



Articles » Development Lifecycle » Design and Architecture » Patterns and Practices

Design Patterns 2 of 3 - Structural Design Patterns

By Kanasz Robert, 16 Oct 2012

★★★★★ 4.94 (96 votes)

[Download StructuralPatterns_StructuralCodeExamples-noexe.zip- 22.7 KB](#)

[Download StructuralPatterns_StructuralCodeExamples.zip - 127.1 KB](#)

[Download StructuralPatterns_RealWorldExamples-noexe.zip - 36.1 KB](#)

[Download StructuralPatterns_RealWorldExamples.zip - 198.3 KB](#)

- [Introduction](#)
- [Adapter pattern](#)
 - [Structural code example](#)
 - [Real world example](#)
- [Bridge pattern](#)
 - [Structural code example](#)
 - [Real world example](#)
- [Composite pattern](#)
 - [Structural code example](#)
 - [Real world example](#)
- [Decorator pattern](#)
 - [Structural code example](#)
 - [Real world example](#)
- [Facade pattern](#)
 - [Structural code example](#)
 - [Real world example](#)
- [Flyweight pattern](#)
 - [Structural code example](#)
 - [Real world example](#)
- [Proxy pattern](#)
 - [Structural code example](#)

- Real world example

Introduction

This is the second article about design patterns. In the [first](#) article I have discussed about creational design patterns and now I will describe another set of patterns called Structural design patterns.

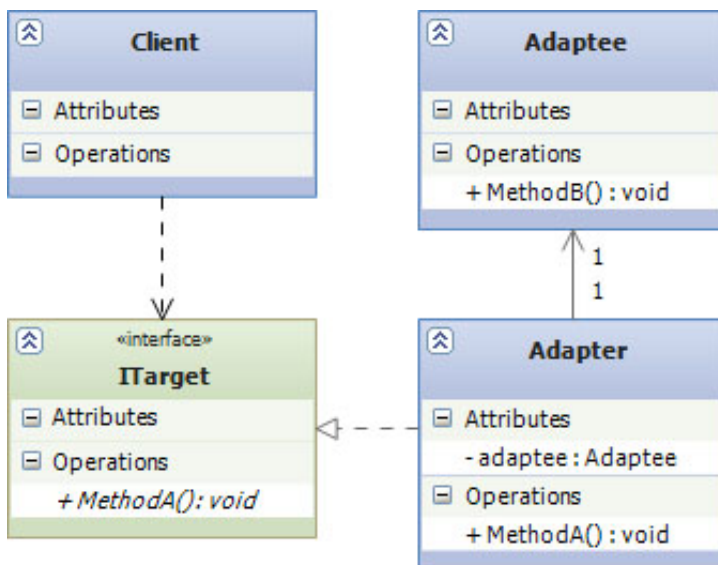
In software engineering, structural design patterns are design patterns that ease the design by identifying a simple way to realize relationships between entities.

Adapter pattern

The adapter pattern is a design pattern that is used to allow two incompatible types to communicate. Where one class relies upon a specific interface that is not implemented by another class, the adapter acts as a translator between the two types.

Adapter pattern is structural pattern which defines a manner for creating relationships between objects. This pattern translates one interface for a class into another compatible interface. Adapter pattern is newer used when creating a new system. It is usually implemented when requirements are changed and we must implement some functionality of classes which interfaces are not compatible with ours.

Structural code example



The UML diagram below describes an implementation of the adapter design pattern. Diagram consists of four parts:

- **Client**: represents the class which need to use an incompatible interface. This incompatible interface is implemented by **Adaptee**.
- **ITarget**: defines a domain-specific interface that client uses. In this case it is an simple interface, but in some situations it could be an abstract class which adapter inherits. In this case methods of this abstract class must be overridden by concrete adapter.
- **Adaptee**: represents a class provides a functionality that is required by client.
- **Adapter**: is concrete implementation of adapter. This class translates incompatible interface of

Adaptee into interface of **Client**.

```
static class Program
{
    static void Main()
    {
        var client = new Client(new Adapter());
        client.Request();
    }

    public interface ITarget
    {
        void MethodA();
    }

    public class Client
    {
        private readonly ITarget _target;

        public Client(ITarget target)
        {
            _target = target;
        }

        public void Request()
        {
            _target.MethodA();
        }
    }

    public class Adaptee
    {
        public void MethodB()
        {
            Console.WriteLine("Adaptee's MethodB called");
        }
    }

    public class Adapter : ITarget
    {
        readonly Adaptee _adaptee = new Adaptee();

        public void MethodA()
        {
            _adaptee.MethodB();
        }
    }
}
```

Real world example

In this example I have decided to create a simple serialization application which allows you to serialize objects into JSON or XML format. For example I have created **PersonInfo** class which will be serialized. For serialization I have used **JavaScriptSerializer** and **XmlSerializer** classes. These classes play in this example role as Adaptees with incompatible interfaces. **ISerializerAdapter (ITarget)** is interface which must be implemented by concrete adapter. It has one method called **Serialize** which serializes object into appropriate format. **JSONSerializerAdapter** and **XMLSerializerAdapter** classes implement **ISerializerAdapter** interface. These two classes are adapters.

```
public interface ISerializerAdapter
{
    string Serialize<T>(object objectToSerialize);
}

public class JsonSerializerAdapter:ISerializerAdapter
{
    public string Serialize<T>(object objToSerialize)
    {
        var serializer = new JavaScriptSerializer();
        return serializer.Serialize(objToSerialize);
    }
}

public class XMLSerializerAdapter:ISerializerAdapter
{
    public string Serialize<T>(object objToSerialize)
    {
        using(var writer = new StringWriter())
        {
            var serializer = new XmlSerializer(typeof(T));
            serializer.Serialize(writer, objToSerialize);
            return writer.ToString();
        }
    }
}

public class PersonInfo
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime BirthDate { get; set; }
    public double Height { get; set; }
    public double Weight { get; set; }
}

class Program
{
    static void Main()
    {
        var serializer = new DataSerializer(new XMLSerializerAdapter());
        Console.WriteLine(serializer.Render());

        serializer = new DataSerializer(new JsonSerializerAdapter());
        Console.WriteLine(serializer.Render());
    }
}

public class DataSerializer
{
    private readonly ISerializerAdapter _serializer;

    public DataSerializer(ISerializerAdapter serializer)
    {
        _serializer = serializer;
    }

    public string Render()
    {
        var sb = new StringBuilder();

        var list = new List<PersonInfo>
```

```

        {
            new PersonInfo
            {
                FirstName = "Robert",
                LastName = "Kanasz",
                BirthDate = new DateTime(1985, 8, 19),
                Height = 168, Weight = 71
            },
            new PersonInfo
            {
                FirstName = "John",
                LastName = "Doe",
                BirthDate = new DateTime(1981, 9, 25),
                Height = 189,
                Weight = 80
            },
            new PersonInfo
            {
                FirstName = "Jane",
                LastName = "Doe",
                BirthDate = new DateTime(1989, 12, 1),
                Height = 164,
                Weight = 45
            }
        };

        foreach (var personInfo in list)
        {
            sb.AppendLine(_serializer.Serialize<PersonInfo>(personInfo));
        }

        return sb.ToString();
    }
}

