**Design Patterns**

**What are deign patterns?**

*Design patterns are recurring solutions to software design problems you find again and again in real-world application development.* Patterns are about design and interaction of objects, as well as providing a communication platform concerning elegant, reusable solutions to commonly encountered programming challenges.

**What** **are the different categories of design patterns?**

*Creational Patterns:*The creational patterns abstract the object instantiation process by hiding how the objects are created and make the system independent of the object creation process (encapsulates object creation).

*Structural Patterns:*Describes how classes and objects can be combined to form larger structures. It eases the design by identifying a simple way to realize relationships between entities.

*Behavioral Patterns:*Defines a manner for controlling communication between classes or entities.

| **Creational Patterns** |  |
| --- | --- |
| [Abstract Factory](http://www.dofactory.com/Patterns/PatternAbstract.aspx) | Creates an instance of several families of classes |
| [Builder](http://www.dofactory.com/Patterns/PatternBuilder.aspx) | Separates object construction from its representation |
| [Factory Method](http://www.dofactory.com/Patterns/PatternFactory.aspx) | Creates an instance of several derived classes |
| [Prototype](http://www.dofactory.com/Patterns/PatternPrototype.aspx) | A fully initialized instance to be copied or cloned |
| [Singleton](http://www.dofactory.com/Patterns/PatternSingleton.aspx) | A class of which only a single instance can exist |

| **Structural Patterns** |  |
| --- | --- |
| [Adapter](http://www.dofactory.com/Patterns/PatternAdapter.aspx) | Match interfaces of different classes |
| [Bridge](http://www.dofactory.com/Patterns/PatternBridge.aspx) | Separates an object’s interface from its implementation |
| [Composite](http://www.dofactory.com/Patterns/PatternComposite.aspx) | A tree structure of simple and composite objects |
| [Decorator](http://www.dofactory.com/Patterns/PatternDecorator.aspx) | Add responsibilities to objects dynamically |
| [Facade](http://www.dofactory.com/Patterns/PatternFacade.aspx) | A single class that represents an entire subsystem |
| [Flyweight](http://www.dofactory.com/Patterns/PatternFlyweight.aspx) | A fine-grained instance used for efficient sharing |
| [Proxy](http://www.dofactory.com/Patterns/PatternProxy.aspx) | An object representing another object |

| **Behavioral Patterns** |  |
| --- | --- |
| [Chain of Resp.](http://www.dofactory.com/Patterns/PatternChain.aspx) | A way of passing a request between a chain of objects |
| [Command](http://www.dofactory.com/Patterns/PatternCommand.aspx) | Encapsulate a command request as an object |
| [Interpreter](http://www.dofactory.com/Patterns/PatternInterpreter.aspx) | A way to include language elements in a program |
| [Iterator](http://www.dofactory.com/Patterns/PatternIterator.aspx) | Sequentially access the elements of a collection |
| [Mediator](http://www.dofactory.com/Patterns/PatternMediator.aspx) | Defines simplified communication between classes |
| [Memento](http://www.dofactory.com/Patterns/PatternMemento.aspx) | Capture and restore an object's internal state |
| [Observer](http://www.dofactory.com/Patterns/PatternObserver.aspx) | A way of notifying change to a number of classes |
| [State](http://www.dofactory.com/Patterns/PatternState.aspx) | Alter an object's behavior when its state changes |
| [Strategy](http://www.dofactory.com/Patterns/PatternStrategy.aspx) | Encapsulates an algorithm inside a class |
| [Template Method](http://www.dofactory.com/Patterns/PatternTemplate.aspx) | Defer the exact steps of an algorithm to a subclass |
| [Visitor](http://www.dofactory.com/Patterns/PatternVisitor.aspx) | Defines a new operation to a class without change |

**Singleton**

**What is a singleton pattern? How do you code it in C#?AQ**

A singleton is a class that can be instantiated **only one time in a CLR per class loader**. Repeated calls always return the same instance. Ensures that a class has only one instance, and provide a **global point of access**.

Even though [*Singleton*](http://msdn.microsoft.com/en-us/library/ms998426.aspx) is a comparatively simple pattern, there are various tradeoffs and options, depending upon the implementation.

using System;

public class Singleton

{

private static Singleton instance;

private Singleton() {}

public static Singleton Instance

{

get

{

if (instance == null)

{

instance = new Singleton();

}

return instance;

}

}

}

Below are the advantages and disadvantages of this implementation.

*Advantage*

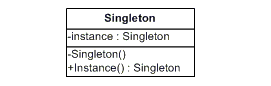
* Because the instance is created inside the **Instance** property method, the class can exercise additional functionality (for example, instantiating a subclass), even though it may introduce unwelcome dependencies.
* The instantiation is not performed until an object asks for an instance; this approach is referred to as *lazy instantiation*. Lazy instantiation avoids instantiating unnecessary singletons when the application starts.

*Disadvantage*

It is not safe for multithreaded environments. If separate threads of execution enter the **Instance** property method at the same time, more that one instance of the **Singleton** object may be created. Each thread could execute the following statement and decide that a new instance has to be created:

if (instance == null)

**Show the singleton in UML class diagram.**

****

### Participants

    The classes and/or objects participating in this pattern are:

* **Singleton**
  + defines an Instance operation that lets clients access its unique instance. Instance is a class operation.
  + responsible for creating and maintaining its own unique instance.

**How to make singleton a thread safe?**

There are two approaches in .NET:

* *Static initialization*
* *Double-Check Locking*

**Explain the static initialization approach to make singleton a thread safe.**

In this approach, the instance is created the first time any member of the class is referenced. The common language runtime takes care of the variable initialization. *The static initialization approach is possible because the .NET Framework explicitly defines how and when static variable initialization occurs.*

The class is marked **sealed** to prevent derivation, which could add instances. In addition, the variable is marked **readonly**, which means that it can be assigned only during static initialization (which is shown here) or in a class constructor.

public sealed class Singleton

{

private static readonly Singleton instance = new Singleton();

private Singleton(){}

public static Singleton Instance

{

get

{

return instance;

}

}

}

*Advantages*

* This implementation relies on *the common language runtime to initialize the variable which makes it different from thread unsafe implementation.* It still addresses the two basic problems that the *Singleton* pattern is trying to solve: *global access and instantiation control*. *The public static property provides a global access point to the instance.* Also, because the constructor is private, the **Singleton** class cannot be instantiated outside of the class itself; therefore, the variable refers to the only instance that can exist in the system.
* Because the **Singleton** instance is referenced by a private static member variable, the instantiation does not occur until the class is first referenced by a call to the **Instance** property. This solution therefore implements a form of the lazy instantiation property, as in the *Design Patterns* form of *Singleton*.

*Disadvantages*

The only potential downside of this approach is that you have less control over the mechanics of the instantiation. In the *Design Patterns* form, you were able to use a non default constructor or perform other tasks before the instantiation. Because the .NET Framework performs the initialization in this solution, you do not have these options. *In most cases, static initialization is the preferred approach for implementing a Singleton in .NET.*

**Explain the Double-Check Locking approach to make singleton a thread safe.**

The following implementation allows only a single thread to enter the critical area, which the **lock** block identifies, when no instance of **Singleton** has yet been created:

using System;

public sealed class Singleton

{

private static volatile Singleton instance;

private static object syncRoot = new Object();

private Singleton() {}

public static Singleton Instance

{

get

{

if (instance == null)

{

lock (syncRoot)

{

if (instance == null)

instance = new Singleton();

}

}

return instance;

}

}

}

This approach ensures that only one instance is created and only when the instance is needed. Also, the variable is declared to be **volatile** to ensure that assignment to the instance variable completes before the instance variable can be accessed. Lastly, this approach uses a **syncRoot** instance to lock on, rather than locking on the type itself, to avoid deadlocks.

This double-check locking approach solves the thread concurrency problems while avoiding an exclusive lock in every call to the **Instance** property method. It also allows you to delay instantiation until the object is first accessed. In practice, an application rarely requires this type of implementation. In most cases, the static initialization approach is sufficient.

**What are the benefits and liabilities of implementing Singleton in C#?**

Implementing *Singleton* in C# results in the following benefits and liabilities:

### *Benefits*

* The static initialization approach is possible because the .NET Framework explicitly defines how and when static variable initialization occurs.
* The *Double-Check Locking* idiom described earlier in "Multithreaded Singleton" is implemented correctly in the common language runtime.

### *Liabilities*

If your multithreaded application requires explicit initialization, you have to take precautions to avoid threading issues.

**How would you decide which approach you use to implement singleton for multithreaded application?**

There are times when your application must delay the instantiation, use a non-default constructor or perform other tasks before the instantiation, and work in a multithreaded environment, you need a different solution. Cases do exist, however, in which you cannot rely on the common language runtime to ensure thread safety, as in the Static Initialization example. In such cases, you must use specific language capabilities to ensure that only one instance of the object is created in the presence of multiple threads. One of the more common solutions is to use the *Double-Check Locking* idiom to keep separate threads from creating new instances of the singleton at the same time.

**When to use a singleton?**

You are building an application in C#. You need a class that has only one instance, and you need to provide a global point of access to the instance. You want to be sure that your solution is efficient and that it takes advantage of the Microsoft .NET common language runtime features. You may also want to make sure that your solution is thread safe.

Use it when only a single instance of an object is required in memory for a single point of

access. For example the following situations require a **single point of access**, which gets invoked from various parts of the code.

* Accessing application specific properties through a singleton object, which reads them for the first time from a properties file and subsequent accesses are returned from in-memory objects. Also there could be another piece of code, which periodically synchronizes the in-memory properties when the values get modified in the underlying properties file. This piece of code accesses the in-memory objects through the singleton object (i.e. global point of access).
* Accessing in-memory object cache or object pool, or non-memory based resource pools like sockets, connections etc through a singleton object (i.e. global point of access).

**What is the difference between a singleton class and a static class?**

Static class is one approach to make a class singleton by declaring all the methods as static so that you can’t create any instance of that class and can call the static methods directly.

**Discuss singleton pattern vs static class.**

Although aesthetically static classes and the singleton pattern appear to provide the same type

functionality, providing a mechanism for sharing of redundant objects in memory. There is

however differences in the implementation. There advantages and disadvantages to both these

solutions and it is entirely dependent on the situation in which you need to implement either. *A*

*big difference between Static classes and singletons is that Static classes cannot implement*

*interfaces and you cannot pass a static class as if it were just another implementation.*

*Advantages of Singletons*

* Singleton classes can inherit classes and implement interfaces
* Singleton classes can be initialized lazily or asynchronously.
* Singleton classes can be inherited and its methods can be overridden
* Singleton classes can be handled polymorphic ally without forcing their users to assume

that there is only one instance.

