevalent today that Microsoft created an entire framework around it and named it after the pattern, ASP.NET MVC. The best way to check this out is to simply start up Visual Studio and create a new ASP.NET MVC project (ASP.NET Web Application). You'll get a whole lot of files and folders (an often heard complaint about ASP.NET MVC), but the classes we'll be interested in are Controllers, Models and Views folders. The **`HomeController`**, for example, has three methods, **`Index`**, **`About`**, and **`Contact`**. All of them return a View without a Model. The **`AccountController`** has some methods that return a View and pass in a Model, but they're a bit complex to discuss here (this isn't a book about ASP.NET MVC after all). If you look at the Models, for example the classes in **`AccountViewModels`**, you'll see that these are just classes with some properties and attributes. The Models are initialized in the **`AccountController`** and used in the Account Views (which is composed of many cshtml files). For some simpler Views check out the Home Views, About.cshtml, Contact.cshtml and Index.cshtml.

As a simple example, I'm going to create a new ASP.NET MVC project, rename it to "MyMvcApp", change the authentication type to "No Authentication" and create a very simple, static website. First, throw away all Views, except Home/Index.cshtml. Now I want to show some data in my View, so I'm going to create a Model, the **`PersonModel`**.

Code Listing 84: The Model

```
public class PersonModel
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

Next, I’m going to change the **`HomeController`** so it will pass a **`PersonModel`** to the View.

Code Listing 85: The Controller

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        PersonModel model = new PersonModel();
        model.FirstName = "Bill";
        model.LastName = "Gates";
        return View(model);
    }
}
```

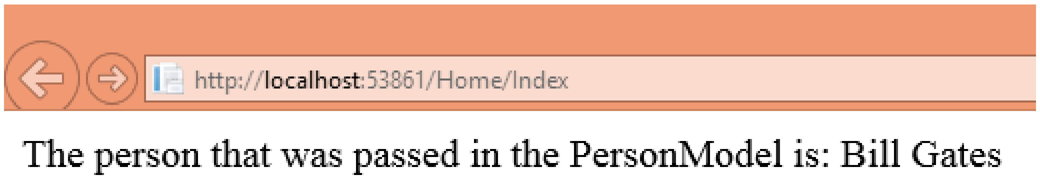
And finally I’m going to change my Home/Index View (which isn’t exactly valid HTML, but it’ll do) so it will show the data in the Model.

Code Listing 86: The View

```
@using MyMvcApp.Models
@model PersonModel

<p>The person that was passed in the PersonModel is: @Model.FirstName
@Model.LastName</p>
```

And that will show the following web page:



The person that was passed in the PersonModel is: Bill Gates

Figure 19: The Resulting View

Now if you change the name in the Controller to something else it will be reflected in the View. You don’t have to change the Model or the View. To recap, a user requests a View from the Controller. The Controller (optionally) creates a Model, passes it to the View, and returns the View. The View displays the data in the Model on the screen.

Two Design Patterns related to MVC are Model View ViewModel (MVVM) and Model View Presenter (MVP).

MVP is a pattern for the user interface. The Presenter sits between the Model and the View and formats data so the View can use it.

In MVVM, the ViewModel is an abstraction of the View that the View uses for display. Because of the abstraction of the View, two-way binding is made possible (updates in the Model are directly presented in the View and updates in the View are directly pushed to the Model). MVVM was created for Microsoft's Windows Presentation Foundation (WPF), but can now also be found in many JavaScript frameworks like KnockoutJS and AngularJS.
Together MVC, MVP, and MVVM are referred to as the MV* Patterns. Note that they are not mutually exclusive.

Repository

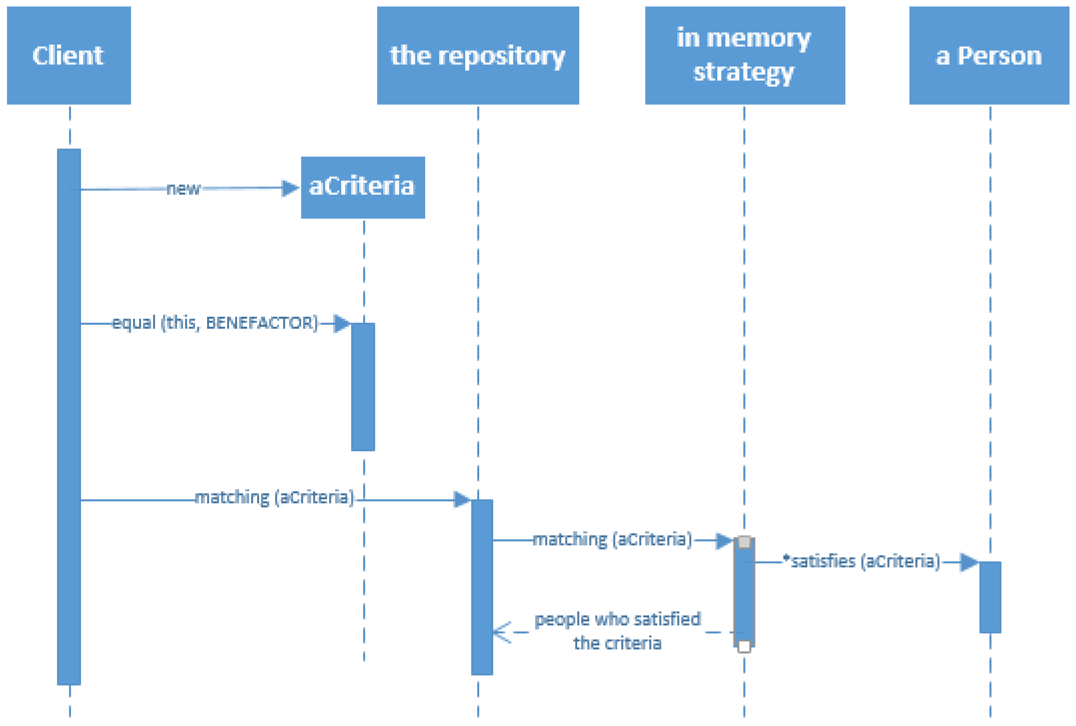


Figure 20: Sequence Diagram of the Repository Pattern

The Repository Pattern is a difficult pattern that, luckily, you don't have to implement yourself. It makes use of other patterns, the Data Mapper and the Query Object. A Data Mapper is a piece of code that maps database records to C# domain objects. The database and the domain objects are unaware of each other. A Query Object is a C# object that behaves like a database WHERE clause and which can be passed to the Data Mapper ("aCriteria" in the diagram).
The Repository, finally, is a collection-like interface that can retrieve domain objects and sits between your domain and data mapping layers. Once again, the benefit to be had is separation of concerns. Your domain objects don't know about your database and can be reused for other purposes. Your database layer can be swapped (for example, going from Oracle to SQL Server) and you'll only have to change your Data Mapper. Your Data Mapper, Query Object, and Repository can also be reused in other projects.