```

Bridge pattern

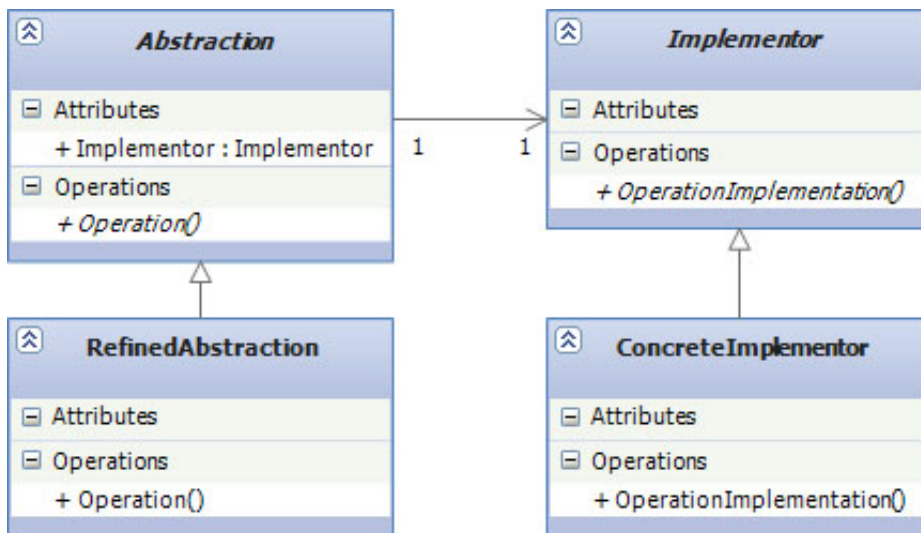
he bridge pattern is a design pattern that separates the abstract elements of a class from its technical implementation. This provides a cleaner implementation of real-world objects and allows the implementation details to be changed easily.

Bridge pattern is very valuable pattern because it allows you to separate abstract elements of class from the implementation details. This pattern can be used, when the class vary often because changes to the code base can be made very easily with minimal knowledge about the program.

Sometimes an implementation can have two or more different implementations. Let's consider a program that handles persistence of objects on different platforms. Some of objects should be saved into database and other objects into file system. When simply extends the program with this functionality, it could cause problems because we binds abstraction with implementation. In this case it is more suitable to use Bridge pattern and separate abstraction from its implementation. If we don't use this pattern, we will find that implementation details are included within an abstraction.

One of the biggest benefits of Bridge pattern is ability to change implementation details at run time. This could permit the user to switch implementations to determine how the software interoperates with other systems.

Structural code example



The UML diagram below describes an implementation of the bridge design pattern. Diagram consists of four parts:

- **Abstraction:** defines an abstraction interface. It acts as base class for other refined abstraction classes. It also holds reference to particular implementation that it is using for platform specific functionality.
- **RefinedAbstraction:** it provides more specific variations upon abstraction but it doesn't contain any implementation details. De facto it only extends abstraction.
- **Implementor:** it defines the interface for implementation classes.
- **ConcreteImplementor:** this class inherits from RefinedAbstraction class. There may be more than one instances of Implementor classes providing the same interface but platform specific functionality.

```

static class Program
{
    static void Main()
    {
        Abstraction abstraction = new RefinedAbstraction
        {
            Implementor = new ConcreteImplementorA()
        };

        abstraction.Operation();
    }
}

abstract class Implementor
{
    public abstract void Operation();
}

class Abstraction
{
    protected Implementor implementor;

    public Implementor Implementor
    {
        set { implementor = value; }
    }
}
  
```

```

        public virtual void Operation()
        {
            implementor.Operation();
        }
    }

    class RefinedAbstraction : Abstraction
    {
        public override void Operation()
        {
            implementor.Operation();
        }
    }

    class ConcreteImplementorA : Implementor
    {
        public override void Operation()
        {
            Console.WriteLine("ConcreteImplementor's Operation");
        }
    }

```

Real world example

In this example I decided to create a simple messaging application. For this purpose **Message** class was created. This class acts as **Abstraction** for **UserEditedMessage** class. This class has one protected field of type **MessageSenderBase**. **MessageSender** base is implementor and abstract class from all concrete implementations of message senders. Three message senders were created: **EmailSender**, **MsmqSender** and **WebServiceSender**. This is only demonstrative example and hence no realistic functionality of this senders is implemented.

```

public class Message
{
    protected MessageSenderBase messageSender { get; set; }
    public string Title { get; set; }
    public string Body { get; set; }
    public int Importance { get; set; }

    public Message()
    {
        messageSender = new EmailSender();
    }

    public virtual void Send()
    {
        messageSender.SendMessage(Title, Body, Importance);
    }
}

public class EmailSender : MessageSenderBase
{
    public override void SendMessage(string title, string body, int importance)
    {
        Console.WriteLine("Email\n{0}\n{1}\n{2}\n", title, body, importance);
    }
}