*You can use static class when:*

1) All its methods are utilities (nice example - class Math)

2) You don't want to deal with preserving your instance from garbage collector (in applets), but

I would better use singleton there.

3) You are absolutely sure that it wouldn't become stateful in the future and you are sure that

you will always need only one instance of that class

If you are using singleton and in one moment you realize that you need several instances then

your singleton easily can be transformed to multi one, but you'll have a problem with static

class. Static class is better for when you don't need to change the implementation. With a

singleton, you can have an interface with various implementations. A Static class, can only be

an implementation.

If your class doesn't store any state, then use a Static class. If it stores state and you require a

single instance, then (maybe) use a Singleton. Otherwise use a regular class.

**Give an example where you might use a static method? AQ**

Static methods prove useful for creating **utility classes, singleton classes** and **factory methods.**

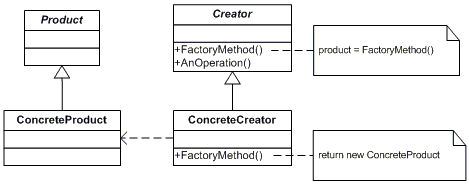
Utility classes are not meant to be instantiated. Improper coding of utility classes can lead to

procedural coding. [Math](http://msdn.microsoft.com/en-us/library/system.math(VS.80).aspx) class in the [System](http://msdn.microsoft.com/en-us/library/system(VS.80).aspx) namespace **is** an example of utility class in C#.

**Factory method**

**What is a factory pattern? FAQ**

A **Factory method pattern** (aka **Factory pattern**) is a creational pattern. An **Abstract factory** pattern is one level of abstraction higher than a factory method pattern, which means it *returns the factory classes.*

**

**What are factories responsible for?**

Factories are responsible for creating objects and ensuring objects are available to be used.

**Give an example to show how factory pattern works.**

We will be creating dog objects. As with any good OO design we start with an interface for our related objects.   
  
The IDog interface   
  
**Code:**

    public interface IDog //product  
    {   
        void Bark();   
        void Scratch();   
    }

We just define a couple of simple methods for our dogs to do.   
  
    Now for the two actual concrete dog classes. We define a bulldog and poodle class:   
  
**Code:**

    public class CPoodle : IDog //concrete product  
    {   
        public CPoodle()   
        {   
            Console.WriteLine("Creating Poodle");   
        }   
        public void Bark()   
        {   
            Console.WriteLine("Yip Yip");   
        }   
        public void Scratch()   
        {   
            Console.WriteLine("Scratch Scratch");   
        }   
    }   
  
    public class CBullDog : IDog //concrete product  
    {   
        public CBullDog()   
        {   
            Console.WriteLine("Creating Bulldog");   
        }   
        public void Bark()   
        {   
            Console.WriteLine("Wooof Wooof");   
        }   
        public void Scratch()   
        {   
            Console.WriteLine("Scratch Slobber Scratch");   
        }   
    }

    Now for our factory class. It’s static and contains one method to return the correct dog.   
  
**Code:**

    public class CDogFactory   
    {   
        public enum DogType   
        {   
            Poodle, Bulldog   
        }   
        static CDogFactory()   
        {   
        }   
        public static IDog CreateDog(DogType TypeOfDog)   
        {   
            switch (TypeOfDog)   
            {   
                case DogType.Bulldog:   
                    return new CBullDog();   
                case DogType.Poodle:   
                    return new CPoodle();   
                default:   
                    throw new ArgumentException("Invalid Dog Type!");   
            }   
        }   
    }

We make the class static so we don’t need an instance of the class.   
  
To test the class, I have created a simple function. Our test function just return the dogs and uses them. In a more real world app, the type of dog would have been determined by the user or through program logic.   
  
**Code:**

        IDog dog1;   
        IDog dog2;   
  
        dog1 = CDogFactory.CreateDog(CDogFactory.DogType.Bulldog);   
        dog2 = CDogFactory.CreateDog(CDogFactory.DogType.Poodle);   
  
        dog1.Bark();   
        dog1.Scratch();   
  
        dog2.Bark();   
        dog2.Scratch();

**When one should use factory method pattern?**

Use the Factory Method pattern in any of the following situations:

* A class can't anticipate the class of objects it must create
* A class wants its subclasses to specify the objects it creates

**What is the essential reason to use a Factory Method?**

You want to defer the decision for instantiating a derivation of another class to a derived class

**The Factory Method pattern has been implemented in all of the major object-oriented languages. How has it been used in Java, C#, and C++?**

In Java, the iterator method on collections is a Factory Method.

In C#, the GetEnumerator is a Factory method on the differen C# collections where it is present. In C++, the methods used include begin() and end().

**Why is this pattern called a "factory method?"**

It uses a method to handle the factory

**How does the Factory Method pattern fit in with other factories?**

The Factory Method allows these other patterns to defer instantiation to subclasses. One uses the Factory Method to defer responsibility to subclass objects. The Abstract Factory can use a family of Factory Methods, one for each different family of objects involved. The Template Method can use a method to handle the instantiation; giving responsibility to that method is the factory.

**What exactly does it mean when we say that "the Factory Method Pattern lets subclasses decide which class to instantiate?"**

* It means that Creator class is written without knowing what actual ConcreteProduct class will be instantiated. The ConcreteProduct class which is instantiated is determined solely by which ConcreteCreator subclass is instantiated and used by the application.
* It does not mean that somehow the subclass decides at runtime which ConreteProduct class to create

**The Gang of Four says that the intent of the Factory Method is to "define an interface for creating an object, but let subclasses decide which class to instantiate." Why is this important?**

It is not always desirable for a class to have to know how to instantiate derived classes.

**What are the benefits and liabilities of using factory method?**

Benefits

* Code is made more flexible and reusable by the elimination of instantiation of application-specific classes
* Code deals only with the interface of the Product class and can work with any ConcreteProduct class that supports this interface

Liabilities

Clients might have to subclass the Creator class just to instantiate a particular ConcreteProduct

| | **What is object pooling?**    Object Pool is nothing but a container of objects that are ready for use. Whenever there is a request for a new object, the pool manager will take the request and it will be served by allocating an object from the pool.    **How do you implement object pooling? Give an example so that maximum number of objects in the pool is 2.**    We are going to use Factory pattern for this purpose. We will have a factory method, which will take care about the creation of objects. Whenever there is a request for a new object, the factory method will look into the object pool (we use Queue object). If there is any object available within the allowed limit, it will return the object (value object), otherwise a new object will be created and give you back.    The below code is just an example to give you an idea, and is neither tested nor error-proof. You can modify it as you wish: up-to your creativity.    **Code 1:** Object Pool and Employee class.    using System;  using System.Collections;    namespace ObjectPooling  {  class Factory  {  // Maximum objects allowed!  private static int \_PoolMaxSize = 2;    // My Collection Pool  private static readonly Queue objPool = new Queue(\_PoolMaxSize);    public Employee GetEmployee()  {  Employee oEmployee;    // check from the collection pool. If exists return object else create new  if (Employee.Counter >= \_PoolMaxSize && objPool.Count>0)  {  // Retrieve from pool  oEmployee = RetrieveFromPool();  }  else  {  oEmployee = GetNewEmployee();  }  return oEmployee;  }    private Employee GetNewEmployee()  {  // Creates a new employee  Employee oEmp = new Employee();  objPool.Enqueue(oEmp);  return oEmp;  }    protected Employee RetrieveFromPool()  {  Employee oEmp;    // if there is any objects in my collection  if (objPool.Count>0)  {  oEmp = (Employee)objPool.Dequeue();  Employee.Counter--;  }  else  {  // return a new object  oEmp = new Employee();  }  return oEmp;  }  }    class Employee  {  public static int Counter = 0;  public Employee()  {  ++Counter;  }    private string \_Firstname;  public string Firstname  {  get  {  return \_Firstname;  }  set  {  \_Firstname = value;  }  }  }  }    **Code 2:** How to use it?    private void button1\_Click(object sender, EventArgs e)  {  Factory fa = new Factory();    Employee myEmp = fa.GetEmployee();  Console.WriteLine("First object");    Employee myEmp1 = fa.GetEmployee();  Console.WriteLine("Second object")  } | | --- | |
| --- | --- |
|  |
| **Why is an Object Pool required?** |
| The biggest advantage of using Object Pooling is that it minimizes the consumption of memory and the system's resources by recycling and re-using objects as and when it is needed and serving the request for new objects from the pool of ready-to-be-used objects. The objects that the application is done with (the objects are no longer needed) are sent back to the pool rather than destroying them from the memory. According to MSDN, "Once an application is up and running, memory utilization is affected by the number and size of objects the system requires. Object pooling reduces the number of allocations, and therefore the number of garbage collections, required by an application. Pooling is quite simple: an object is reused instead of allowing it to be reclaimed by the garbage collector. Objects are stored in some type of list or array called the pool, and handed out to the client on request. This is especially useful when an instance of an object is repeatedly used, or if the object has an expensive initialization aspect to its construction such that it's better to reuse an existing instance than to dispose of an existing one and to create a completely new one from scratch." |
| **How does an Object Pool work?** |
| When an object is requested, it is served from the pool. When the object is disposed, it is placed back into the pool to await the next request that might come in at a later point in time. The pool initially consists of a number of objects. When a request for creation of an object comes in, the request is server from the pool of objects and the number of the available objects in the pool decreases by one. This process continues until the pool runs out of objects. The pool remains in memory as long as there is at least one object in the pool. The pool facilitates reusability and eliminates the overhead involved in creation of objects whenever they are requested. The following section discusses how an Object Pool (though somewhat similar to a Connection Pool) differs from a Connection Pool. |
| **How does Object Pooling and Connection Pooling differ?** |
| There are distinct differences between Object pooling and Connection Pooling. Object Pooling is great in the sense that it can optimize access to expensive resources (like file handles or network connections) by pooling them in memory and reusing them as and when they are needed. According to MSDN, *"Object pooling lets you control the number of connections you use, as opposed to connection pooling, where you control the maximum number reached."* |