As much as I'd like building a Data Mapper, Query Object, and Repository, we're not going to do that. That costs a lot of time and can be a book on its own. Instead, I'm going to create a simple Console application, save it, add an ADO.NET Entity Model to my project, and connect to some database where I have a Person table. The Entity Framework will generate the **`Person`** class for me (I'm not taking the code first approach).

Code Listing 87: Generated Person Class

```
public partial class Person
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public Nullable<DateTime> DateOfBirth { get; set; }
}
```

It will also generate the Repository, which inherits from **`DbContext`**, and which gives access to my **`Person`** objects like they were just in memory C# objects.

Code Listing 88: Generated DbContext

```
public partial class ApplicationEntities : DbContext
{
    public ApplicationEntities()
        : base("name=ApplicationEntities")
    {
    }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        throw new UnintentionalCodeFirstException();
    }

    public virtual DbSet<Person> People { get; set; }
}
```

And here comes the beauty. The usage of the Repository Pattern (when done right) is just really easy.

Code Listing 89: Usage of the Repository

```
using (ApplicationEntities context = new ApplicationEntities())
{
    // Construct a query.
    // context.People is the Repository.
    // The Query Object (an Expression) is constructed using .Where(lambda).
    // The Data Mapper is not visible, but is used by the Entity Framework internally.
    IQueryable<Person> query = context.People.Where(p => p.FirstName.Contains("Sand"));
    // ToList() will actually call the database.
    List<Person> people = query.ToList();
}
```

Personally I'm a fan of the whole "lambda creates **`Expression`** (Query Object)" philosophy, but it does obscure the pattern a bit. Other APIs that I've used have a more explicit Query Object.

Code Listing 90: Explicit Query Object

```
List<Person> people = SomeApi.People.Select(Query.Contains("FirstName", "Sand"));
```

It still beats writing SQL!

A word of caution when using a Repository: everything may look like plain C#, but the dark truth is that it isn't. Ultimately, some query will be sent to your database, and if you're not careful it's not going to be pretty. The Entity Framework (and other ORM's) are capable of joining, aggregating, using inner queries, what have you. Some SQL code will have to be generated and if you don't know what you're doing it will be ugly and slow. Sometimes it's just a lot quicker to write some SQL and be done with it!

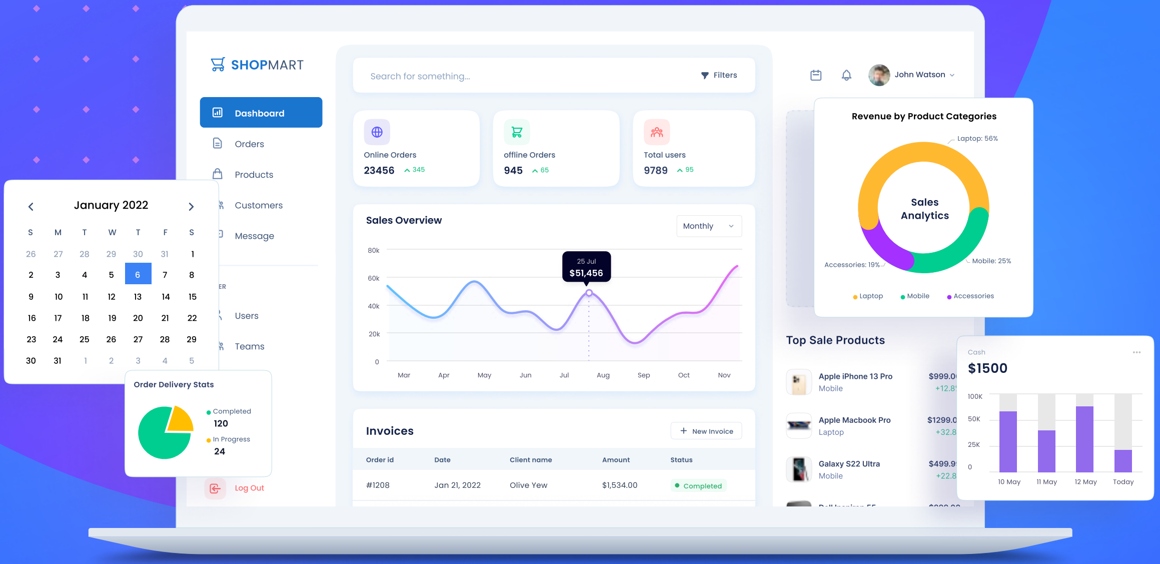
The Takeaway

There are many more patterns. Some patterns can be generally applied in any (object-oriented) language, while others are language specific. Some patterns are even specific to a certain programming paradigm. For example, JavaScript, which has no access modifiers, has Design Patterns, like Module, to still be able to hide object internals. When you're building a multi-threaded application, you might want to use patterns specifically designed to simplify multi-threaded code. To discuss all these patterns, you'd need many books that aren't exactly succinct. Design Patterns, like most things in life, have fans and opponents. While it's true that Design Patterns do not guarantee beautiful software, and may even lead to overly complex software, it's also important that you know the tools and methods that are available to you. Design Patterns are in no way the holy grail of programming, but, when applied correctly, can help you in creating robust and maintainable software.

The .NET Framework makes use of many other Design Patterns that aren't discussed here, for example; the Iterator Pattern, with **`IEnumerable`** and **`IEnumerator`**, making it possible to loop through collections; the Builder Pattern, for constructing complex objects, see for example the **`ConnectionStringBuilder`**; the Visitor Pattern in the **`ExpressionVisitor`** (when working with **`Expressions`**/the Entity Framework); the Proxy Pattern when working with Windows Communication Foundation (WCF); and the Unit of Work Pattern to gracefully handle database transactions in the Entity Framework.