```

```

public class MsmqSender : MessageSenderBase
{
    public override void SendMessage(string title, string body, int importance)
    {
        Console.WriteLine("MSMQ\\n{0}\\n{1}\\n{2}\\n", title, body, importance);
    }
}

public class WebServiceSender : MessageSenderBase
{
    public override void SendMessage(string title, string body, int importance)
    {
        Console.WriteLine("Web Service\\n{0}\\n{1}\\n{2}\\n", title, body, importance);
    }
}

public abstract class MessageSenderBase
{
    public abstract void SendMessage(string title, string details, int importance);
}

public class UserEditedMessage : Message
{
    public string UserComments { get; set; }

    public UserEditedMessage(MessageSenderBase messageSender)
    {
        this.messageSender = messageSender;
    }

    public override void Send()
    {
        string fullBody = string.Format("{0}\\nCOMMENTS\\n{1}", Body, UserComments);
        messageSender.SendMessage(Title, fullBody, Importance);
    }
}

class Program
{
    static void Main()
    {
        var listOfMessages = new List<Message>
        {
            new Message
            {
                Body = "Hello World 1",
                Importance = 1,
                Title = "Hello World Title 1"
            },
            new UserEditedMessage(new EmailSender())
            {
                Body = "User Edited Message",
                Importance = 3,
                UserComments = "Comments",
                Title = "User Edited Title"
            }
        };

        foreach (var message in listOfMessages)
        {
            message.Send();
        }
    }
}

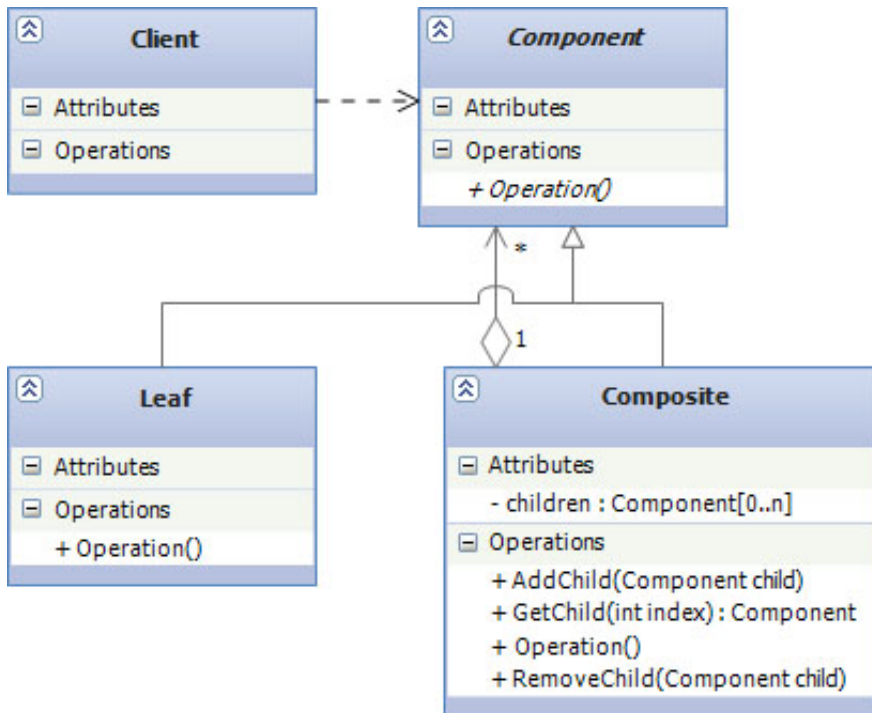
```


Composite pattern

The composite pattern is a design pattern that is used when creating hierarchical object models. The pattern defines a manner in which to design recursive tree structures of objects, where individual objects and groups can be accessed in the same manner.

When the program needs to manipulate a tree structure of objects, you can use composite design pattern. The intent of this design pattern is to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Structural code example



The UML diagram below describes an implementation of the composite design pattern. This diagram consists of three parts:

- **Component:** is an abstraction for all components, including composite ones. It declares the interface for objects in the composition. Sometimes it defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.
- **Composite:** this is a key element of this design pattern. It represent a Composite components. Composite element are elements which have child elements. It implements the methods to add and remove children elements and it implements all Component methods, generally by delegating them to its children.
- **Leaf:** represents leaf objects in the composition and implements all Component methods.

```

static class Program
{
    static void Main()
    {
        var root = new Composite("root");
        root.AddChild(new Leaf("Leaf 1"));
        root.AddChild(new Leaf("Leaf 2"));
    }
}
  
```

```

        var comp = new Composite("Composite C");
        comp.AddChild(new Leaf("Leaf C.1"));
        comp.AddChild(new Leaf("Leaf C.2"));

        root.AddChild(comp);
        root.AddChild(new Leaf("Leaf 3"));

        var leaf = new Leaf("Leaf 4");
        root.AddChild(leaf);
        root.RemoveChild(leaf);

        root.Display(1);
    }
}

public abstract class Component
{
    protected readonly string name;

    protected Component(string name)
    {
        this.name = name;
    }

    public abstract void Operation();
    public abstract void Display(int depth);
}

class Composite : Component
{
    private readonly List<Component> _children = new List<Component>();

    public Composite(string name)
        : base(name)
    {
    }

    public void AddChild(Component component)
    {
        _children.Add(component);
    }

    public void RemoveChild(Component component)
    {
        _children.Remove(component);
    }

    public override void Display(int depth)
    {
        Console.WriteLine(new String('-', depth) + name);

        foreach (Component component in _children)
        {
            component.Display(depth + 2);
        }
    }

    public override void Operation()
    {
        string message = string.Format("Composite with {0} child(ren).", _children.Count);
        Console.WriteLine(message);
    }
}

```

```

public class Leaf : Component
{
    public Leaf(string name)
        : base(name)
    {
    }

    public override void Operation()
    {
        Console.WriteLine("Leaf.");
    }

    public override void Display(int depth)
    {
        Console.WriteLine(new String('-', depth) + name);
    }
}

```

Real world example

The good examples how to demonstrate this design pattern in real world are graphic editors. Every shape that can be created by graphic editor can be a basic or complex. The example of a simple shape is a line. Complex shape consists of many simple shapes. These shapes (complex and simple) have a lot of operations in common. One of the operations is rendering of a shape to the screen. Since shapes follow a part-whole hierarchy, composite pattern can be used to enable the program to deal with all shapes uniformly. In this example I have created **IShape** interface that acts as Component. All composites and leafs must implement this interface. This interface consists of public property **Name** and method **Render()**. Another class called **ComplexShape** is Composite class. It has one private collection of child shapes **_shapes**. This collection is managed by **AddShape()** and **RemoveShape()** methods. This collection may contain every object which implements **IShape** interface. Render method of **ComplexShape** class loops through collection of child elements **_shapes** and call their **Render()** method. The last three classes: **Rectangle**, **Circle** and **Line** represent Leaf classes.

```

public interface IShape
{
    string Name { get; set; }
    void Render(int depth);
}

public class ComplexShape : IShape
{
    public ComplexShape(string name)
    {
        Name = name;
    }

    private readonly List<IShape> _shapes = new List<IShape>();

    public string Name { get; set; }

    public void Render(int depth)
    {
        Console.WriteLine(new String('-', depth) + Name);
        foreach (var shape in _shapes)
        {
            shape.Render(depth+1);
        }
    }
}