| **What is connection pooling?** |
| --- |
| Connecting to the database is resource intensive and a relatively slow operation in an application but the most crucial of them all. *A Connection Pool is a container of open and reusable connections.* *A Connection Pool is released from the memory when the last connection to the database is closed.* The basic advantage of using Connection Pooling is an improvement of performance and scalability while the main disadvantage is that one or more database connections, even if they are currently not being used, are kept open. The Data Providers in ADO.NET have Connection Pooling turned on by default; if you need to turn it off, specify Pooling = false in the connection string being used. Connection Pooling gives you an idle, open, reusable connection instead of opening a new one every time a connection request to the database is made. When the connection is closed or disposed, it is returned to the pool and remains idle until a request for a new connection comes in. If we use Connection Pooling efficiently, opening and closing of connections to the database becomes less resource expensive. |
| **How does a Connection Pool work?** |
| Connection pools are actually containers that contain open and reusable connections. Multiple pools can exist in the same application domain at the same point in time, but Connection Pools cannot be shared across application domains. Note that one pool is created per unique connection string. A Connection Pool is created the first time a request for a connection to the database comes in with a unique connection string. Note that if another request comes in with a different connection string for the database, another Connection Pool would be created. Hence, we have one Connection Pool per connection string and not per database. The following code listings below illustrate this.  **Listing 1**  // A new pool is created.  SqlConnection sqlConnection = new SqlConnection();  sqlConnection.ConnectionString =  "Server=localhost;Database=test;User ID=joydip;Password=joydip;Trusted\_Connection=False";  sqlConnection.Open();    **Listing 2**  // A new pool is created as the connection strings differ.  SqlConnection conn = new SqlConnection();  sqlConnection.ConnectionString =  "Server=localhost;Database=test;User ID=test;Password=test;Trusted\_Connection=False";  sqlConnection.Open();  **Listing 3**  // The connection string is the same as in Listing 1 so no new pool is created.  SqlConnection conn = new SqlConnection();  sqlConnection.ConnectionString =  "Server=localhost;Database=test;User ID=joydip;Password=joydip;Trusted\_Connection=False";  sqlConnection.Open();    When a request for a new connection comes in the connection is served from the Connection Pool without creating a new one, i.e., the connections are re-used without creating new ones.  Therefore, it improves the performance and scalability of your applications. When your application closes an open connection, it is returned to the pool where it waits until a reconnect time out period expires. This is the period within which it waits to connect to the same database using the same credentials. If none comes in within this period, the connection to the database is closed and the connection instance is removed from the pool.  When a new Connection Pool is created, the connection objects are placed in the pool and the pool and the connections contained in it are made available for immediate use. The pool can house connections up to the maximum limit as specified in the connection string that was used to connect to the database. Connections are removed from the pool when they remain inactive for a long period of time or have outlived its specified life time or server connectivity.  A Connection Pool is maintained internally by the Connection Pool Manager. When a request for a subsequent connection comes in, the Connection Pool Manager searches the pool for the availability of a free connection and returns it to the application if one is available. The following points elaborate how the Connection Pool Manager works- the operations that it performs when a request for a new connection comes to it.   * If any unused connection is available, it returns one (unused connection). * If all connections are used up, a new connection is created and added to the pool. * If the number of connections reaches the maximum number of connections in the pool, the requests are queued until a connection becomes free for reuse.   Connection Pooling is controlled and the parameters passed to a connection string that basically comprises the following:   * Connect Timeout * Min Pool Size * Max Pool Size * Pooling   In order to use Connection Pooling efficiently, remember to close the connections once you are done with it so that it returns to the pool. |
| **Improving Connection Pooling Performance** |
| We should always open the connections late and release them early; in other words, immediately after we are done using it. Connections should be opened only at the time when they are actually required.  The connections should not be acquired prior to its usage as it would decrease the number of available connections in the Connection Pool and, hence, have detrimental effects to the operation of the Connection Pool and the application's performance. The connections should be explicitly released immediately when we are done using it. This would facilitate better Connection Pooling as the connection would be returned to the pool and be available for reuse. The following snippet of code shows how we should open and close connections in our applications efficiently.  **Listing 4**  SqlConnection sqlConnection = new SqlConnection(connectionString);  try  {    sqlConnection.Open();    //Some Code  }    finally  {    sqlConnection.Close();  }  The code in Listing 4 can be simplified further by using the "using" keyword as shown in the code snippet below.  **Listing 5**  using(SqlConnection sqlConnection = new SqlConnection(connectionString))  {    sqlConnection.Open();    //Some Code  }  Note that using the "using" keyword as shown in Listing V above, generates the try-finally block implicitly.  The following points illustrate the measures that can be taken for better utilization of the Connection Pool.   * Always open connections when needed and close it immediately when you are done using it. * Close the user-defined transactions before closing the related connections. * Ensure that there is at least one connection open in the pool to maintain the Connection Pool. * Avoid using connection pooling if integrated security is being used.   Connection Pooling can be monitored in one of the following ways:   * Executing the sp\_who or sp\_who2 stored procedures * Using the SQL Server Profiler *          Using the Performance Monitor and Performance Counters |

**Factory method pattern (aka Factory pattern) Abstract factory pattern**

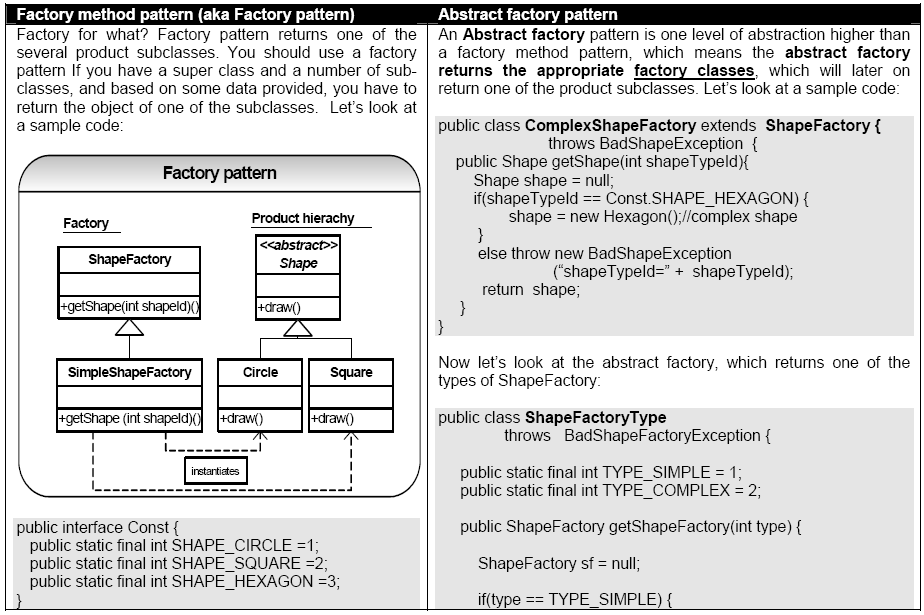
**Abstract factory**

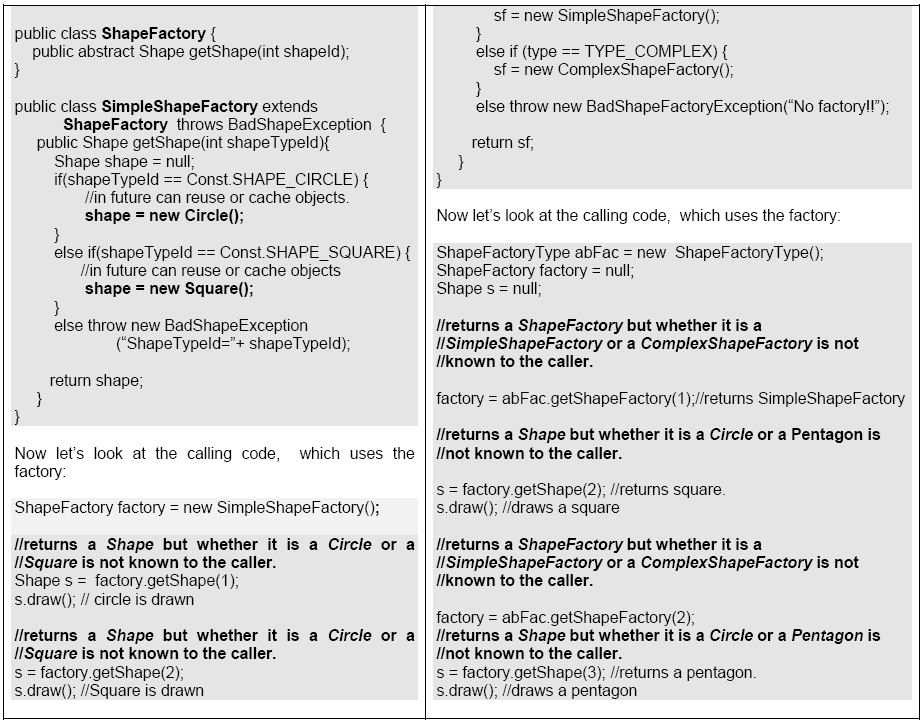
**Draw the structure of abstract factory pattern.**

Factory for what? Factory pattern returns one of the several product subclasses. You should use a factory pattern If you have a super class and a number of subclasses, and based on some data provided, you have to return the object of one of the subclasses. Let’s look at a sample code:

**What are the differences between factory method and abstract factory pattern?**

| **Factory method** | **Abstract factory** |
| --- | --- |
| All creators needs to be sub-classed | Only one class is need to be sub-classed. |
| Factory method has only one creator | Abstract factory method has multiple creators |
| Scope of the Factory method is class level i.e. factory method is implemented using the sub-classing (inheritance). Factory Method pattern uses inheritance and relies on a subclass to handle the desired object instantiation. | Abstract Factory method uses delegation. A  class delegates the responsibility of object instantiation to another object via composition |
| Factory method is a way to get one individual member of a family. | Abstract factory is a creational pattern for the family of objects. Hence Abstract Factory makes use of Factory Methods. The delegated object frequently uses factory methods to perform the instantiation |





**Why use factory pattern or abstract factory pattern?**

Factory pattern returns an instance of several (product hierarchy) subclasses (like ***Circle, Square*** etc), but the calling code is unaware of the actual implementation class.

The calling code invokes the method on the interface for example ***Shape*** and using polymorphism the correct draw() method gets invoked [Refer **Q10** in Java section for polymorphism]. So, as you can see, the factory pattern reduces the coupling or the dependencies between the calling code and called objects like *Circle*, *Square* etc. This is a very powerful and common feature in many frameworks. You do not have to create a new *Circle* or a new *Square* on each invocation as shown in the sample code, which is for the purpose of illustration and simplicity. In future, to conserve memory you can decide to cache objects or reuse objects in your factory with no changes required to your calling code. You can also load objects in your factory based on attribute(s) read from an external properties file or some other condition. Another benefit going for the factory is that unlike calling constructors directly, factory patterns have more meaningful names like getShape(…), getInstance(…) etc, which may make calling code more clear.