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

IBM SIEMENS intel VISA

Syncfusion

Chapter 5 General Responsibility Assignment Software Patterns or Principles (GRASP)

One set of patterns that you aren't going to see a lot are the GRASP patterns, General Responsibility Assignment Software Patterns or Principles (I'm pretty sure the abbreviation came first). They are not quite patterns like Design Patterns, they are more like advice that should help you solve common programming problems. I'm not sure why GRASP is not as popular as SOLID or Design Patterns, as the principles are no less important and every developer should know them. Craig Larman discusses them in his book *Applying UML and Patterns – An Introduction to Object-Oriented Analysis and Design and Iterative Development*[4]. There are a total of nine GRASP principles, and some of them overlap with what was already discussed. In this chapter I'll discuss the GRASP Patterns that have no overlap with what we previously discussed. The ones that overlap are Controller (like we've seen in MVC), Polymorphism (one of the three pillars of OOP), Indirection (another word for Dependency Inversion), and Protected Variations (another form of Dependency Inversion and Encapsulation).

Creator

One of the first problems you will encounter with any OO application is that you need to create an instance of some object, but you're not sure what other object is going to create it. In some cases this is obvious, such as when working with certain Design Patterns such as Factory Method or Singleton. Other times it's not as obvious. At the start of this book, we saw a small example with a **`Car`** and an **`Engine`** class (Inheritance vs. Composition).

Code Listing 91: Car and Engine

```
public class Engine
{
    // ...
}

public class Car
{
    private Engine engine = new Engine();
    // ...
}
```

[4] Craig Larman - *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*

A **`Car`** has an **`Engine`**, so it should obviously create an **`Engine`**, right? Well, maybe. There are a few pointers for whether one class creates another. If some class contains or aggregates another class, closely uses another class, or has the initializing data for another class (see also Information Expert in the next section), it might be a good idea to have that class create another class. A **`Car`** does contain an **`Engine`**, but it does not aggregate it. A **`Car`** closely uses an **`Engine`**. Does the **`Car`** also have the initializing data for an **`Engine`**? Who knows? It depends on your design. In this case I would probably go for another solution, which is to compose the **`Car`** with an **`Engine`** class through the constructor (or through some other means).

Code Listing 92: Different Engines

```
public abstract class Engine { }
public class GasolineEngine : Engine { }
public class DieselEngine : Engine { }

public class Car
{
    private Engine engine;
    public Car(Engine engine)
    {
        this.engine = engine;
    }
    // ...
}
```

And now that another class creates an **`Engine,`** you can give your **`Car`** any **`Engine`** you want.

Code Listing 93: Compose your Car

```
Car diesel = new Car(new DieselEngine());
Car gas = new Car(new GasolineEngine());
```

Of course, that still doesn't solve the entire problem, because who creates the **`Car,`** and does that same class or yet another create the **`Engine`**?

It is important to note here that these objects are code objects and not domain objects. The domain, or real world objects, are probably created on some assembly line. Chances are you haven't modeled the assembly line in your code, so that's probably not going to create the **`Car`** or **`Engine`**. One thing you know for sure, a **`Car`** has never created an **`Engine`** in the real world.

Information Expert

Information Expert is a principle that you are probably already applying. Information Expert addresses the problem of which object handles which responsibility. How often have you wondered what the right class was for a certain method? The answer lies with Information Expert and is deceptively simple. A responsibility should be assigned to the class that has all the information necessary to fulfill that responsibility.

Let's take a look at an example. Earlier we created an Adapter for a **`SqlServerApi`** class. Here's a small reminder:

Code Listing 94: The SqlServerApiAdapter

```
public class SqlServerApiAdapter : IDbApiAdapter
{
    private SqlServerApi api = new SqlServerApi();

    public object GetData()
    {
        DataTable table = api.GetData();
        object o = ConvertToObject(table);
        return o;
    }

    private object ConvertToObject(DataTable table)
    {
        // ...
        return null; // Placeholder.
    }

    public void SendData(object data)
    {
        DataTable table = ConvertToDataTable(data);
        api.SendData(table);
    }

    private DataTable ConvertToDataTable(object obj)
    {
        // ...
        return new DataTable();
    }
}

public class OracleApiAdapter : IDbApiAdapter
{
    private OracleApi api = new OracleApi();

    public object GetData()
    {
        return api.GetQuery();
    }

    public void SendData(object data)
    {
        api.ExecQuery(data);
    }
}
```

Now as it turns out we want to be able to create new databases. That is new functionality as it is not supported by **`GetData`** or **`SendData`**. Now where could we put this new **`CreateDatabase`** method? Maybe in our **`Program`** class?

Code Listing 95: SqlServerApi in Program

```
class Program
{
    static void Main(string[] args)
    {
        CreateDatabase("MyDB");
        SqlServerApiAdapter adapter = new SqlServerApiAdapter();
        object data = adapter.GetData();
        // ...
    }

    static void CreateDatabase(string name)
    {
        SqlServerApi api = new SqlServerApi();
        api.CreateDatabase(name);
    }
}
```

I think you can see the problem here. We've gone through all the trouble of creating an Adapter for the **`SqlServerApi`** and now we're going to use it directly in our application. It makes no sense, as **`SqlServerApiAdapter`** is the much more obvious choice. It already uses **`SqlServerApi`** anyway! And that's exactly the point of Information Expert. The **`SqlServerApiAdapter`** has all the information that's necessary to create a new database, so it is a good candidate for getting this new functionality. The alternative, **`Program`**, knows nothing about **`SqlServer`** or databases, so the choice is easily made.

The example here is kind of obvious, but it demonstrates what Information Expert is all about. I'll tell you a little secret too: I've seen software that had **`CreateDatabase`** in **`Program`** (well, not exactly that, but a likewise situation). When a code base grows large, it is not always obvious where new functionality goes.

Low Coupling

The Low Coupling principle helps in keeping the impact of changes to a minimum. Coupling between classes is when a class depends upon another. Needless to say that when a lot of classes depend upon a certain class it becomes difficult to change that class. The Information Expert Principle actually helps towards this end. Let's check the previous example with the **`SqlServerApi`**. Putting the **`CreateDatabase`** method in **`Program`** was bad design because **`Program`** didn't know about **`SqlServerApi`** and shouldn't need to know about it. It created unnecessary coupling between **`SqlServerApi`** and **`Program`**.