```

```
    }

    public void AddShape(IShape shape)
    {
        _shapes.Add(shape);
    }

    public void RemoveShape(IShape shape)
    {
        _shapes.Remove(shape);
    }
}

public class Circle : IShape
{
    public Circle(string name)
    {
        Name = name;
    }

    public string Name { get; set; }

    public void Render(int depth)
    {
        Console.WriteLine(new String('-', depth) + Name);
    }
}

public class Line : IShape
{
    public Line(string name)
    {
        Name = name;
    }

    public string Name { get; set; }

    public void Render(int depth)
    {
        Console.WriteLine(new String('-', depth) + Name);
    }
}

public class Rectangle:IShape
{
    public Rectangle(string name)
    {
        Name = name;
    }

    public string Name { get; set; }

    public void Render(int depth)
    {
        Console.WriteLine(new String('-', depth) + Name);
    }
}

class Program
{
    static void Main(string[] args)
    {

```

```

ComplexShape shape1= new ComplexShape("Complex Shape #1");
Line line1 = new Line("Blue Line #1");
Line line2 = new Line("Blue Line #2");

shape1.AddShape(line1);
shape1.AddShape(line2);

ComplexShape shape2 = new ComplexShape("Complex Shape #2");
Circle circle1 = new Circle("Yellow Circle #1");
shape2.AddShape(circle1);
shape1.AddShape(shape2);

shape1.Render(1);

    }
}

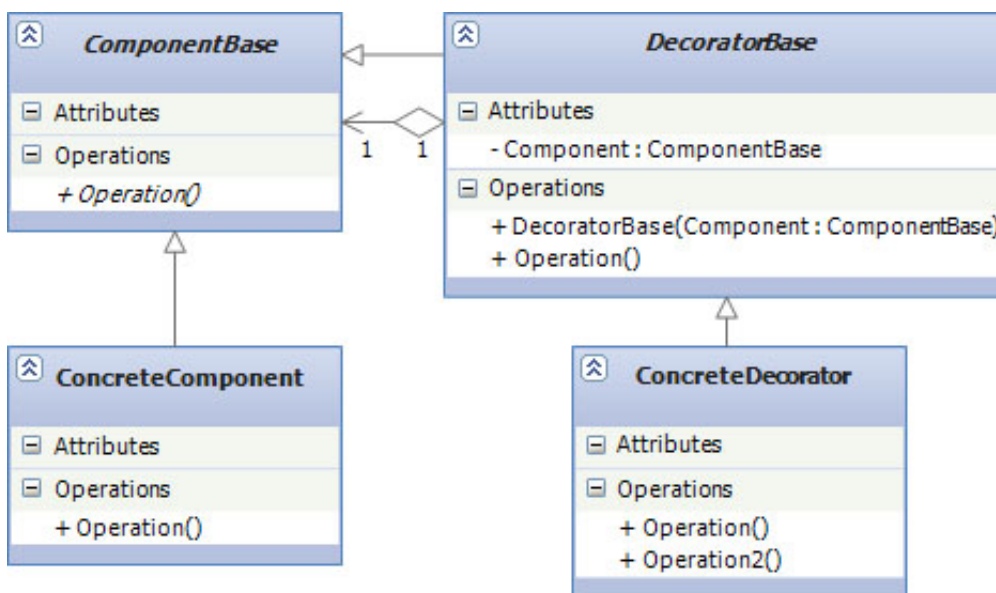
```

Decorator pattern

The decorator pattern is a design pattern that extends the functionality of individual objects by wrapping them with one or more decorator classes. These decorators can modify existing members and add new methods and properties at run-time.

Extending object's functionality can be done statically (at compile time) by using inheritance or dynamically (at run time) by wrapping them in an object of a decorator class. Decorator pattern is applied on individual objects, not classes. Decorator pattern is an alternative to subclassing. This pattern allows you to create a multiple decorators that can be stacked on the top of each other, each time adding a new functionality to the overridden method.

Structural code example



The UML diagram below describes an implementation of the decorator design pattern. This diagram consists of four parts:

- **ComponentBase**: is a base abstract class for all concrete components and decorators. This class

defines standard members, that must be implemented by this types of classes.

- **ConcreteComponent:** this class inherits from **ComponentBase**. There may be multiple concrete component classes, each defining a type of object that may be wrapped by the decorators.
- **DecoratorBase:** prepresents abstract base class for all decorators. It adds a constructor that accepts a **ComponentBase** object as its parameter. The passed object is the component that will be wrapped. As the wrapped object must inherit from **ComponentBase**, it may be a concrete component or another decorator. This allows for multiple decorators to be applied to a single object.
- **ConcreteDecorator:** this class represents the concrete decorator for a component. It may include some additional methods which extends functionality of components. Operations members can be used in two manners. They can be unchanged and in this case the base method of component is called or the operation may be changed and in this case the operation method is changed or entirely replaced by new implementation.

```
static class Program
{
    static void Main()
    {
        var component = new ConcreteComponent();
        var decorator = new ConcreteDecorator(component);
        decorator.Operation();
    }
}

public abstract class ComponentBase
{
    public abstract void Operation();
}

class ConcreteComponent : ComponentBase
{
    public override void Operation()
    {
        Console.WriteLine("ConcreteComponent.Operation()");
    }
}

public abstract class DecoratorBase : ComponentBase
{
    private readonly ComponentBase _component;

    protected DecoratorBase(ComponentBase component)
    {
        _component = component;
    }

    public override void Operation()
    {
        _component.Operation();
    }
}

public class ConcreteDecorator : DecoratorBase
{
    public ConcreteDecorator(ComponentBase component) : base(component) { }

    public override void Operation()
    {
        base.Operation();
        Console.WriteLine("[ADDITIONAL CODE BLOCK]");
    }
}
```

```
}
```