**Can we use the singleton pattern within our factory pattern code?**

Yes. Another important aspect to consider when writing your factory class is that, it does not make sense to create a new factory object for each invocation as it is shown in the sample code, which is just fine for the illustration purpose.

ShapeFactory factory = new SimpleShapeFactory();

To overcome this, you can incorporate the singleton design pattern into your factory pattern code. The singleton design pattern will create only a single instance of your SimpleShapeFactory class. Since an abstract factory pattern is unlike factory pattern, where you need to have an instance for each of the two factories (i.e.SimpleShapeFactory and ComplexShapeFactory) returned, you can still incorporate the singleton pattern as an access point and have an instance of a HashMap, store your instances of both factories. Now your calling method uses a static method to get the same instance of your factory, hence conserving memory and promoting object reuse:

ShapeFactory factory = ShapeFactory.getFactoryInstance();//returns a singleton

factory.getShape();

**While using "switches" can be a reasonable solution to a problem that requires choosing among alternatives, it caused problems for the driver problem discussed in this chapter. What were these problems? What might a switch indicate the need for?** The rules for determining which driver to use are intermixed with the actual use of the drivers. This creates both tight coupling and strong cohesion. (p. 165)

Switches may indicate a need for abstraction (p. 166)

**Why is this pattern called "Abstract Factory"?**

At first glance, you might be tempted to conclude it is because the factory is implemented as an abstract class with a derivation for each case. But that is not the case. This pattern is called the "Abstract Factory" because the things it is intended to build are themselves defined by abstractions. How you choose to implement the factory variations is not specific to the pattern.

**What are the three key strategies in the Abstract Factory?**

Find what varies and encapsulate it

Favor aggregation over inheritance

Design to interfaces, not to implementations.

**In this pattern, there are two kinds of factories. What does the "Abstract Factory" class do? What do the "concrete factory" classes do?**

The "Abstract Factory" class specifies which objects can be instantiated by defining a method for each type of object.

The "concrete factory" classes specify which objects are to be instantiated.

**What are the consequences of the Abstract Factory pattern?**

The Abstract Factory isolates the rules about which objects to use from the logic about how to use these objects.

**Interpretations**

**1. The Gang of Four says that the intent of the Abstract Factory pattern is to "provide an interface for creating families of related or dependent objects without specifying their concrete classes." What does this mean? Give an example.**

It means that I need to coordinate the instantiation of several objects, a family of objects. However, I want to insulate my system from having to know specifics of the particular concrete object being instantiated. That is, the selection of which particular concrete instance to use might depend upon another factor. An example would be a system that wants to manage records in a database but be insulated from the specifics of which DBMS is being used. (p. 163)

**Opinions and Applications**

**1. Why do you think the Gang of Four call this pattern "Abstract Factory"? Is it an appropriate name for what it is doing? Why or why not?**

**2. How do you know when to use the Abstract Factory pattern?**

# Strategy Design Pattern

**What is the Strategy Pattern?**

The strategy pattern is a Gang of Four design pattern. This is a *behavioral* pattern as it defines a manner for controlling communication between classes or entities. The strategy pattern is used to create an interchangeable family of algorithms from which the required process is chosen at run-time. This allows the behavior of a program to change dynamically according to configuration details or user preferences. It also increases flexibility by allowing new algorithms to be easily incorporated in the future.

**What is the intent of the Strategy pattern?**

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.

**What are the consequences of the Strategy pattern?**

The Strategy pattern defines a family of algorithms.

Switches and/or conditionals can be eliminated. You must invoke all algorithms in the same way (they must all have the same interface).

**What are some alternatives for handling new requirements?**

Cut and paste

Switches or ifs on a variable specifying the case we have Using function pointers or delegates (a different one representing each case) Inheritance (make a derived class that does it the new way) Design patterns

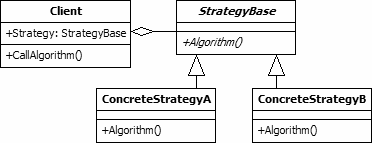
**What are the three fundamental principles proposed by the Gang of Four that guide how to anticipate change?**

Program to an interface, not an implementation.

Favor object aggregation over class inheritance.

Consider what should be variable in your design.

**Draw the UML class diagram for the Strategy Pattern**

****

The UML class diagram above describes an implementation of the strategy design pattern. The items in the diagram are described below:

* **Client**. This class is the user of an interchangeable algorithm. The class includes a property to hold one of the strategy classes. This property will be set at run-time according to the algorithm that is required.
* **StrategyBase**. This abstract class is the base class for all classes that provide algorithms. In the diagram the class includes a single method. However, there is no reason why a number of properties and methods may not be included. This class may be implemented as an interface if it provides no real functionality for its subclasses.
* **ConcreteStrategy A/B**. The concrete strategy classes inherit from the StrategyBase class. Each provides a different algorithm that may be used by the client.

**Give an example from real world where this pattern can be used?**

This real-world code demonstrates the Strategy pattern which encapsulates sorting algorithms in the form of sorting objects. This allows clients to dynamically change sorting strategies including Quicksort, Shellsort, and Mergesort.

| // Strategy pattern -- Real World example |
| --- |
| using System;  using System.Collections.Generic;    namespace DoFactory.GangOfFour.Strategy.RealWorld  {    /// <summary>    /// MainApp startup class for Real-World    /// Strategy Design Pattern.    /// </summary>    class MainApp    {      /// <summary>      /// Entry point into console application.      /// </summary>      static void Main()      {        // Two contexts following different strategies        SortedList studentRecords = new SortedList();          studentRecords.Add("Samual");        studentRecords.Add("Jimmy");        studentRecords.Add("Sandra");        studentRecords.Add("Vivek");        studentRecords.Add("Anna");          studentRecords.SetSortStrategy(new QuickSort());        studentRecords.Sort();          studentRecords.SetSortStrategy(new ShellSort());        studentRecords.Sort();          studentRecords.SetSortStrategy(new MergeSort());        studentRecords.Sort();          // Wait for user        Console.ReadKey();      }    }      /// <summary>    /// The 'Strategy' abstract class    /// </summary>    abstract class SortStrategy    {      public abstract void Sort(List<string> list);    }      /// <summary>    /// A 'ConcreteStrategy' class    /// </summary>    class QuickSort : SortStrategy    {      public override void Sort(List<string> list)      {        list.Sort(); // Default is Quicksort        Console.WriteLine("QuickSorted list ");      }    }      /// <summary>    /// A 'ConcreteStrategy' class    /// </summary>    class ShellSort : SortStrategy    {      public override void Sort(List<string> list)      {        //list.ShellSort(); not-implemented        Console.WriteLine("ShellSorted list ");      }    }      /// <summary>    /// A 'ConcreteStrategy' class    /// </summary>    class MergeSort : SortStrategy    {      public override void Sort(List<string> list)      {        //list.MergeSort(); not-implemented        Console.WriteLine("MergeSorted list ");      }    }      /// <summary>    /// The 'Context' class    /// </summary>    class SortedList    {      private List<string> \_list = new List<string>();      private SortStrategy \_sortstrategy;        public void SetSortStrategy(SortStrategy sortstrategy)      {        this.\_sortstrategy = sortstrategy;      }        public void Add(string name)      {        \_list.Add(name);      }        public void Sort()      {        \_sortstrategy.Sort(\_list);          // Iterate over list and display results        foreach (string name in \_list)        {          Console.WriteLine(" " + name);        }        Console.WriteLine();      }    }  } |
| Output  QuickSorted list  Anna  Jimmy  Samual  Sandra  Vivek  ShellSorted list  Anna  Jimmy  Samual  Sandra  Vivek  MergeSorted list  Anna  Jimmy  Samual  Sandra  Vivek |

**How is it implemented?**

Let’s take a look at implementing a simple application that serializes data into different formats (such as XML, JSON). We will first look at a code sample that does not use the Strategy Pattern. We will later rewrite to use the Strategy Pattern to understand the benefits.

It’s pretty common experience for any developer to be faced with situations which require him/her to execute an algorithm based on a variable. In the snippet below, based on the serializeFormat variable, we add logic for serialization.

* 1. class Program
  2. {
  3. static void Main(string[] args)
  4. {
  5. string serializeFormat = "XML";
  7. switch (serializeFormat)
  8. {
  9. case "XML":
  10. //Logic to serialize a data source to XML
  11. Console.WriteLine("Serializing to XML");
  12. break;
  13. case "JSON":
  14. //Logic to serialize a data source to JSON
  15. Console.WriteLine("Serializing to JSON");
  16. break;
  17. default:
  18. break;
  19. }
  20. }
  21. } class Program

{

static void Main(string[] args)

{

string serializeFormat = "XML";

switch (serializeFormat)

{

case "XML":

//Logic to serialize a data source to XML

Console.WriteLine("Serializing to XML");

break;

case "JSON":

//Logic to serialize a data source to JSON

Console.WriteLine("Serializing to JSON");

break;

default:

break;

}

}

}

Let’s rewrite this code using the Strategy Pattern. Let’s take a look at the code first.