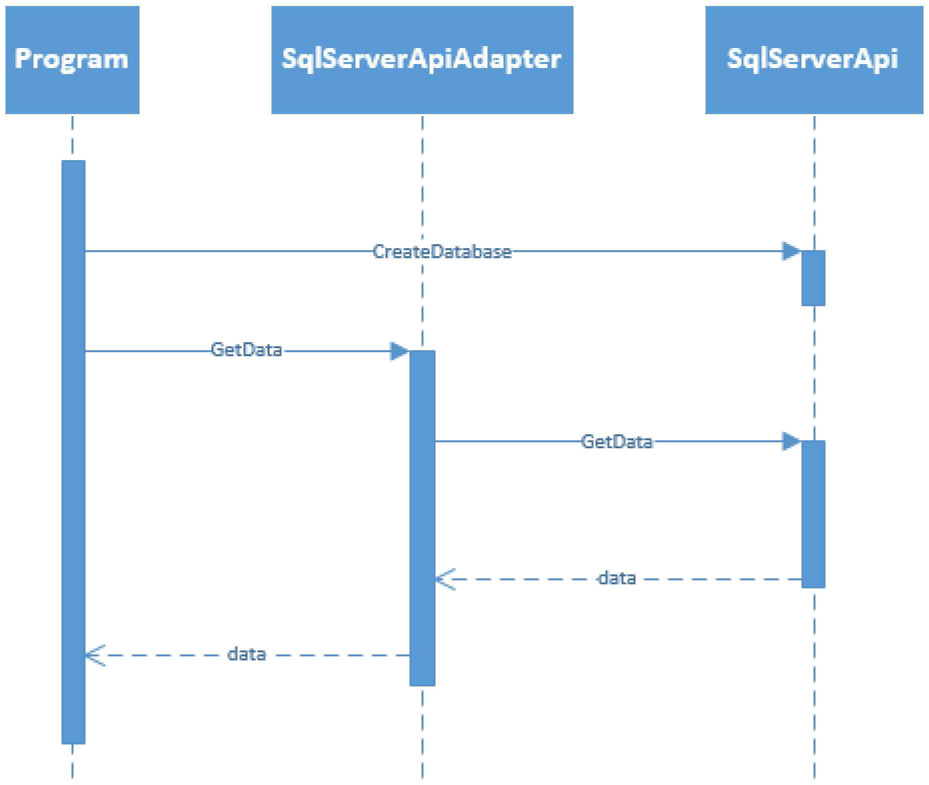


Figure 21: Sequence Diagram of High Coupling

As you can see in the diagram, there is a direct line between the **`Program`** and the **`SqlServerApi`**. If the **`SqlServerApi`** changes, it is possible we need to change **`Program`** and **`SqlServerApiAdapter`**. In the other design, where the **`SqlServerApiAdapter`** creates the database, we get another figure.

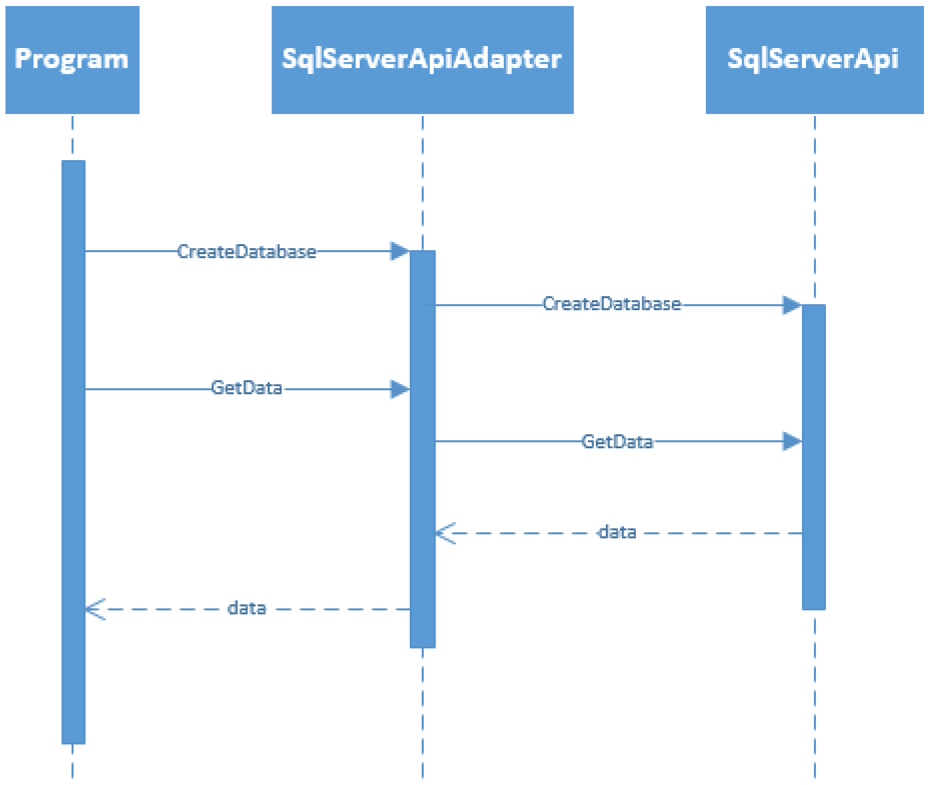


Figure 22: Sequence Diagram of Low Coupling

In this design the **`Program`** depends upon **`SqlServerApiAdapter`** and the **`SqlServerApiAdapter`** depends upon **`SqlServerApi`**. Now if **`SqlServerApi`** changes we only need to change **`SqlServerApiAdapter`**. Of course, if the change in **`SqlServerApi`** causes another (breaking) change in **`SqlServerApiAdapter`** we need to change **`Program`** too.

As you see, the Low Coupling principle can be used to compare different design solutions and we should, in most situations, pick the design which has the lowest coupling between classes. Because Information Expert guides you in finding the class that has all the necessary information for a certain responsibility, it helps lead to a design with the least coupling.