Real world example

Let's take a look at a real world example. In this example you can prepare your custom favourite sandwich. In this case we have **Sandwich** class which is a component base class. This class has two public abstract methods. **GetDescription** method returns full name of sandwich with all ingredients. Second method called **GetPrice** returns price of current sandwich. I have created two sandwiches: **TunaSandwich** and **VeggieSandwich**. Both of these classes inherit from **Sandwich** class. **SandwichDecorator** class is a base class for all decorators. This class inherits from **Sandwich** class too. Three decorators were prepared: Olives, Cheese, Corn. These decorators override **GetDescription** and **GetPrice** methods of the **Sandwich** class. The first step is to create a concrete sandwich. You can choose from two options: **VeggieSandwich** and **TunaSandwich**. If it is done, you can add some ingredients by creating an appropriate decorator and wrapping your sandwich with this decorator. If you want to add another ingredient, just create a new instance of decorator and wrap sandwich with this new decorator. By calling the method **GetDescription** you will get name of sandwich with all ingredients.

```
public abstract class Sandwich
{
    public abstract string GetDescription();

    public abstract double GetPrice();

    public string Description { get; set; }
}

public class TunaSandwich : Sandwich
{
    public TunaSandwich()
    {
        Description = "Tuna Sandwich";
    }

    public override string GetDescription()
    {
        return Description;
    }

    public override double GetPrice()
    {
        return 4.10;
    }
}

public class VeggieSandwich : Sandwich
{
    public VeggieSandwich()
    {
        Description = "Veggie Sandwich";
    }

    public override string GetDescription()
    {
        return Description;
    }

    public override double GetPrice()
```

```
    {
        return 3.45;
    }
}

public class Cheese:SandwichDecorator
{
    public Cheese(Sandwich sandwich) : base(sandwich)
    {
        Description = "Cheese";
    }

    public override string GetDescription()
    {
        return Sandwich.GetDescription()+" "+Description;
    }

    public override double GetPrice()
    {
        return Sandwich.GetPrice()+ 1.23;
    }
}

public class Corn : SandwichDecorator
{
    public Corn(Sandwich sandwich)
        : base(sandwich)
    {
        Description = "Corn";
    }

    public override string GetDescription()
    {
        return Sandwich.GetDescription()+" "+Description;
    }

    public override double GetPrice()
    {
        return Sandwich.GetPrice()+ 0.35;
    }
}

public class Olives : SandwichDecorator
{
    public Olives(Sandwich sandwich)
        : base(sandwich)
    {
        Description = "Olives";
    }

    public override string GetDescription()
    {
        return Sandwich.GetDescription()+" "+Description;
    }

    public override double GetPrice()
    {
        return Sandwich.GetPrice()+ 0.89;
    }
}

public class SandwichDecorator : Sandwich
```



```

{
    protected Sandwich Sandwich;

    public SandwichDecorator(Sandwich sandwich)
    {
        Sandwich = sandwich;
    }

    public override string GetDescription()
    {
        return Sandwich.GetDescription();
    }

    public override double GetPrice()
    {
        return Sandwich.GetPrice();
    }
}

class Program
{
    static void Main(string[] args)
    {
        Sandwich mySandwich = new VeggieSandwich();
        Console.WriteLine(mySandwich.GetPrice());
        Console.WriteLine(mySandwich.GetDescription());
        mySandwich = new Cheese(mySandwich);
        Console.WriteLine(mySandwich.GetPrice());
        Console.WriteLine(mySandwich.GetDescription());
        mySandwich = new Corn(mySandwich);
        Console.WriteLine(mySandwich.GetPrice());
        Console.WriteLine(mySandwich.GetDescription());
        mySandwich = new Olives(mySandwich);
        Console.WriteLine(mySandwich.GetPrice());
        Console.WriteLine(mySandwich.GetDescription());
    }
}

```

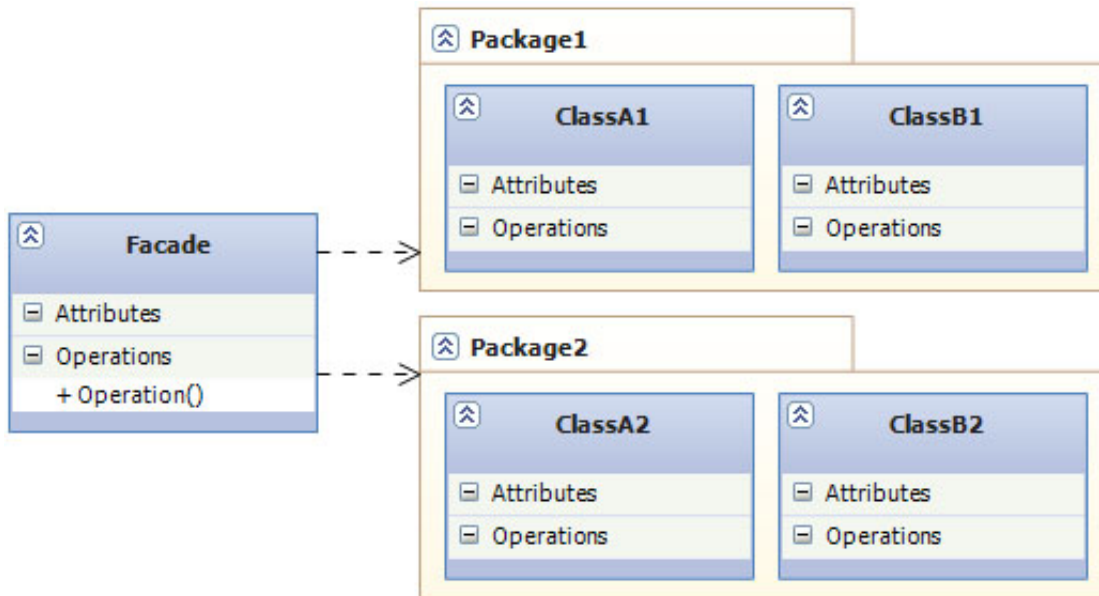
Facade pattern

The facade pattern is a design pattern that is used to simplify access to functionality in complex or poorly designed subsystems. The facade class provides a simple, single-class interface that hides the implementation details of the underlying code.

Facade pattern is generally used to simplify interface to a larger body of code. This pattern is very helpful when you deal with many independent classes or classes that require the use of multiple methods, especially when they are difficult to use or difficult to understand. We can tell that facade pattern is some kind of wrapper that contains a set of members they are easy to understand and use. This pattern is used when wrapped subsystem is poorly designed and you can have no possibility to refactor its code.

Facade pattern makes software library easier to use, understand and test. It makes library more readable and can reduce dependencies.

Structural code example



The UML diagram below describes an implementation of the facade design pattern. This diagram consists of three parts:

- **Facade:** This class contains the set of simple functions that are made available to its users and that hide the complexities of the difficult-to-use subsystems.
- **PackageA/B:** The complex functionality that is accessed via the facade class does not necessarily reside in a single assembly. The packages in the diagram illustrate this, as each can be an assembly containing classes.
- **ClassA/B:** These classes contain the functionality that is being presented via the facade.

```

static class Program
{
    static void Main()
    {
        Facade facade = new Facade();
        facade.PerformAction();
    }
}

public class Facade
{
    public void PerformAction()
    {
        var c1a = new Class1A();
        var c1b = new Class1B();
        var c2a = new Class2A();
        var c2b = new Class2B();

        var result1a = c1a.Method1A();
        var result1b = c1b.Method1B(result1a);
        var result2a = c2a.Method2A(result1a);
        c2b.Method2B(result1b, result2a);
    }
}

public class Class1A
{
    public int Method1A()
    {
        Console.WriteLine("Class1A.Method1A return value: 1");
    }
}
  