1. namespace StrategyPattern
2. {
3. class Program
4. {
5. static void Main(string[] args)
6. {
7. SerializeContext context = new SerializeContext(new XmlSerializerStrategy());
9. context.Serialize();
10. }
11. }
12. }

namespace StrategyPattern

{

class Program

{

static void Main(string[] args)

{

SerializeContext context = new SerializeContext(new XmlSerializerStrategy());

context.Serialize();

}

}

}

***The “Client” implementation***

1. namespace StrategyPattern
2. {
3. public interface ISerializeStrategy
4. {
5. void Serialize();
6. }
8. public class XmlSerializerStrategy : ISerializeStrategy
9. {
10. public void Serialize()
11. {
12. //Logic to serialize to Xml
13. Console.WriteLine("Serializing to Xml");
14. }
15. }
17. public class JsonSerializerStrategy : ISerializeStrategy
18. {
19. public void Serialize()
20. {
21. //Logic to serialize to Xml
22. Console.WriteLine("Serializing to Json");
23. }
24. }
26. public class SerializeContext
27. {
28. private ISerializeStrategy serialize;
30. public SerializeContext(ISerializeStrategy serialize)
31. {
32. this.serialize = serialize;
33. }
35. public void Serialize()
36. {
37. this.serialize.Serialize();
38. }
39. }
40. }

namespace StrategyPattern

{

public interface ISerializeStrategy

{

void Serialize();

}

public class XmlSerializerStrategy : ISerializeStrategy

{

public void Serialize()

{

//Logic to serialize to Xml

Console.WriteLine("Serializing to Xml");

}

}

public class JsonSerializerStrategy : ISerializeStrategy

{

public void Serialize()

{

//Logic to serialize to Xml

Console.WriteLine("Serializing to Json");

}

}

public class SerializeContext

{

private ISerializeStrategy serialize;

public SerializeContext(ISerializeStrategy serialize)

{

this.serialize = serialize;

}

public void Serialize()

{

this.serialize.Serialize();

}

}

}

***Implementation of Strategies, Context and the Interface***

The Strategy Pattern consists of 3 distinct components:

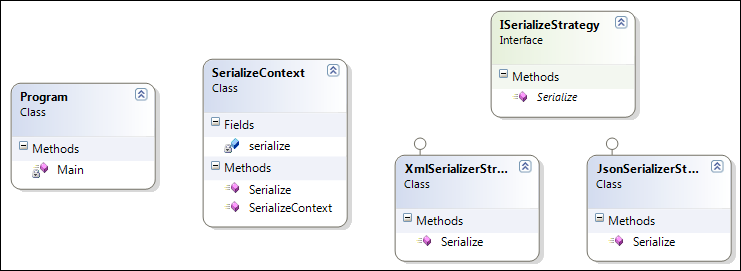
1. A client (In our example, the console application’s Program.cs class)

2. An Interface that all strategies implement. In our example, the ISerializeStrategy interface.

3. Different algorithm classes (or Concrete Strategies) that implement the interface. In our example, the classes XmlSerializerStrategy and JsonSerializerStrategy

4. A Context class. The Context object receives requests from the client and delegates them to the appropriate Concrete Strategies. Note that the client only interacts with the context class. The client does not instantiate the concrete class objects directly. Technically, the Context class contains a reference to the Concrete Strategies. In our example, this is done by declaring a private member of type ISerializeStrategy

Here’s the class diagram for the above implementation.



As the application grows, you may need to support other types of Serialization as well (say Binary Serializer). All you need to do is create a new BinarySerializerStrategy class that implements the ISerializeStrategy interface and add implementation to the Serialize() method.

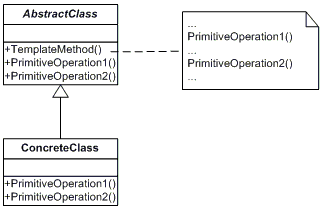
It’s important to note that the Strategy Pattern is a classic example of Aggregation at work. Aggregation is a OOP concept of having a “has a” relationship between the aggregate and the component(as opposed to Inheritence that has a “is a” relationship)

**Template Method**

**Define template pattern.**

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. This is a *behavioral* pattern. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

**Draw the UML class diagram.**

****

The classes and/or objects participating in this pattern are:

* **AbstractClass**
  + defines abstract *primitive operations* that concrete subclasses define to implement steps of an algorithm
  + implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.
* **ConcreteClass**
  + implements the primitive operations to carry out subclass-specific steps of the algorithm

**Give an example of code showing template pattern.**

| // Template Method pattern -- Structural example |
| --- |
| using System;    namespace DoFactory.GangOfFour.Template.Structural  {    /// <summary>    /// MainApp startup class for Real-World    /// Template Design Pattern.    /// </summary>    class MainApp    {      /// <summary>      /// Entry point into console application.      /// </summary>      static void Main()      {        AbstractClass aA = new ConcreteClassA();        aA.TemplateMethod();          AbstractClass aB = new ConcreteClassB();        aB.TemplateMethod();          // Wait for user        Console.ReadKey();      }    }      /// <summary>    /// The 'AbstractClass' abstract class    /// </summary>    abstract class AbstractClass    {      public abstract void PrimitiveOperation1();      public abstract void PrimitiveOperation2();        // The "Template method"      public void TemplateMethod()      {        PrimitiveOperation1();        PrimitiveOperation2();        Console.WriteLine("");      }    }      /// <summary>    /// A 'ConcreteClass' class    /// </summary>    class ConcreteClassA : AbstractClass    {      public override void PrimitiveOperation1()      {        Console.WriteLine("ConcreteClassA.PrimitiveOperation1()");      }      public override void PrimitiveOperation2()      {        Console.WriteLine("ConcreteClassA.PrimitiveOperation2()");      }    }      /// <summary>    /// A 'ConcreteClass' class    /// </summary>    class ConcreteClassB : AbstractClass    {      public override void PrimitiveOperation1()      {        Console.WriteLine("ConcreteClassB.PrimitiveOperation1()");      }      public override void PrimitiveOperation2()      {        Console.WriteLine("ConcreteClassB.PrimitiveOperation2()");      }    }  } |
| Output  ConcreteClassA.PrimitiveOperation1() ConcreteClassA.PrimitiveOperation2() |

**The Template Method pattern makes the method call in a special way. What is that?**

The method itself is general. It makes the method call via a reference pointing to one of the derived classes to handle the special details.

**According to the Gang of Four, the intent of the Template Method pattern is to "Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Redefine the steps in an algorithm without changing the algorithm's structure" What does this mean?**

It helps us to generalize a common process - at an abstract level - from a set of different procedures. It helps to identify the common ground between the set of different procedures while encapsulating the differences in derived classes

**The Gang of Four calls this a "Template Method". Why do they do this?**

Because it provides a boilerplate (or a "template") that specifies the generic actions and derived class implements the specific steps required for the actions to take

**What is the difference between the Strategy pattern and the Template Method pattern?**

The Template Method pattern is applicable when there are different, but conceptually similar processes. The Strategy pattern controls a family of algorithms. They do not have to be conceptually similar. You choose the algorithm to employ just in time.

| **Strategy pattern** | **Template Method pattern** |
| --- | --- |
| Encapsulates an algorithm inside a class. | Defer the exact steps of an algorithm to a subclass. It is better to use template method in cases when you have a "template" algorithm having defined steps where subclasses override these steps to change some details. |
| It allows algorithm vary independently (algorithm structure can be changed). | It does not allow algorithm vary independently (algorithm structure cannot be changed), only certain steps of an algorithm can be redefined in subclasses. |
| It uses object composition (has-a). | It forces use of class inheritance instead of object composition (is-a). |
| In case of strategy, you need to create an interface, and instead of inheritance you are using delegation. It is a bit more powerful pattern and may be better in accordance to DIP - dependency inversion principles. It is more powerful because you clearly define a new abstraction of strategy - a way of doing something, which does not apply to template method. So, if this abstraction makes sense - use it. | It does not take advantage of DI principle. |

**Facade pattern**

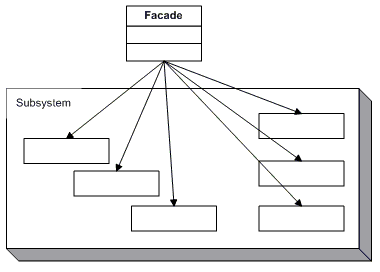
**Define Façade.**

A Façade is "The face of a building, especially the principal face" - dictionary.com. It is the front that separates the street from the inside. It is a structural pattern.

**What is the intent of the Façade pattern?**

Provides a unified interface to a set of interfaces in a sub-system. Façade defines a higher-level interface that makes the subsystem easier to use.

**Explain UML class diagram of façade pattern.**

****

### Participants

    The classes and/or objects participating in this pattern are:

* **Facade**
  + knows which subsystem classes are responsible for a request.
  + delegates client requests to appropriate subsystem objects.
* **Subsystem classes**
  + implement subsystem functionality.
  + handle work assigned by the Facade object.
  + have no knowledge of the facade and keep no reference to it.

This structural code demonstrates the Facade pattern which provides a simplified and uniform interface to a large subsystem of classes.

| // Facade pattern -- Structural example |
| --- |
| using System;    namespace DoFactory.GangOfFour.Facade.Structural  {    /// <summary>    /// MainApp startup class for Structural    /// Facade Design Pattern.    /// </summary>    class MainApp    {      /// <summary>      /// Entry point into console application.      /// </summary>      public static void Main()      {        Facade facade = new Facade();          facade.MethodA();        facade.MethodB();          // Wait for user        Console.ReadKey();      }    }      /// <summary>    /// The 'Subsystem ClassA' class    /// </summary>    class SubSystemOne    {      public void MethodOne()      {        Console.WriteLine(" SubSystemOne Method");      }    }      /// <summary>    /// The 'Subsystem ClassB' class    /// </summary>    class SubSystemTwo    {      public void MethodTwo()      {        Console.WriteLine(" SubSystemTwo Method");      }    }      /// <summary>    /// The 'Subsystem ClassC' class    /// </summary>    class SubSystemThree    {      public void MethodThree()      {        Console.WriteLine(" SubSystemThree Method");      }    }      /// <summary>    /// The 'Subsystem ClassD' class    /// </summary>    class SubSystemFour    {      public void MethodFour()      {        Console.WriteLine(" SubSystemFour Method");      }    }      /// <summary>    /// The 'Facade' class    /// </summary>    class Facade    {      private SubSystemOne \_one;      private SubSystemTwo \_two;      private SubSystemThree \_three;      private SubSystemFour \_four;        public Facade()      {        \_one = new SubSystemOne();        \_two = new SubSystemTwo();        \_three = new SubSystemThree();        \_four = new SubSystemFour();      }        public void MethodA()      {        Console.WriteLine("\nMethodA() ---- ");        \_one.MethodOne();        \_two.MethodTwo();        \_four.MethodFour();      }        public void MethodB()      {        Console.WriteLine("\nMethodB() ---- ");        \_two.MethodTwo();        \_three.MethodThree();      }    }  } |
| Output  MethodA() ---- SubSystemOne Method SubSystemTwo Method SubSystemFour Method  MethodB() ---- SubSystemTwo Method SubSystemThree Method |

**What are the consequences of the Façade pattern? Give an example.**

The Façade simplifies the use of the required subsystem. However, since the Façade is not complete, certain functionality may be unavailable to the client.

**Give an example where façade can be used.**

Example is a reporting application that needs a routine way to access on certain portions of a database system: The Façade would provide an interface to those portions and not the entire API of the database.