High Cohesion

Cohesion is the degree to which certain elements of a class belong together. The High Cohesion Principle states that a class' cohesion should be high. In other words, the methods and properties of a class should belong together. Let's look at another example. Suppose we have a **`Person`** class with some properties and methods:

Code Listing 96: A Person Class

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime DateOfBirth { get; set; }

    public string GetFullName ()
    {
        return String.Format("{0} {1}", FirstName, LastName);
    }

    public int GetAge()
    {
        int age = DateTime.Today.Year - DateOfBirth.Year;
        // Month and Day correction.
        if (DateOfBirth.AddYears(age) > DateTime.Today)
        {
            age -= 1;
        }
        return age;
    }
}
```

Cohesion is high because all properties and methods describe a person. Let's add a method.

Code Listing 97: Person with SalesOrders

```
public class Person
{
    // ...

    public List<SalesOrder> GetSalesOrders()
    {
        // ...
    }
}
```

The cohesion of **Person** just went down. It's not completely wrong, but you should ask yourself if a **Person** really needs to know about a **SalesOrder**. It's not unthinkable that you want to use the **Person** class in a context where you don't also want to depend on the **SalesOrder** class. Maybe we could delegate the retrieval of **SalesOrders** for a **Person** to some other class.

Code Listing 98: A SalesOrderRepository

```
public class SalesOrderRepository
{
    public List<SalesOrder> GetPersonsSalesOrders(Person person)
    {
        // ...
    }
}
```

Usage would now look as follows:

Code Listing 99: Usage of the SalesOrderRepository

```
Person sander = new Person
{
    FirstName = "Sander",
    LastName = "Rossel",
    DateOfBirth = new DateTime(1987, 11, 8)
};
SalesOrderRepository repo = new SalesOrderRepository();
List<SalesOrder> orders = repo.GetPersonsSalesOrders(sander);
```

I'm not saying this is the best solution, but it does increase the cohesion, and lower the coupling, of the **`Person`** class.

As you see, High Cohesion and Low Coupling go hand in hand. When the cohesion of a class is high the coupling is typically low. Again, the Information Expert and Low Coupling Principles lead towards a design with the highest cohesion.

Pure Fabrication

We've seen Pure Fabrication a lot already. In fact, it's what you do when some responsibility doesn't fit in any class you already have: you make up a new one specifically designed for this one task. In the previous example, we created a **`SalesOrderRepository`** because getting **`SalesOrders`** for a person in the **`Person`** class violated low coupling and high cohesion. Basically, any class that does not model a domain object is a pure fabrication (a fabrication of our imagination that is pure because it has only one responsibility).

Let's look at another example. We still have our **`Person`** class. Now what class is going to save our **`Person`** to the database? We don't have any database classes yet and a **`Person`** has most of the information necessary to save itself to the database. So it would seem alright to have the **`Person`** class do the database logic necessary to save itself to the database. We already know that's not quite a good idea, mostly because we want our **`Person`** to have low coupling and high cohesion.

Code Listing 100: Person Class that Breaks Low Coupling and High Cohesion

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime DateOfBirth { get; set; }

    public void Save()
    {
        // Database logic here.
    }
}
```

So we go about and create a pure fabrication, a class with the sole purpose of saving a **`Person`** to the database. I won't include the code here because I think you get the point. When we apply other patterns and principles we've seen in this book, we end up with a Data Mapper and, possibly, a Repository.

The Takeaway

GRASP is really useful when designing classes. Unlike anything else we've seen so far, they aren't of a very technical nature, and they don't describe classes or methods. They describe how to get an efficient design for your software. This chapter has shown that there is not one design for an application. In fact, when we only had a **`Car`** and an **`Engine`** we already had two design choices (and even more less obvious ones). Keeping GRASP in mind, we can get to some design that's at least better than many of the alternatives (there is no such thing as a "best design"). The fun thing about GRASP is that a lot of it is already common sense to most developers (like a **`Person`** shouldn't save itself), but with GRASP you have some formal rules that describe why it makes such common sense.

Chapter 6 Architecture

So far you've seen a lot of code on small scale. We've seen how to design a single class, how to have multiple classes communicate with each other, and how to abstract away certain components (like a database). Chances are you need to write more than a few classes though. When looking at the high level structures of a software system, we speak about the architecture of that system. In this chapter I'm going to discuss some technical challenges and methods that you may encounter when building an application.

I want to emphasize what I will not discuss as well. I will not discuss the job of a software architect, which is talking to the stakeholders, deciding on development processes (Agile or Waterfall), documentation, budget, and other external factors that you'll have to deal with when building an actual application.

The architecture of a system is the part that is not easily changed. Opinions on this vary, but let's consider a database. Many people would point at the database, call it the heart of any system, and tell you it's pretty hard to change later on in the development process. We now know, however, that when we abstract away the database behind a Data Mapper and a Repository (on which the software depends through abstractions) changing databases is "simply" a matter of switching our Data Mapper, and possibly Repository. Of course it's still a lot of work, and everything well have to be tested thoroughly, but we shouldn't have to change our front-end, for example.
So what is hard to change, then? Suppose half-way into the project you decide to change the interface of your Data Mapper or your Repository. That could be pretty problematic. Or maybe you decide to throw in an MVC and/or MVVM architecture. That will need a good rewriting of many components.

Don't Repeat Yourself (DRY)

One important principle we have not yet discussed is the DRY Principle, or Don't Repeat Yourself. Why do we go through all the trouble of putting everything behind abstractions, thinking about architecture, and reuse anyway? DRY is the answer. Prevent writing the same code twice. The next example will illustrate why.

Suppose you have a Person class which has save logic. We broke low coupling and high cohesion, but we can save our Person.

Code Listing 101: Person with Save Logic

```
public class Person
{
    // ...

    public void Save()
    {
        // Dataase logic for Person here.
    }
}
```

And now we need a **SalesOrder**. Because we don't have a single class that can save domain objects, we'll need to write the save functionality again, but this time for our **SalesOrder**. No problem, we'll just copy/paste it from **Person**!

Code Listing 102: SalesOrder Class with a Bug

```
public class SalesOrder
{

    // ...

    public void Save()
    {
        // Dataase logic for Person here.
    }
}
```

It's a no-brainer, but the programmer copy/pasted the **Save** method from **Person** and now the **SalesOrder** literally reads "Database logic for **Person** here." I have seen this happen many times, even in production code!