```

```

        return 1;
    }
}

public class Class1B
{
    public int Method1B(int param)
    {
        Console.WriteLine("Class1B.Method1B return value: {0}", param+1);
        return param+1;
    }
}

public class Class2A
{
    public int Method2A(int param)
    {
        Console.WriteLine("Class2A.Method2A return value: {0}", param+2);
        return param+2;
    }
}

public class Class2B
{
    public void Method2B(int param1, int param2)
    {
        Console.WriteLine("Class2B.Method2B return value: {0}", param1+param2 );
    }
}

```

Real world example

In this example we need to start computer. **Computer** class acts as facade which encapsulates other complex classes represented by: **HardDrive** class, **Memory** class and **CPU** class. Each of these classes has operations which must be performed when **Start()** method of Computer class is called.

```

public class Computer
{
    private readonly CPU _cpu;
    private readonly HardDrive _hardDrive;
    private readonly Memory _memory;

    private const long BootAddress = 1;
    private const long BootSector = 1;
    private const int SectorSize = 10;

    public Computer()
    {
        _cpu = new CPU();
        _hardDrive = new HardDrive();
        _memory = new Memory();
    }

    public void Start()
    {
        _cpu.Freeze();
        _memory.Load(BootAddress, _hardDrive.Read(BootSector, SectorSize));
        _cpu.Jump(BootAddress);
        _cpu.Execute();
    }
}

```

```

    }
}

public class CPU
{
    public void Freeze()
    {
        Console.WriteLine("CPU is frozen");
    }

    public void Jump(long position)
    {
        Console.WriteLine("Jumping to position: {0}", position);
    }

    public void Execute()
    {
        Console.WriteLine("Executing...");
    }
}

public class HardDrive
{
    public byte[] Read(long lba, int size)
    {
        var bytes = new byte[size];
        var random = new Random();
        random.NextBytes(bytes);
        return bytes;
    }
}

public class Memory
{
    public void Load(long position, byte[] data)
    {
        Console.WriteLine("Loading data: ");
        foreach (var b in data)
        {
            Console.Write(b+ " ");
            Thread.Sleep(1000);
        }

        Console.WriteLine("\nLoading completed");
    }
}

class Program
{
    static void Main()
    {
        var computer = new Computer();
        computer.Start();
    }
}

```

Flyweight pattern

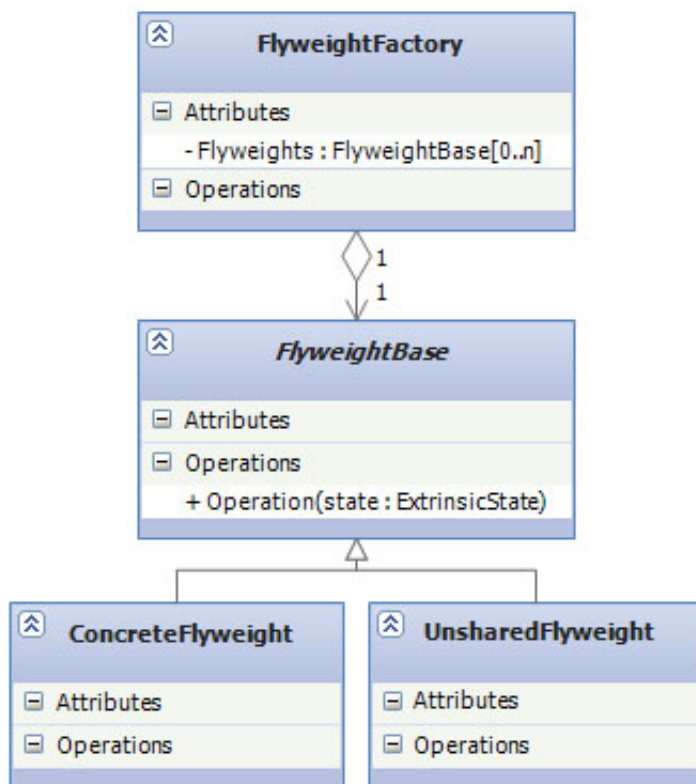
The flyweight pattern is a design pattern that is used to minimize resource usage when working with very large numbers of objects. When creating many thousands of identical objects, stateless flyweights can lower the memory used to a manageable level.

Sometimes programs work with a large number of objects which have the same structure and some states of this objects don't vary in time. When we use traditional approach and we create instances of this objects and fill state variables with values, the memory and storage requirements may unacceptably increase. To solve this problem we can use Flyweight pattern.

A flyweight is an object that minimizes memory use by sharing as much data as possible with other similar objects; it is a way to use objects in large numbers when a simple repeated representation would use an unacceptable amount of memory. For each of objects which use shared data only reference to shared object is saved.

The flyweight design pattern often uses a variation on the factory method pattern for the generation of the shared objects. The factory receives a request for a flyweight instance. If a matching object is already in use, that particular object is returned. If not, a new flyweight is generated.

Structural code example



The UML diagram below describes an implementation of the flyweight design pattern. This diagram consists of four parts:

- **FlyweightBase:** declares an interface through which flyweights can receive and act on extrinsic state.
- **ConcreteFlyweight:** implements the **Flyweight** interface and adds storage for intrinsic state (shared state), if any. A **ConcreteFlyweight** object must be sharable. Any state it stores must be intrinsic, that is, it must be independent of the **ConcreteFlyweight** object's context.
- **UnsharedFlyweight:** not all Flyweight subclasses need to be shared. The Flyweight interface enables sharing, but it doesn't enforce it. It is common for **UnsharedFlyweight** objects to have

ConcreteFlyweight objects as children at some level in the flyweight object structure.

- **FlyweightFactory:** The factory creates and manages flyweight objects. In addition the factory ensures sharing of the flyweight objects. The factory maintains a pool of different flyweight objects and returns an object from the pool if it is already created, adds one to the pool and returns it in case it is new.

```
static class Program
{
    static void Main()
    {
        int extrinsicstate = 22;

        var factory = new FlyweightFactory();
        var flyweightA = factory.GetFlyweight("A");
        flyweightA.StatefulOperation(--extrinsicstate);

        var flyweightB = factory.GetFlyweight("B");
        flyweightB.StatefulOperation(--extrinsicstate);

        var flyweightC = factory.GetFlyweight("C");
        flyweightC.StatefulOperation(--extrinsicstate);

        var unsharedFlyweight = new UnsharedFlyweight();
        unsharedFlyweight.StatefulOperation(--extrinsicstate);
    }
}

public abstract class FlyweightBase
{
    public abstract void StatefulOperation(object o);
}

public class FlyweightFactory
{
    private readonly Hashtable _flyweights = new Hashtable();

    public FlyweightBase GetFlyweight(string key)
    {
        if (_flyweights.Contains(key))
        {
            return _flyweights[key] as FlyweightBase;
        }
        var newFlyweight = new ConcreteFlyweight();

        _flyweights.Add(key, newFlyweight);
        return newFlyweight;
    }
}

public class ConcreteFlyweight : FlyweightBase
{
    public override void StatefulOperation(object o)
    {
        Console.WriteLine(o);
    }
}
```

```

public class UnsharedFlyweight : FlyweightBase
{
    private object _state;

    public override void StatefulOperation(object o)
    {
        _state = o;
        Console.WriteLine(_state);
    }
}

```

Real world example

In a war game example **UnitFactory** can create military units. In this example I have created two types of units: **Soldier** and **Tank**. These classes are concrete flyweights which inherit from Unit flyweight base class.