**In the Façade pattern, how do clients work with subsystems?**

Clients work with sub-systems through the Façade's interfaces. They do not interact with the underlying methods directly (p. 91)

**Does the Façade pattern usually give you access to the entire system?**

Not usually. In general, Façade give access to a portion of the system, one that is customized to our needs. (p. 89)

**The Gang of Four says that the intent of the Façade pattern is to "provide a unified interface to a set of interfaces in a sub-system. Façade defines a higher-level interface that makes the subsystem easier to use." What does this mean? Give an example.**

The Façade gives a simpler way to access an existing system by giving an interface that is customized to the needs you have. (p. 90)

Example is a class that insulates a client program from a database system (p. 92)

**Here is an example of a Facade that comes from outside of software. Pumps at gasoline stations in the US can be very complex. There are many options on them: how to pay, the type of gas to use, watch an advertisement. One way to get a unified interface to the gas pump is to use a human gas attendant. Some states even require this.**

* What is another example from real life that illustrates a Facade?
* Another example could be a stockbroker who serves as the interface to a complex system of stock trades.

**Observer**

**How can an object notify other objects of state changes without being dependent on their classes?**

The easiest way to inform dependent objects of a state change is to call them directly. However, direct collaboration between objects creates dependency between their classes.

Use the *Observer* pattern to maintain a list of interested dependents (observers) in a separate object (the subject). Have all individual observers implement a common *Observer* interface to eliminate direct dependencies between the subject and the dependent objects (see Figure 1).

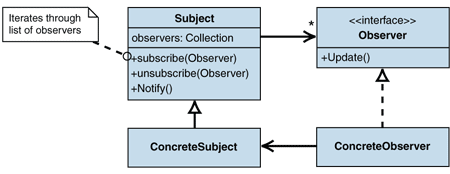


Figure 1: Basic Observer structure

When a state change occurs in the client that is relevant to the dependent objects, **ConcreteSubject** invokes the **Notify()** method. The **Subject** superclass maintains a list of all interested observers so that the **Notify()** method can loop through the list of all observers and invoke the **Update()** method on each registered observer. The observers register and unregister for updates by calling the **subscribe()** and **unsubscribe()** methods on **Subject** (see Figure 2). One or more instances of **ConcreteObserver** may also access **ConcreteSubject** for more information and therefore usually depend on the **ConcreteSubject** class. However, as Figure 1 illustrates, there is no direct or indirect dependency from the **ConcreteSubject** class on the **ConcreteObserver** class.

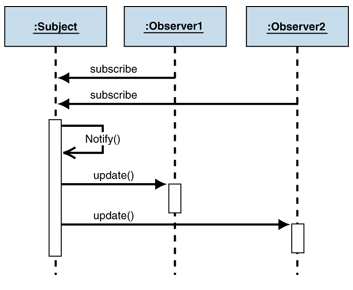


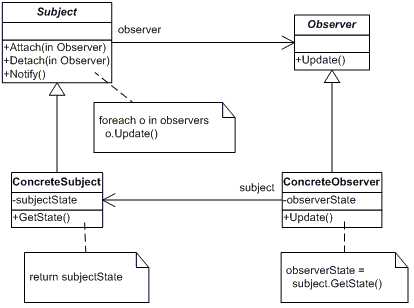
Figure 2: Basic Observer interaction

With this generic way of communicating between the subject and observers, collaborations can be built dynamically instead of statically. Due to the separation of notification logic and synchronization logic, new observers can be added without modifying the notification logic, and notification logic can also be changed without affecting the synchronization logic in observers. The code is now much more separate, and thus easier to maintain and reuse.

Notifying objects of changes without incurring a dependency on their classes is such a common requirement that some platforms provide language features to perform this function. For example, the Microsoft .NET Framework defines the notion of delegates and events to accomplish the *Observer* role. Therefore, you would rarely implement the *Observer* pattern explicitly in .NET, but should use delegates and events instead. Most .NET developers will think of the *Observer* pattern as a complicated way to implement events.

**Explain how observer pattern can be implemented in .NET.**

*Observer Pattern in C# = Events & delegates*

One of the most interesting patterns in Design Patterns is the Observer pattern which is listed under Behavioral Patterns, it is really important how to make other classes which are interested in the state of another object get notified when the state changed.  
  
  
This is the classic Observer pattern which used to be good with C++ and Java, however in C# you can implement the same idea using Delegates and Events which is really more concise and elegant way of writing this pattern

using System;  
using System.Collections.Generic;  
using System.Text;  
  
namespace Patterns  
{  
 delegate void StateChangeHandler(State newState);  
 private event StateChangeHandler \_onChange;  
  
 enum State  
 {  
 State1, State2, State3  
 }  
  
 class Product  
 {  
 private State \_state;  
  
 public State MyState  
 {  
 get { return \_state; }  
 set  
 {  
 if (\_state != value)  
 {  
 \_state = value;  
 Notify();  
 }  
 }  
 }  
  
 private void Notify()  
 {  
 if (\_onChange != null)  
 \_onChange(\_state);  
 }  
  
 public event StateChangeHandler OnStateChange  
 {  
 add  
 {  
 \_onChange += value;  
 }  
 remove  
 {  
 \_onChange -= value;  
 }  
 }  
 }  
}

Take a look on the previous code, the Product class has an important piece of info called \_state, and is encapsulated in the property MyState, this class expects that other classes may be interested in observing the changes in the MyState, so the class adds another member which is an Event (\_onChange) of type StateChangeHandler delegate, and encapsulated in the Event Property called OnStateChange, and in the setter of the property MyState a small check is made to see whether the new value is different than the older value then the event gets fired.  
A typical class which makes use of the Product class will look similar to this.

using System;  
using System.Collections.Generic;  
using System.Text;  
  
namespace Patterns  
{  
 class Program  
 {  
 static void Main(string[] args)  
 {  
 Product myProduct = new Product();  
 myProduct.OnStateChange += new StateChangeHandler(myProduct\_OnStateChange);  
 myProduct.MyState = State.State3;  
 }  
  
 static void myProduct\_OnStateChange(State newState)  
 {  
 Console.WriteLine("State changed to {0}", newState);  
 }  
 }  
}

So now the Program class instantiates an object of type Product and registers itself in the OnStateChange event so whoever changes the MyState property of the object myProduct, the Program class gets notified.

**What are the benefits and tradeoffs when employing the *Observer* pattern?**

*Benefits:*

**Loose coupling and reduced dependencies**. The client is no longer dependent on the observers because it is isolated through the use of a subject and the **Observer** interface. This advantage is used in many frameworks where application components can register to be notified when (lower-level) framework events occur. As a result, the framework calls the application component, but is not dependent on it.

*Tradeoffs:*

 **Memory leaks**. The callback mechanism (when an object registers to be called later) used in *Observer* can lead to a common mistake that results in memory leaks, even in managed C# code. Assuming that an observer goes out of scope but forgets to unsubscribe from the subject, the subject still maintains a reference to the observer. This reference prevents garbage collection from reallocating the memory associated with the observer until the subject object is destroyed as well. This can lead to serious memory leaks if the lifetime of the observers is much shorter than the lifetime of the subject (which is often the case).

 **Hidden dependencies**. The use of observers turns explicit dependencies (through method invocations) into implicit dependencies (via observers). If observers are used extensively throughout an application, it becomes nearly impossible for a developer to understand what is happening by looking at the source code. This makes it very difficult to understand the implications of code changes. The problem grows exponentially with the levels of propagation (for example, an observer acting as **Subject**). Therefore, you should limit the use of observers to few well-defined interactions, such as the interaction between model and view in the *Model-View-Controller* pattern. The use of observers between domain objects should generally be cause for suspicion.

 **Testing/Debugging difficulties**. As much as loose coupling is a great architectural feature, it can make development more difficult. The more you decouple two objects, the more difficult it becomes to understand the dependencies between them when looking at the source code or a class diagram. Therefore, you should only loosely couple objects if you can safely ignore the association between them (for example, if the observer is free of side effects).

**What is the one true thing about requirements?**

Requirements always change! Plan for it.

**What is the intent of the Observer pattern?**

The Gang of Four says that the intent of the Observer pattern is to "define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."

Command Pattern

The Principles and Strategies of Design Patterns

**When it comes to choosing how to implement a design, what question do I suggest asking?**

Rather than ask, "Which implementation is better?" ask, for each alternative, "Under what circumstances would this alternative be better than the other alternative" and then "Which of these circumstances to I have in my problem domain?"

**What are the five errors of using design patterns?**

Superficiality, Bias, Selection, Misdiagnosis, Fit

**The "open-closed" principle says, "modules, methods, and classes should be open for extension while closed for modification." What does this mean?**

Bertrand Meyer puts this forward as a way to minimize risk when changing systems. Basically, it means that we want to be able to extend the capabilities of our systems without substantially changing it. Design in such a way that the software can absorb new variations without having to introduce new fundamental structure.

**In what way does the Bridge pattern illustrate the open-closed principle?**

Bridge allows us to add new implementations without changing any existing classes.

How Do Experts Design?

**1. Alexander uses the term, "alive" to characterize good designs. What terms do I suggest using when it comes to software?**

"When you read ‘alive', think ‘robust' and ‘flexible' systems. (p. 189)

**2. Good design requires keeping what in mind?**

Keeping the big picture in mind. Being able to consider the forest first and then the trees. (p. 189)

**3. Alexander suggests that the best approach to design involves"complexification." What does this mean?**

Complexification is the approach to design to starts by looking at the problem in its simplest terms and then adds additional features (distinctions), making the deisgn more complex as we go because we are adding information (p. 190)

**4. To Alexander, what relationships does a pattern define?**

A pattern defines relationships between the entities in his problem domain (p. 191, 192) This is why define a pattern as a solution to a problem in a context. The entities describe the context in which the pattern exists.

**5. What are Alexander's five steps to design?**

Identify patterns that are present in your problem.

Start with context patterns (those that create context for other patterns)

Work inward from the context

Refine the design

Implement (p. 193)

**Interpretations**

**1. I quote Alexander, "But it is impossible to form anything which has the character of nature by adding preformed parts." What does Alexander mean by this?**

Alexander believes that designs that have the "character of nature" are those that humans would judge has being superior in design. They are "alive" and feel right. He believes that buildings (or in our case, software) that is built simply by assembling stock parts will not feel "alive". They will have all the charm of 60s style block houses: functional but dead. In software terms, it works the same way: cobbling together objects does not create solutions that are easily maintained: robust and flexible.

Prototype

**What is the Prototype Pattern?**

*The prototype design pattern is a design pattern that is used to instantiate a class by*

*copying, or cloning, the properties of an existing object. The new object is an exact copy*

*of the prototype but permits modification without altering the original.*

The prototype pattern is a Gang of Four design pattern. This is a *creational* pattern as it

is used to control class instantiation and object generation. The pattern is used to

instantiate a new object by copying all of the properties of an existing object, creating an

independent *clone*. This practise is particularly useful when the construction of a brand

new object, using the *new* operator, is inefficient.