There's another problem though. The method doesn't read "Database", but "Dataase". It seems we have another bug. Let's fix the **SalesOrder** class.

Code Listing 103: Fixed SalesOrder

```
public class SalesOrder
{

    // ...

    public void Save()
    {
        // Database logic for SalesOrder here.
    }
}
```

Neat, our **SalesOrder** works now! But our **Person** class has the same bug. We must now remember to fix this or it will keep the same bug we just fixed in **SalesOrder**.

If we had one **Save** method that worked on both **SalesOrder** and **Person** we would not have the first bug (**SalesOrder** saving a **Person**) and we would only have to solve the second bug once. And that's why you should strive to make code abstract: so you can write it once, test it once, fix bugs once, and use it often.

Packages

By creating classes and methods, we can reuse our code throughout our application. But many times you want to reuse code between applications. One example is a tool, maybe an XML or JSON parser, or a data mapper, that you've written and that you want to use in other applications as well. We've seen that copy/pasting, even between applications, is not practical. You can reuse these components by creating packages (Dynamic Link Libraries or DLL's) that can be used by other applications. When you use .NET you're actually just using packages Microsoft created. Just like with class design there are some best practices when releasing and using packages. In his book *Agile Principles, Patterns, and Practices in C#*, Robert C. Martin discusses these practices[5]. I'll quickly summarize them here.

Reuse/Release Equivalence Principle (REP)

The Reuse/Release Equivalence Principle states that only what is released can be reused. A release is something that is maintained by another author or entity. The release is a product and you are the customer. Releases get updates, which may or may not be of interest to you, and you may choose to update or stick to the old version for a while. Clearly, copied code does not fit this description (as we've seen with DRY). As applications often consist of hundreds of classes and interfaces, releasing single classes is tedious and overwhelming. A package is a good candidate for release so everything that is released can be reused.

Common Reuse Principle (CRP)

The Common Reuse Principle is kind of like the High Cohesion Principle for packages. Classes that are packaged together should belong together. If you reuse one class in a package you reuse everything from that package. If any class in a package changes you need to update, revalidate your software, and redistribute your software. For this reason classes in a package should have a high cohesion (I'm talking about cohesion between classes, not per class). You won't be very happy if you get an update and have to do all that work, but you don't actually need the update. Of course this happens a lot in practice, since the alternative is having many packages, which is also not great.

It might be nice to mention that some script libraries, like jQuery UI (a JavaScript library), allow you to download only the code that's necessary for your use. You can then tick the components you want and leave the rest. Of course, such practice is not possible for compiled packages because you compile everything or nothing.

Common Closure Principle (CCP)

The Common Closure Principle states that classes that change together should be packaged together. That is because if you change a class, or responsibility, you don't want to update more than one package. You don't want to have unrelated functionality in one package (CRP) and you don't want a single functionality in more than one package (CCP).

[5] R.C. Martin, M. Martin - *Agile Principles, Patterns and Practices in C#*

Acyclic Dependencies Principle (ADP)

The Acyclic Dependency Principle is really not an issue in C# and Visual Studio. According to this principle, packages cannot depend upon packages that depend upon them, creating a cycle in your dependencies. Making such dependencies is not possible in Visual Studio, though. If you attempt to compile .NET code that violates the ADP, you'll get an error with a message that complains you are creating a circular reference. Let’s dig into it a little deeper anyway. Suppose you have four packages: Data, Business, UIComponents, and UI. UI depends upon Business and UIComponents, and Business depends upon Data.

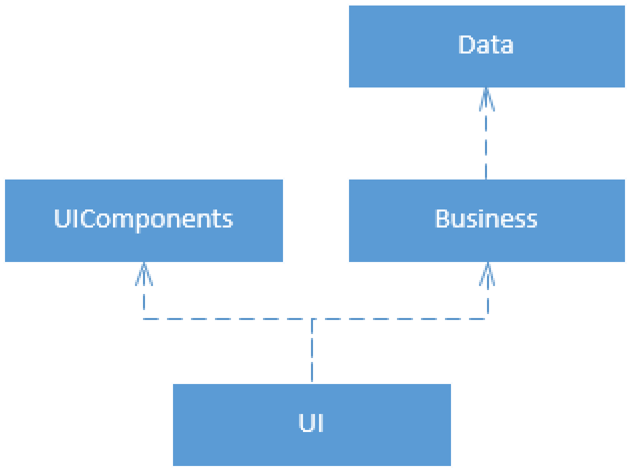


Figure 23: Dependencies without Cycles

If Business changes we know UI depends on it, so we should check whether UI still works. We know that UIComponents and Data remain unaltered though. Likewise, if Data changes we know we should check Business and UI, but not UIComponents.

Now suppose Data depends upon UI.

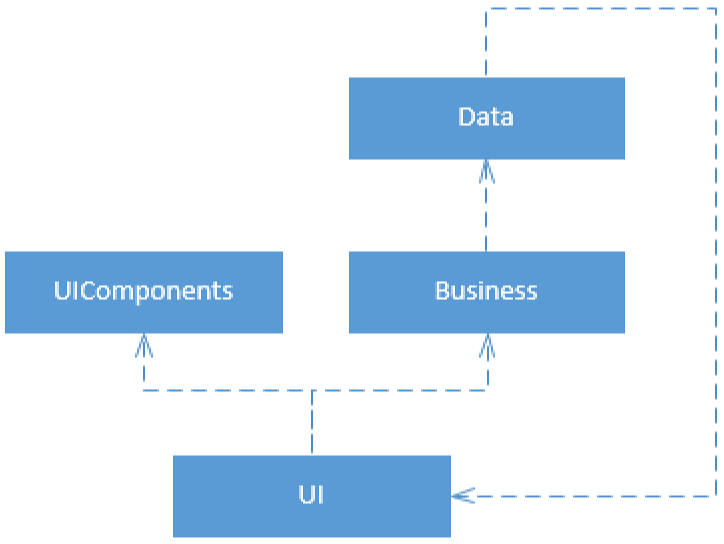


Figure 24: Dependencies with a Cycle

We have now created a cycle. If Business changes now we should check if UI still works, but because Data now depends upon UI we should also check Data. If UIComponents changes, you have to revalidate every package. This is tiresome for four packages, but in a real world application with tens of packages it's pretty much undoable. In any case, it's very poor design and prohibited by Visual Studio.