UnitFactory class contains a method called **GetUnit** which accepts one parameter that specifies the type of unit to create. This class has one private dictionary of unit types. When a new unit is required, the first step is to look into this dictionary whether this type of unit was created earlier. If yes, the program returns a reference to this unit. If no, it just creates a new unit and places it into the dictionary. You can control the number of instances of each unit type by the static field **NumberOfInstances**.

```

public class Soldier:Unit
{
    public static int NumberOfInstances;

    public Soldier()
    {
        NumberOfInstances++;
    }

    public override void FireAt(Target target)
    {
        Console.WriteLine("Shooting at unit {0} with power of {1}."
            , target.ID, FirePower);
    }
}

public class Tank:Unit
{
    public static int NumberOfInstances;

    public Tank()
    {
        NumberOfInstances++;
    }

    public override void FireAt(Target target)
    {
        Console.WriteLine("Firing at {0} with power of {1}.", target.ID, FirePower);
    }
}

public abstract class Unit
{
    public string Name { get; internal set; }
    public int Armour { get; internal set; }
    public int Speed { get; internal set; }
}

```

```
public int RotationRate { get; internal set; }
public int FireRate { get; internal set; }
public int Range { get; internal set; }
public int FirePower { get; internal set; }
public abstract void FireAt(Target target);
}

public class UnitFactory
{
    private readonly Dictionary<string, Unit> _units = new Dictionary<string, Unit>();

    public Unit GetUnit(string type)
    {
        if (_units.ContainsKey(type))
        {
            return _units[type];
        }
        Unit unit;

        switch (type)
        {
            case "Infantry":
                unit = new Soldier
                {
                    Name = "Standard Infantry",
                    Armour = 5,
                    Speed = 4,
                    RotationRate = 180,
                    FireRate = 5,
                    Range = 100,
                    FirePower = 5
                };
                break;

            case "Marine":
                unit = new Soldier
                {
                    Name = "Marine",
                    Armour = 25,
                    Speed = 4,
                    RotationRate = 180,
                    FireRate = 3,
                    Range = 200,
                    FirePower = 10
                };
                break;

            case "Tank":
                unit = new Tank
                {
                    Name = "Tank",
                    Armour = 1000,
                    Speed = 25,
                    RotationRate = 5,
                    FireRate = 30,
                    Range = 1000,
                    FirePower = 250
                };
                break;

            default:
                throw new ArgumentException();
        }
    }
}
```



```

    }

    _units.Add(type, unit);
    return unit;
}

public class Target
{
    public Unit UnitData;
    public Guid ID;
}

class Program
{
    static void Main(string[] args)
    {
        UnitFactory factory = new UnitFactory();

        Target tank1 = new Target();
        tank1.ID = Guid.NewGuid();
        tank1.UnitData = factory.GetUnit("Tank");

        Target tank2 = new Target();
        tank2.ID = Guid.NewGuid();
        tank2.UnitData = factory.GetUnit("Tank");

        bool result = tank1.UnitData == tank2.UnitData;    // result = true
        int firepower = tank1.UnitData.FirePower;

        Console.WriteLine("Tank Instances: " + Tank.NumberOfInstances);

        Target soldier1 = new Target();
        soldier1.ID = Guid.NewGuid();
        soldier1.UnitData = factory.GetUnit("Marine");

        var soldier2 = new Target();
        soldier2.UnitData = factory.GetUnit("Infantry");
        soldier2.ID = Guid.NewGuid();

        var soldier3 = new Target();
        soldier3.UnitData = factory.GetUnit("Infantry");
        soldier3.ID = Guid.NewGuid();

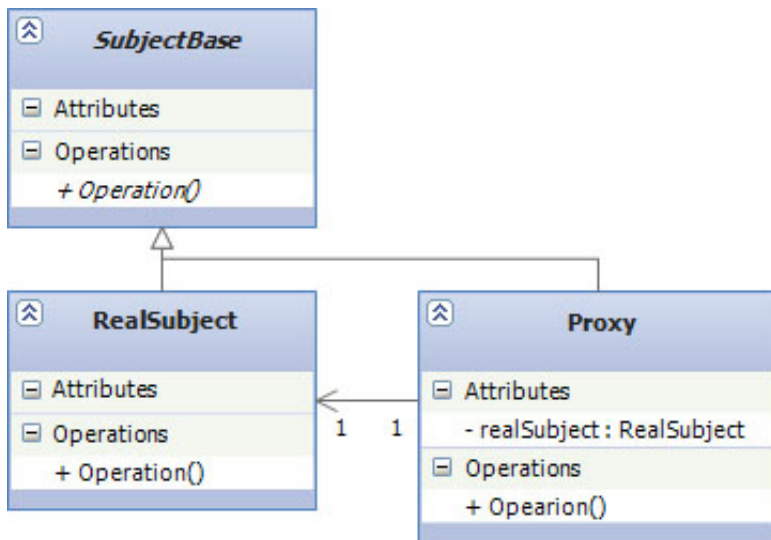
        Console.WriteLine("Soldier Instances: " + Soldier.NumberOfInstances);
    }
}

```

Proxy pattern

Proxy in a most general form is an interface to something else (Subject class). Proxy can be used when we don't want to access to the resource or subject directly because of some base of permissions or if we don't want to expose all of the methods of subject class. In some cases proxies can add some extra functionality. Using proxies is very helpful when we need to access resources which are hard to instantiate, are slow to execute or are resource sensitive.

