Facade pattern

From Wikipedia, the free encyclopedia

The **facade pattern** (or **façade pattern**) is a software design pattern commonly used with object-oriented programming. The name is by analogy to an architectural facade.

A facade is an object that provides a simplified interface to a larger body of code, such as a class library. A facade can:

- make a software library easier to use, understand and test, since the facade has convenient methods for common tasks;
- make the library more readable, for the same reason;
- reduce dependencies of outside code on the inner workings of a library, since most code uses the facade, thus allowing more flexibility in developing the system;
- wrap a poorly designed collection of APIs with a single well-designed API.

The Facade design pattern is often used when a system is very complex or difficult to understand because the system has a large number of interdependent classes or its source code is unavailable. This pattern hides the complexities of the larger system and provides a simpler interface to the client. It typically involves a single wrapper class which contains a set of members required by client. These members access the system on behalf of the facade client and hide the implementation details.

Contents

Usage

A Facade is used when an easier or simpler interface to an underlying object is desired.[1] Alternatively, an adapter can be used when the wrapper must respect a particular interface and must support polymorphic behavior. A decorator makes it possible to add or alter behavior of an interface at run-time.

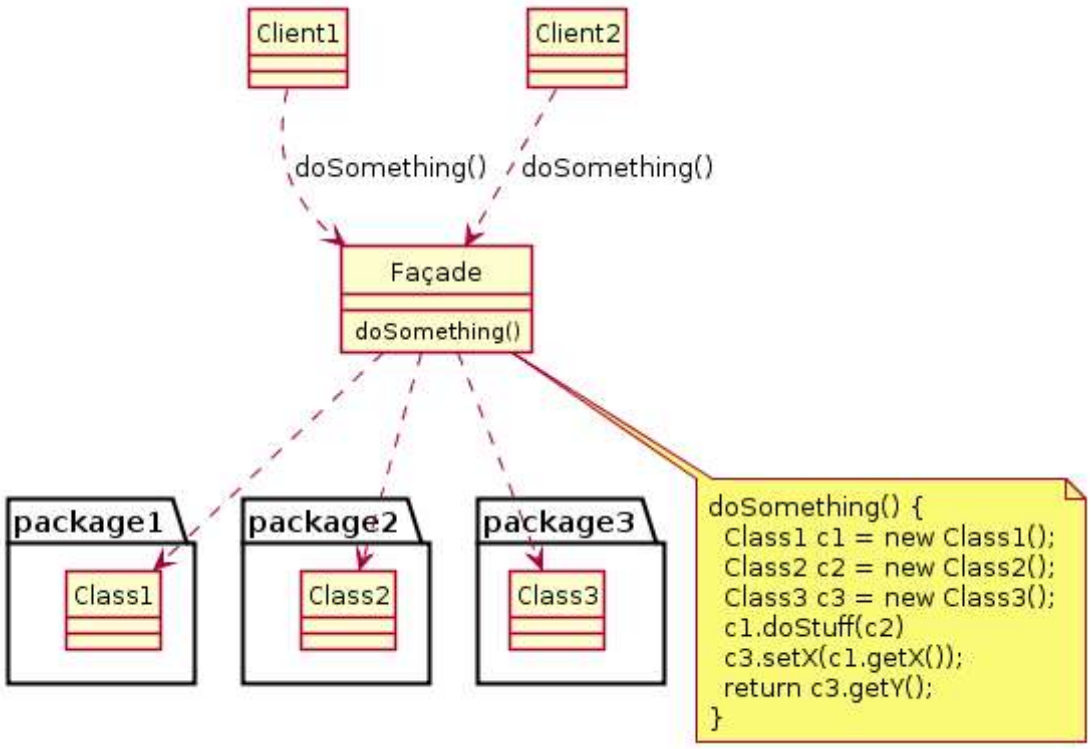
Pattern	Intent
Adapter	Converts one interface to another so that it matches what the client is expecting
Decorator	Dynamically adds responsibility to the interface by wrapping the original code
Facade	Provides a simplified interface

The facade pattern is typically used when:

- a simple interface is required to access a complex system;
- the abstractions and implementations of a subsystem are tightly coupled;

- need an entry point to each level of layered software; or
- a system is very complex or difficult to understand.

Structure



Facade
The facade class abstracts Packages 1, 2, and 3 from the rest of the application.
Clients
The objects are using the Facade Pattern to access resources from the Packages.

Example

This is an abstract example of how a client ("you") interacts with a facade (the "computer") to a complex system (internal computer parts, like CPU and HardDrive).