In general, when an object is cloned, the new object is either a *shallow* or *deep copy*. A

shallow copy duplicates all of the object's properties. If any property contains a

reference type, the reference is copied. This means that changes to the referenced

object are visible in both the clone and the original object. A deep copy clones the main

object and all child objects. Any properties of reference types are also cloned, giving a

truly independent copy. The prototype pattern usually generates deep copies, though

this is dependant upon the situation.

**How can one Implement the Prototype Pattern?**

The UML class diagram above describes an implementation of the prototype pattern.

The classes in the diagram are as follows:

\_ **Prototype**. This abstract class is the base class for the types of object that can be

generated and cloned. The class contains a single virtual method named "Clone"

that returns a Prototype object. This class can be replaced using an interface if no

inheritable functionality is required. Indeed, the .NET framework includes a

standard interface, named "ICloneable", for this purpose. However, as the Clone

method of the ICloneable interface returns an object of the type System.Object, you

may prefer to implement your own version.

\_ **ConcretePrototype (A/B)**. These classes inherit from the Prototype class and

include any additional required functionality. They also override the Clone method

to implement the duplication of their properties. There may be any number of

concrete prototypes.

The following code shows the basic code of the prototype design pattern implemented

using C#:

\_\_

\_

public abstract class Prototype

{

public abstract Prototype Clone();

}

public class ConcretePrototypeA : Prototype

{

public override Prototype Clone()

{

return (Prototype)MemberwiseClone();

}

}

public class ConcretePrototypeB : Prototype

{

public override Prototype Clone()

{

return (Prototype)MemberwiseClone();

}

}

In the example code, the base class and two subclasses are shown. As this code shows

only the basic structure of the pattern, no properties or fields have been defined so

nothing will be copied during the cloning operation.

*NB: The* MemberwiseClone *method creates shallow copies. For more complex objects*

*that contain reference types, you will probably want to alter this to perform a deep copy*

**Give an example of prototype pattern.**

The prototype design pattern is generally used for complex classes or for classes that

are costly to instantiate. In this example, we will use a simple class to keep the code to

a manageable size. The base prototype class represents an employee of a business.

This class incorporates two properties to hold the employee's name and role. It also, of

course, includes the Clone method.

Two concrete classes are created to represent typists and software developers. Each

inherits from the employee prototype base class and adds an extra property for the

specific subclass. As both classes contain only value types and strings, it is acceptable

to use MemberwiseClone for copying. If reference types were included and needed to

be duplicated, rather than just copying the reference, manual cloning of each property

would be required. Finally, the ToString method for each class is overridden to make it

easier to output details of the objects.

*NB: For brevity the properties are defined using the .NET 3.0 automatically*

*implemented property syntax. For earlier versions of the .NET framework, these must*

*be expanded.*

\_\_

\_

public abstract class Employee

{

public abstract Employee Clone();

public string Name { get; set; }

public string Role { get; set; }

}

public class Typist : Employee

{

public int WordsPerMinute { get; set; }

public override Employee Clone()

{

return (Employee)MemberwiseClone();

}

public override string ToString()

{

return string.Format("{0} - {1} - {2}wpm", Name, Role,

WordsPerMinute);

}

}

public class Developer : Employee

{

public string PreferredLanguage { get; set; }

public override Employee Clone()

{

return (Employee)MemberwiseClone();

}

public override string ToString()

{

return string.Format("{0} - {1} - {2}", Name, Role,

PreferredLanguage);

}

}

***Testing the Prototype***

The above implementation of the prototype pattern can be tested by creating and

cloning an object of each concrete class. In the sample code below, a Developer object

is instantiated and cloned. The Name property of the new Developer object is changed

before string representations of both objects are outputted to the console. The outputted

\_\_

\_

text shows that the properties have been cloned and are now independent. This

process is then repeated with a Typist object, which is cloned before the copy has its

name and words per minute properties modified.

Developer dev = new Developer();

dev.Name = "Bob";

dev.Role = "Team Leader";

dev.PreferredLanguage = "C#";

Developer devCopy = (Developer)dev.Clone();

devCopy.Name = "Sue";

Console.WriteLine(dev);

Console.WriteLine(devCopy);

/\* OUTPUT

Bob - Team Leader - C#

Sue - Team Leader - C#

\*/

Typist typist = new Typist();

typist.Name = "Kay";

typist.Role = "Typist";

typist.WordsPerMinute = 120;

Typist typistCopy = (Typist)typist.Clone();

typistCopy.Name = "Tim";

typistCopy.WordsPerMinute = 115;

Console.WriteLine(typist);

Console.WriteLine(typistCopy);

/\* OUTPUT

Kay - Typist - 120wpm

Tim - Typist - 115wpm

\*/*.*

**Builder Pattern**

**Fluent Builder Pattern for classes with long-ish constructors**

Last week I discovered a rather wonderful construct for objects with long constructors, e.g. immutable value types:

| public class UserProfile |  |
| --- | --- |
| { |

| public string City { get; protected set; } |
| --- |
| public string Country { get; protected set; } |  |

| public Uri Url { get; protected set; } |
| --- |
| public string Email { get; protected set; } |  |

| public string Tagline { get; protected set; } |  |
| --- | --- |
|  |

| public UserProfile(string city, string country, Uri url, string email, |  |
| --- | --- |
| string tagline) |

| { |
| --- |
| ... |  |

| } |  |
| --- | --- |
| } |

This constructor has bad [Connascence of Position (CoP)](http://urgetopunt.com/2009/03/27/sor-connascence.html); to construct a UserProfile instance, users have to know the position of each parameter. Otherwise they might mix up the city with the country for example:

| // Spot the bug! |
| --- |
| var profile = new UserProfile("NZ", "Wellington", |  |

| new Uri("<http://richarddingwall.name>"), "rdingwall@gmail.com", ".NET guy"); |
| --- |
|  |

This won’t be a problem with [named parameters in C# 4.0](http://bartdesmet.net/blogs/bart/archive/2008/11/01/c-4-0-feature-focus-part-2-named-parameters.aspx), but until then, a nice alternative is a fluent builder class as described [here](http://nhforge.org/blogs/nhibernate/archive/2008/09/17/value-objects.aspx):

| UserProfile profile = new UserProfileBuilder() |  |
| --- | --- |
| .WithCity("Wellington") |

| .WithCountry("NZ") |
| --- |
| .WithUrl(new Uri("<http://richarddingwall.name>")) |  |

| .WithEmail("rdingwall@gmail.com") |  |
| --- | --- |
| .WithTagline(".NET guy"); |
|  |

Builders are very easy to implement. Each *With* method records its value and returns the current builder instance. Then we provide an implicit cast operator that finally constructs a UserProfile with all the parameters in the right places.

| Public class UserProfileBuilder |  |
| --- | --- |
| { |

| internal string City { get; set; } |
| --- |
| internal string Country { get; set; } |  |

| // ... etc |  |
| --- | --- |
|  |

| public UserProfileBuilder WithCity(string city) |  |
| --- | --- |
| { |

| this.City = city; |  |
| --- | --- |
| return this; |

| } |  |
| --- | --- |
|  |

| // ... etc |  |
| --- | --- |
|  |

| public static implicit operator UserProfile(UserProfileBuilder builder) |  |
| --- | --- |
| { |

| return new UserProfile(builder.City, builder.Country, builder.Url, |  |
| --- | --- |
| builder.Email, builder.Tagline); |

| } |  |
| --- | --- |
| } |

**Software Architecture**

**What is Software Architecture?**

It is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships between them.

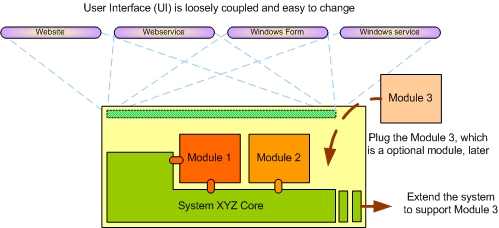
Software Architecture is defined to be the rules, heuristics and patterns governing:

* Partitioning the problem and the system to be built into discrete pieces
* Techniques used to create interfaces between these pieces
* Techniques used to manage overall structure and flow
* Techniques used to interface the system to its environment
* Appropriate use of development and delivery approaches, techniques and tools.

**What should be the objective of a good architecture?**

A good architecture is often harder to implement, but is easier to maintain because it often reduces the volume of code. This means that hours spent supporting an application are reduced.

**Why Architecture is important? Or What are the benefits of architecture?**

****

The primary goal of software architecture is to define the non-functional requirements of a system and define the environment. The detailed design is followed by a definition of how to deliver the functional behavior within the architectural rules. Architecture is important because it:

* Controls complexity
* Enforces best practices
* Gives consistency and uniformity
* Increases predictability
* Enables re-use
* Makes easy to unit test and maintain

**How to identify and design a Class?**

This is an art; each designer uses different techniques to identify classes. However according to Object Oriented Design Principles, there are five principles (SOLID) that you must follow when design a class,

* SRP - The Single Responsibility Principle -   
  A class should have one, and only one, reason to change.
* OCP - The Open Closed Principle -   
  You should be able to extend a classes behavior, without modifying it.
* LSP - The Liskov Substitution Principle-   
  Derived classes must be substitutable for their base classes.
* ISP - The Interface Segregation Principle-   
  Make fine grained interfaces that are client specific.
* DIP - The Dependency Inversion Principle-   
  Depend on abstractions, not on concretions.

For more information on design principles, please refer to [Object Mentor](http://design?).

Additionally to identify a class correctly, you need to identify the full list of leaf level functions/ operations of the system (granular level use cases of the system). Then you can proceed to group each function to form classes (classes will group same types of functions/ operations). However, *a well defined class must be a meaningful grouping of a set of functions and should support the re-usability while increasing expandability/ maintainability of the overall system.*

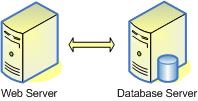
In software world the concept of dividing and conquering is always recommended, if you start analyzing a full system at the start, you will find it harder to manage. So the better approach is to identify the module of the system first and then dig deep in to each module separately to seek out classes.