Structural code example



The UML diagram below describes an implementation of the proxy design pattern. This diagram consists of three parts:

- **SubjectBase:** Interface (or abstract class) implemented by the **RealSubject** and representing its services. The interface must be implemented by the proxy as well so that the proxy can be used in any location where the **RealSubject** can be used.
- **RealSubject:** represents a complex class or resource sensitive class that we wish to use in more effective manner.
- **Proxy:** this is the key stone of this pattern. It references to **RealSubject** object. Application executes methods of proxy class which are transferred to RealSubject's methods.

```

static class Program
{
    static void Main()
    {
        var proxy = new Proxy();
        proxy.Operation();
    }
}

public abstract class SubjectBase
{
    public abstract void Operation();
}

public class RealSubject : SubjectBase
{
    public override void Operation()
    {
        Console.WriteLine("RealSubject.Operation");
    }
}

public class Proxy : SubjectBase
{
    private RealSubject _realSubject;

    public override void Operation()
    {
        if (_realSubject == null)
            _realSubject = new RealSubject();
    }
}
  
```

```

        _realSubject.Operation();
    }
}

```

Real world example

In this example I have created Order application which displays information of selected order. We have the **OrderRepositoryBase** class which is an abstract class for **ProxyOrderRepository** and **RealOrderRepository** classes. **RealOrderRepository** class is a RealSubject which we want to consume by our application using **ProxyOrderRepository**.

Also we have a set of entities which represents data of **RealOrderRepository** class. In this example Order can have a multiple **OrderDetails** but one Customer. This properties could be filled by methods of **RealObjectReporsitor** (**GetOrderDetails** and **GetOrderCustomer**). That means if we want to get all of the information about the order, we must call three different methods of **RealOrderRepository** class (**GetOrder**, **GetOrderDetails** and **GetOrderCustomer**). **ProxyOrderRepository** has all of this methods too. In case of **GetOrderDetails** and **GetOrderCustomer** it calls directly methods of **RealObjectRepository** class. In case of **GetOrder** the situation is a little bit different. It doesn't only call method of **RealObjectRepository** but it sets **OrderDetails** and **Customer** properties of **Order** returned by **GetOrder** method of **RealObjectRepository**. When all if the properties of **Order** object are populated, **Order** is returned to the caller application.

```

public class Customer
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Order
{
    public int Id { get; set; }
    public DateTime OrderDate { get; set; }
    public Customer Customer { get; set; }
    public IEnumerable<OrderDetail> Details { get; set; }
}

public class OrderDetail
{
    public int Id { get; set; }
    public int OrderId { get; set; }
    public string Description { get; set; }
}

public class ProxyOrderRepository:OrderRepositoryBase
{
    private RealOrderRepository _repository;

    public ProxyOrderRepository()
    {
        _repository = new RealOrderRepository();
    }

    public override Order GetOrder(int id)

```

```

    {
        var order = _repository.GetOrder(id);
        order.Customer = GetOrderCustomer(order.Id);
        order.Details = GetOrderDetails(order.Id);
        return order;
    }

    public override IEnumerable<OrderDetail> GetOrderDetails(int orderId)
    {
        return _repository.GetOrderDetails(orderId);
    }

    public override Customer GetOrderCustomer(int orderId)
    {
        return _repository.GetOrderCustomer(orderId);
    }
}

public class RealOrderRepository:OrderRepositoryBase
{
    private List<Order> _orders = new List<Order>
    {
        new Order(){Id=1, OrderDate = new
DateTime(2012,7,17)},
        new Order(){Id=2, OrderDate = new
DateTime(2012,6,16)}
    };

    private List<OrderDetail> _orderDetails = new List<OrderDetail>
    {
        new OrderDetail(){Description =
"Order Item #1",Id = 1,OrderId = 1},
        new OrderDetail(){Description =
"Order Item #2",Id = 2,OrderId = 1},
        new OrderDetail(){Description =
"Order Item #1",Id = 3,OrderId = 2}
    };

    private List<Customer> _customers = new List<Customer>()
    {
        new Customer(){FirstName = "John",LastName
= "Doe",Id=1},
        new Customer(){FirstName = "Jane",LastName
= "Doe",Id=2}
    };

    private Dictionary<int, int> _orderCustomers = new Dictionary<int, int>();

    public RealOrderRepository()
    {
        _orderCustomers.Add(1, 1);
        _orderCustomers.Add(2, 2);
    }

    public override Order GetOrder(int id)
    {
        var order = (from o in _orders where o.Id == id select o).SingleOrDefault();
        return order;
    }

    public override IEnumerable<OrderDetail> GetOrderDetails(int orderId)

```

```

    {
        var orderDetails =
            from o in _orderDetails
            where o.OrderId == orderId
            select o;
        return orderDetails;
    }

    public override Customer GetOrderCustomer(int orderId)
    {
        int cutomerId = _orderCustomers[orderId];
        var customer = (from c in _customers where c.Id == cutomerId select
c).SingleOrDefault();
        return customer;
    }
}

public abstract class OrderRepositoryBase
{
    public abstract Order GetOrder(int id);

    public abstract IEnumerable<OrderDetail> GetOrderDetails(int orderId);

    public abstract Customer GetOrderCustomer(int orderId);
}

class Program
{
    static void Main(string[] args)
    {
        ProxyOrderRepository repository = new ProxyOrderRepository();

        Order order = repository.GetOrder(1);

        Console.WriteLine("Order Id: {0}", order.Id );
        Console.WriteLine("Date: {0}", order.OrderDate);
        Console.WriteLine("Customer: {0}, {1}", order.Customer.LastName,
order.Customer.FirstName);
        Console.WriteLine("# of items: {0}", order.Details.Count());
    }
}

```

History

- 11 August - Original version posted

License

This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)

About the Author



Kanasz Robert

Architect The Staffing Edge &
Marwin Cassovia Soft
Slovakia 🇸🇰


My name is Robert Kanasz and I have been working with ASP.NET, WinForms and C# for several years.

MCTS - .NET Framework 3.5, ASP.NET Applications
- SQL Server 2008, Database Development
- SQL Server 2008, Implementation and Maintenance
- .NET Framework 4, Data Access
- .NET Framework 4, Service Communication Applications
- .NET Framework 4, Web Applications
MCPD - ASP.NET Developer 3.5
- Web Developer 4
MCITP - Database Administrator 2008
- Database Developer 2008

Open source projects: [DBScripter](#) - Library for scripting SQL Server database objects

Please, do not forget vote

Comments and Discussions

 **60 messages** have been posted for this article Visit <http://www.codeproject.com/Articles/438922/Design-Patterns-2-of-3-Structural-Design-Patterns> to post and view comments on this article, or click [here](#) to get a print view with messages.

[Permalink](#) | [Advertise](#) | [Privacy](#) | [Mobile](#)
Web04 | 2.6.130903.1 | Last Updated 16 Oct 2012

Article Copyright 2012 by Kanasz Robert
Everything else Copyright © [CodeProject](#), 1999-2013
[Terms of Use](#)