C#

Implementation

```
namespace Designpattern.Facade
{
    class SubsystemA
    {
        public string OperationA1()
        {
            return "Subsystem A, Method A1\n";
        }
        public string OperationA2()
        {
            return "Subsystem A, Method A2\n";
        }
    }

    class SubsystemB
    {
        public string OperationB1()
```

```
        {
            return "Subsystem B, Method B1\n";
        }

        public string OperationB2()
        {
            return "Subsystem B, Method B2\n";
        }
    }

    class SubsystemC
    {
        public string OperationC1()
        {
            return "Subsystem C, Method C1\n";
        }

        public string OperationC2()
        {
            return "Subsystem C, Method C2\n";
        }
    }

    public class Facade
    {
        private readonly SubsystemA a = new SubsystemA();
        private readonly SubsystemB b = new SubsystemB();
        private readonly SubsystemC c = new SubsystemC();
        public void Operation1()
        {
            Console.WriteLine("Operation 1\n" +
                a.OperationA1() +
                a.OperationA2() +
                b.OperationB1());
        }
        public void Operation2()
        {
            Console.WriteLine("Operation 2\n" +
                b.OperationB2() +
                c.OperationC1() +
                c.OperationC2());
        }
    }
}
```

Sample code

```
namespace DesignPattern.Facade.Sample
{
    // The 'Subsystem ClassA' class
    class CarModel
    {
        public void SetModel()
        {
            Console.WriteLine(" CarModel - SetModel");
        }
    }

    /// <summary>
    /// The 'Subsystem ClassB' class
    /// </summary>
    class CarEngine
    {
        public void SetEngine()
        {
            Console.WriteLine(" CarEngine - SetEngine");
        }
    }

    // The 'Subsystem ClassC' class
    class CarBody
```

```
    {
        public void SetBody()
        {
            Console.WriteLine(" CarBody - SetBody");
        }
    }

    // The 'Subsystem ClassD' class
    class CarAccessories
    {
        public void SetAccessories()
        {
            Console.WriteLine(" CarAccessories - SetAccessories");
        }
    }

    // The 'Facade' class
    public class CarFacade
    {
        private readonly CarModel model;
        private readonly CarEngine engine;
        private readonly CarBody body;
        private readonly CarAccessories accessories;

        public CarFacade()
        {
            model = new CarModel();
            engine = new CarEngine();
            body = new CarBody();
            accessories = new CarAccessories();
        }

        public void CreateCompleteCar()
        {
            Console.WriteLine("******** Creating a Car **********");
            model.SetModel();
            engine.SetEngine();
            body.SetBody();
            accessories.SetAccessories();

            Console.WriteLine("******** Car creation is completed. **********");
        }
    }

    // Facade pattern demo
    class Program
    {
        static void Main(string[] args)
        {
            var facade = new CarFacade();

            facade.CreateCompleteCar();

            Console.ReadKey();
        }
    }
}
```

Java

```
/* Complex parts */

class CPU {
    public void freeze() { ... }
    public void jump(long position) { ... }
    public void execute() { ... }
}

class Memory {
    public void load(long position, byte[] data) { ... }
}
```

```
class HardDrive {
    public byte[] read(long lba, int size) { ... }
}

/* Facade */

class ComputerFacade {
    private CPU processor;
    private Memory ram;
    private HardDrive hd;

    public ComputerFacade() {
        this.processor = new CPU();
        this.ram = new Memory();
        this.hd = new HardDrive();
    }

    public void start() {
        processor.freeze();
        ram.load(BOOT_ADDRESS, hd.read(BOOT_SECTOR, SECTOR_SIZE));
        processor.jump(BOOT_ADDRESS);
        processor.execute();
    }
}

/* Client */

class You {
    public static void main(String[] args) {
        ComputerFacade computer = new ComputerFacade();
        computer.start();
    }
}
```

Ruby

```
# Complex Parts
class CPU
  def freeze; end
  def jump(position); end
  def execute; end
end

class Memory
  def load(position, data); end
end

class HardDrive
  def read(lba, size); end
end

# Facade
class ComputerFacade

  def initialize
    @processor = CPU.new
    @ram = Memory.new
    @hd = HardDrive.new
  end

  def start
    @processor.freeze
    @ram.load(BOOT_ADDRESS, @hd.read(BOOT_SECTOR, SECTOR_SIZE))
    @processor.jump(BOOT_ADDRESS)
    @processor.execute
  end
end

# Client
computer_facade = ComputerFacade.new
```

```
computer_facade.start
```

References

1. Freeman, Eric; Freeman, Elisabeth; Sierra, Kathy; Bates, Bert (2004). Hendrickson, Mike; Loukides, Mike, eds. "Head First Design Patterns" (paperback) **1**. O'Reilly: 243, 252, 258, 260. ISBN 978-0-596-00712-6. Retrieved 2012-07-02.

External links

- Description from the Portland Pattern Repository (http://c2.com/cgi/wiki?FacadePattern)

The Wikibook *Computer Science Design Patterns* has a page on the topic of: ***Facade implementations in various languages***

Wikimedia Commons has media related to ***Facade pattern***.

Retrieved from "https://en.wikipedia.org/w/index.php?title=Facade_pattern&oldid=726785494"

Categories: Software design patterns
Articles with example Ruby code