Stable Dependencies Principle (SDP)

Some packages are written once and rarely updated, while other packages change frequently. Think of the user interface: every new feature to an application needs a change in the user interface to support that feature. Your Data Mapper, hopefully, changes a lot less often. If a package rarely changes, it is said to be stable. According to the Stable Dependencies Principle, stable packages should not depend upon unstable packages. That makes sense because every time a dependency is updated, we should validate a package. That means that if a dependency changes often, we should validate often. By that logic a package is only as stable as the least stable package it depends upon, so stable packages should not depend upon less stable packages.

Stable Abstractions Principle (SAP)

For packages to be stable it is necessary that these packages have a certain level of abstraction. Without abstraction these packages could not be easily changed or extended rendering them unchangeable or unstable. We have already seen how we can create extendable software by using interfaces, inheritance, and SOLID principles.

Layers and Tiers

You will often hear the terms layered or tiered software design. A *layer* is a level of abstraction. A *tier* is a physical separation between software responsibilities. A client-server solution, for example, is a 2-tier architecture. If you decide to throw in an extra server for some business logic service, you have created a 3-tier architecture. N-tier architecture is relatively easy in object-oriented programming because modules can easily be reused on different tiers.
When you practice the principles laid out in this book a layered architecture will come almost naturally. A well-known layered structure is that of the OSI model, a model for describing the different layers of the Internet. Simply put, your browser has no idea how to send bits and bytes over a line, nor does your line know about any browser. Yet, through abstraction, both work together to present websites and applications. The same principle goes for software. Your application has no knowledge of a database, yet, through some **`IDataMapper`** interface, it is able to communicate with a data storage.

Such a layer can easily be replaced without affecting the rest of the system. When you look at the Internet, you can easily switch from a dial-up connection to ADSL to cable to wireless. No need to replace your browser or your computer. Likewise, TCP/IP, the protocol used to send and receive data on the Internet, can be switched for UDP, another faster, but less secure protocol for sending data. In your application, you can switch your databases, or rebuild your user interface, without touching the rest of the system (given that you've successfully separated concerns in different layers).

A common layered or tiered architecture may look as follows:

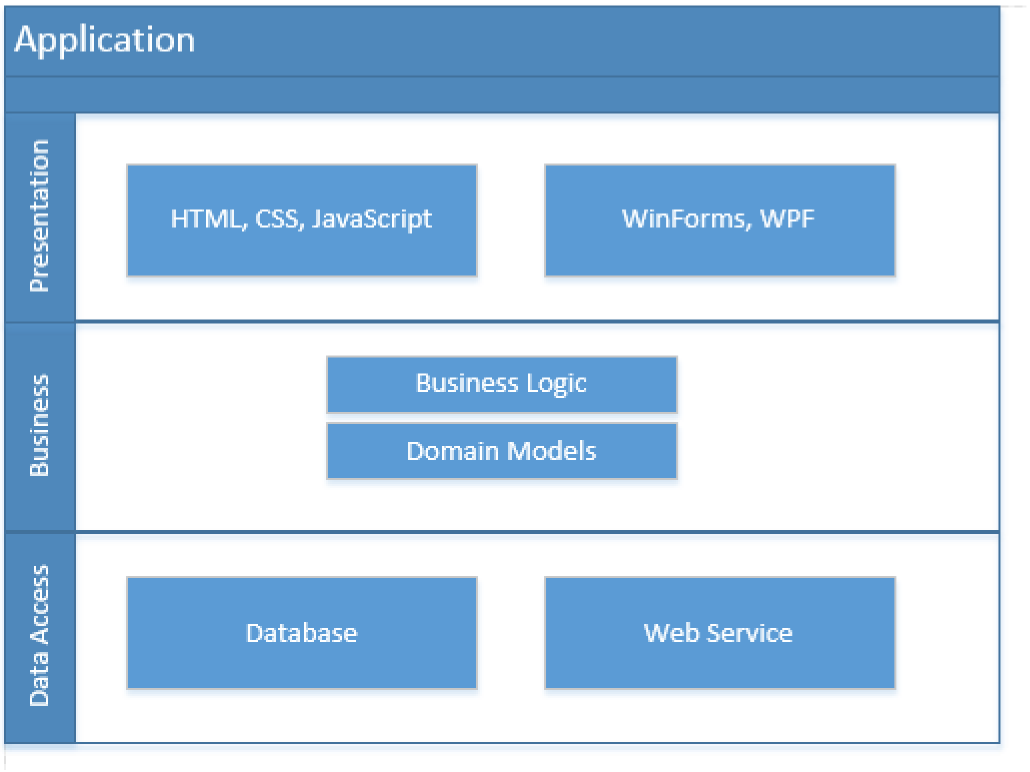


Figure 25: Common Layers or Tiers

Architectural styles and patterns

All these patterns, layers, and tiers can be combined to create greater architectural styles and patterns that are not mutually exclusive. Picking what's right for you and your project is extremely difficult and probably a matter of what you know and what you prefer. Sure, some patterns make more sense for specific applications or technologies, but overall you have some choices.

Consider an ASP.NET MVC application. The name already implies we're using MVC. The MVC framework uses REST (Representational State Transfer, meaning, in simplified terms, that your server just sits there waiting for a request, handles the request, sends a response, and continues waiting). Perhaps you're using AngularJS or KnockoutJS in the front-end, which is MVVM. You're probably already 3-tiered (HTML, CSS, AngularJS/KnockoutJS on the client, a web server, and a database server). You can opt to create some additional services for business logic that you can use for other platforms as well. Additionally, the front-end might make use of an event-driven design.

As you see, you probably use a lot of patterns already and perhaps didn't even know it. Discussing all these patterns individually requires another few books, so I won't do that. Having these tough architectural decisions made for you by the framework or technology you're using may be how you like it, or you may want to create something on your own. There are pros and cons to each approach.

The Takeaway

Object-oriented design should have no secrets for you now. You might be wondering how far you should go in abstraction and reuse. After all, just because you can doesn't mean you should. It's really hard to say where you should draw the line as it's highly subjective. I once saw the following abstraction:

Code Listing 104: NowResolver

```
public interface INowResolver
{
    DateTime GetNow();
}

public class NowResolver : INowResolver
{
    public DateTime GetNow()
    {
        return DateTime.Now;
    }
}
```

Is this going too far? Has the programmer lost himself in a sea of abstraction? You might be tempted to think so, but actually this is pretty smart. Imagine you have to run some tests on a class that gives different results dependent on the current time (or date). That's not uncommon at all; maybe it's a financial application that needs to get the exchange rate at a given date. So try testing that if the codebase has **`DateTime.Now`** directly in the methods. With the **`INowResolver`** you can inject your now and test for yesterday, now, and tomorrow.

Personally, I've never seen abstraction go too far. I have missed lots of abstractions though. Sure, having many unnecessary abstractions is hard to read, but missing abstractions is hard to maintain. So the only advice I can give here is to keep thinking, come up with different designs, compare them using the tools I've laid out in this book, and use your best judgement. Luckily, some of the harder decisions are already made for you by the framework or technology you choose for your next project.

Chapter 7 Other Paradigms

By now I hope you've realized that Object-Oriented design is pretty awesome. It's not the only way to write software out there though. In fact, C# is a multi-paradigm language. Knowing other paradigms helps to identify the strengths and weaknesses of OOP.

Procedural Programming

Procedural Programming has been around for a long time. The name comes from procedures or subroutines, also known as methods. Procedural languages, like OO languages, have methods (procedures), variables, data structures, and even some sort of modularity. Procedural languages have no notion of classes, though. If two methods need access to the same variable that variable must be declared on a global level. In larger applications this becomes problematic as it's very easy to accidentally manipulate such a variable and mess up your entire application. Well known procedural languages are C, Pascal, FORTRAN, and BASIC.

Functional Programming

Even though Functional Programming has been around since the 50's, it has recently seen an uptick in popularity. The first functional language, in 1958, was LISP. In Functional Languages functions, closely related to mathematical functions, have a central role, as just like in mathematics a function can only work with the input it has been given. This makes it possible to reason about functional programs because you know functions will not alter any program state. Since functions are so important, it is easy to compose new functions from existing functions, to return functions from functions, and to pass functions as input to other functions. We say that functions are first class citizens. Functional languages often have a sophisticated type system which makes it unnecessary to declare variable types, yet type safety is guaranteed.

C# has some functional constructs. For example LINQ and lambda's have been borrowed from functional languages. Other popular functional languages are F#, ML, Erlang, and Haskell.

Stateless programming

One big difference between Functional and OO Programming is the notion of state. Every OO language has state, variables inside a class that determine the state of a class. Having state does more for a language than you might think. For example, consider the following **`Incrementer`** class:

Code Listing 105: The Incrementer Class

```
public class Incrementer
{
    private int counter = 0;
    public void Increment()
    {
        counter += 1;
    }

    public int GetIncrement()
    {
        return counter;
    }
}
```

The state of any **`Incrementer`** instance is determined by the value of **`counter`**. When we call the **`Increment`** function we cannot know the state of our instance unless we knew the state before we called **`Increment`**. This makes it very hard to reason about programs. When we call **`GetIncrement,`** we don't know what it will return because we don't know the state of the object.

Let's check out an example. Suppose we have a class **`Math`** with a function **`Add`**.

Code Listing 106: A Math Class

```
public class Math
{
    public int Add (int a, int b)
    {
        return a + b;
    }
}
```

In this case, we know that if we call the function with parameters 1 and 2 the result will be 3. However, and this is very important, in an OO language we can never be sure.

Code Listing 107: Add with Side Effects

```
public class Math
{
    private int something;
    public int Add (int a, int b)
    {
        something += 1;
        return a + b;
    }
}
```

Nothing stops us from writing an **`Add`** function that changes the internal state of our **`Math`** class. The change of a class' internal state is called a side effect.

Functional Programming forbids such side effects. This makes it possible to predict the outcome of a function given the input parameters. It's difficult though; try writing an application without state or side effects.

Aspect-Oriented Programming

OOP is great for many things. One thing it isn't very good at is handling so-called cross-cutting concerns. A cross-cutting concern is a general responsibility that is used throughout the application. Think of logging, for example, which you might want to implement in just about any method in every layer and every project. The result is that all your methods start with something like **`Logger.Log(currentMethodName, inputParameters)`**. Another example is something like the **`INotifyPropertyChanging`** and **`INotifyPropertyChanged`** interfaces in .NET. They can be implemented on a class, which should then raise the **`PropertyChanging`** and **`PropertyChanged`** events whenever a property changes its value. The implementation is rather tedious:

Code Listing 108: INotifyPropertyChanging and INotifyPropertyChanged

```
public class SomeClass : INotifyPropertyChanging, INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;
    public event PropertyChangingEventHandler PropertyChanging;

    private int someProperty;
    public int SomeProperty
    {
        get { return someProperty; }
        set
        {
            if (value != someProperty)
            {
                RaisePropertyChanging("SomeProperty");
                someProperty = value;
                RaisePropertyChanged("SomeProperty");
            }
        }
    }

    private void RaisePropertyChanging(string propertyName)
    {
        if (PropertyChanging != null)
        {
            PropertyChanging(this, new PropertyChangingEventArgs(propertyName));
        }
    }

    private void RaisePropertyChanged(string propertyName)
    {
        if (PropertyChanged != null)
        {
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

That’s a lot of boilerplate code! A class that would need one line of code now has over thirty!

Aspect-Oriented Programming aims to fix this issue by altering existing code dynamically. The Proxy or Decorator Patterns are often used to accomplish this. One of the most known AOP frameworks is probably PostSharp[6]. Another is DynamicProxy by Castle Project[7].

[6] https://www.postsharp.net/

[7] http://www.castleproject.org/projects/dynamicproxy/

Conclusion

Object-oriented programming is an awesome programming paradigm which enables you to write huge enterprise applications that are readable, extendable, and maintainable. The tools available to you, such as Inheritance, Encapsulation, Interfaces, and Design Patterns, are quite numerous. Still, many projects fail completely, never seeing the light of day. When the tools are applied incorrectly, or not at all, you will still end up with a bunch of unmaintainable spaghetti code. For this reason it is important that you study and practice different patterns and designs so you can make an informed decision on what to use when that time comes.