A software system may consist of many classes. But in any case, when you have many, it needs to be managed. Think of a big organization, with its work force exceeding several thousand employees (let’s take one employee as a one class). In order to manage such a work force, you need to have proper management policies in place. Same technique can be applies to manage classes of your software system as well. In order to manage the classes of a software system, and to reduce the complexity, the system designers use several techniques, which can be grouped under four main concepts named *Encapsulation, Abstraction, Inheritance, and Polymorphism.* These concepts are the four main gods of *OOP* world and in software term, they are called four main Object Oriented Programming (*OOP*) Concepts.

**What is two-tier architecture?**

The two-tier architecture is refers to client/ server architectures as well, the term client/ server was first used in the 1980s in reference to personal computers (PCs) on a network. The actual client/ server model started gaining acceptance in the late 1980s, and later it was adapted to World Wide Web programming.

According to the modern days use of two-tier architecture the user interfaces (or with ASP.NET, all web pages) runs on the client and the database is stored on the server. The actual application logic can run on either the client or the server. So in this case the user interfaces are directly access the database. Those can also be non-interface processing engines, which provide solutions to other remote/ local systems. In either case, today the two-tier model is not as reputed as the three-tier model. The advantage of the two-tier design is its simplicity, but the simplicity comes with the cost of scalability. The newer three-tier architecture, which is more famous, introduces a middle tier for the application logic.



When the .NET 2.0 framework became available to the world, there were some neat features that allowed the developer to connect the framework’s GUI controls directly to the database. This approach is very handy when rapidly developing applications. However, it’s not always favorable to embed all of the business logic and data access code directly in the web site, for several reasons:

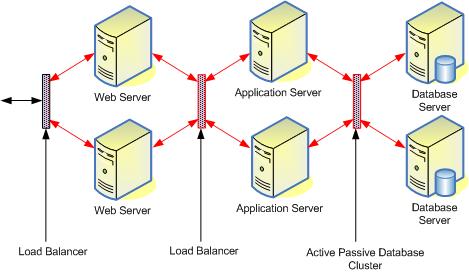
* Putting all of the code in the web site (business logic and data access) can make the application harder to maintain and understand.
* Reusing database queries in the presentation layer often isn’t done, because of the typical data source control setup in the ASP.NET framework.
* Relying on the data source controls can make debugging more difficult, often due to vague error messages.

So in looking for an alternative, we can separate the data access code and business logic into separate “layers”, which we’ll discuss next.

**What is three-tier architecture?**

The three tier software architecture (also known as three layer architectures) emerged in the 1990s to overcome the limitations of the two tier architecture. This architecture has aggressively customized and adopted by modern day system designer to web systems.

Three-tier is a client-server architecture in which the user interface, functional process logic, data storage and data access are developed and maintained as independent modules, some time on separate platforms. The term "*three-tier*" or "*three-layer*", as well as the concept of multi-tier architectures (often refers to as three-tier architecture), seems to have originated within Rational Software.



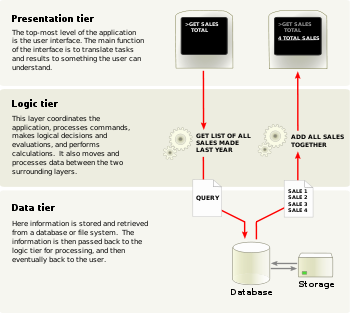
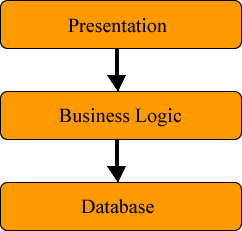
The 3-Tier architecture has the following three tiers.

1. **Presentation Tier or Web Server**: User Interface, displaying/ accepting data/ input to/ from the user
2. **Application Logic/ Business Logic/ Transaction Tier or Application Server**: Data validation, acceptability check before being added to the database and all other business/ application specific operations
3. **Data Tier or Database server**: Simple reading and writing method to database or any other storage, connection, command, stored procedures etc

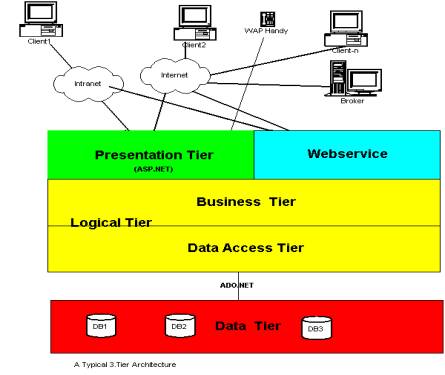
**What is three-tier architecture?**

*Three-tier*  is a client-server architecture in which the user interface, functional process logic ("business rules"), computer data storage and data access are developed and maintained as independent modules, most often on separate platforms.

The three-tier model is software architecture and a software design pattern.

Below is another view of three-tier architecture:



Typically, the user interface runs on a desktop PC or workstation and uses a standard graphical user interface, functional process logic may consist of one or more separate modules running on a workstation or application server, and an RDBMS on a database server or mainframe contains the computer data storage logic. The middle tier may be multi-tiered itself (in which case the overall architecture is called an "n-tier architecture").

Three-tier architecture has the following three tiers:

* *Presentation tier:* This is the topmost level of the application. The presentation tier displays information related to such services as browsing merchandise, purchasing, and shopping cart contents. It communicates with other tiers by outputting results to the browser/client tier and all other tiers in the network. It contains the interface code, that is going to be displayed to the user. This code could contain any technology that can be used on the client side like HTML, JavaScript or VBScript etc.
* *Application tier (Business Logic/Logic Tier/Data Access Tier/Middle Tier):* The logic tier is pulled out from the presentation tier and, as its own layer, it controls an application’s functionality by performing detailed processing. It contains all the code of the server-side technology. This layer mainly contains the code that is used for accessing the database and to query, manipulate, pass data to user interface and handle any input from the UI as well.
* *Data tier:* This tier consists of Database Servers. Here information is stored and retrieved. This tier keeps data neutral and independent from application servers or business logic. Giving data its own tier also improves scalability and performance. It represents the data store like MS Access, SQL Server, an XML file, an Excel file or even a text file containing data.

**What are the advantages of the three-tier architecture?**

* Helps in modular [software](http://software) design with well defined interfaces,
* Allow any of the three tiers to be upgraded or replaced independently as requirements or [technology](http://technology) change. For example, a change of [operating system](http://system) in the presentation tier would only affect the user interface code.
* Easier to maintain because it often reduces the volume of code. This means that hours spent supporting an application are reduced.
* Testing becomes easier. Each module can be tested independently.

**What is the Data Access Layer?**

The data access layer (DAL), which is a key part of every n-tier system, is mainly consist of a simple set of code that does basic interactions with the database or any other storage device. These functionalities are often referred to as CRUD (Create, Retrieve, Update, and Delete).

The data access layer need to be generic, simple, quick and efficient as much as possible. It should not include complex application/ business logics.

I have seen systems with lengthy, complex store procedures (SP), which run through several cases before doing a simple retrieval. They contain not only most part of the business logic, but application logic and user interface logic as well. If SP is getting longer and complicated, then it is a good indication that you are burring your business logic inside the data access layer.

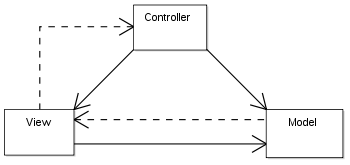
**What is the Business Logic Layer?**

I know for a fact that this is a question for most, but from the other hand by reading many articles I have become aware that not everyone agrees to what business logic actually is, and in many cases it's just the bridge in between the presentation layer and the data access layer with having nothing much, except taking from one and passing to the other. In some other cases, it is not even been well thought out, they just take the leftovers from the presentation layer and the data access layer then put them in another layer which automatically is called the business logic layer. However there are no god said things that cannot be changed in software world. You can change as and when you feel comfortable that the method you apply is flexible enough to support the growth of your system. There are many great ways, but be careful when selecting them, they can over complicating the simple system. It is a balance one needs to find with their experience.

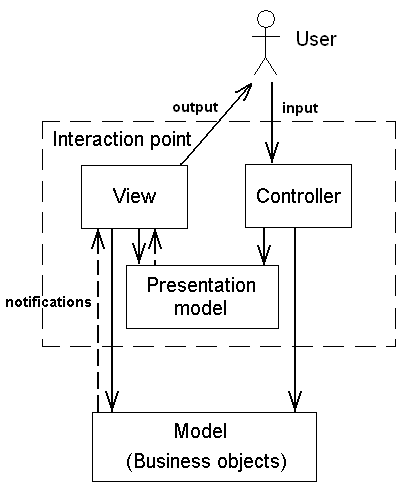
As a general advice when you define business entities, you must decide how to map the data in your tables to correctly defined business entities. The business entities should meaningfully define considering various types of requirements and functioning of your system. It is recommended to identify the business entities to encapsulate the functional/ UI (User Interface) requirements of your application, rather than define a separate business entity for each table of your database. For example, if you want to combine data from couple of table to build a UI (User Interface) control (Web Control), implement that function in the Business Logic Layer with a business object that uses couple of data object to support with your complex business requirement.

**What is MVC architecture?**

Model–View–Controller (MVC) is a software architecture, currently considered an architectural pattern used in [software engineering](http://engineering). The pattern isolates "[domain logic](http://logic)" (the application logic for the user) from [input and presentation](http://interface) ([GUI](http://interface)), permitting [independent development, testing and maintenance of each](http://concerns).



*Model-View-Controller concept. Note: The solid line represents a direct association, the dashed an indirect association via an observer (for example).*

**

The *model* is the domain-specific representation of the data upon which the application operates. Domain logic adds meaning to raw data (for example, calculating whether today is the user's birthday, or the totals, taxes, and shipping charges for shopping cart items). When a model changes its state, it notifies its associated views so they can refresh.

Many applications use a persistent storage mechanism such as a database to store data. MVC does not specifically mention the data access layer because it is understood to be underneath or encapsulated by the model. Models are not data access objects; however, in very simple apps that have little domain logic there is no real distinction to be made. Also, the [ActiveRecord](http://pattern) is an accepted design pattern which merges domain logic and data access code - a model which knows how to persist itself.

The *view* renders the model into a form suitable for interaction, typically a user interface element. Multiple views can exist for a single model for different purposes.

The c*ontrolle*r receives input and initiates a response by making calls on model objects. Controller handles user input, makes subsequent calls to the business objects and manages the application flow. In particular the controller decides what to display to the user. However it is not allowed in MVC to access the view directly, instead the underlying model should be altered and the changes will be propagated to the view through the observer mechanism. Thus in order to make the view update itself the controller should change the presentation model object.

An MVC application may be a collection of model/view/controller triplets, each responsible for a different UI element.

**What is the goal of MVC?**

The goal of MVC is, by decoupling models and views, to reduce the complexity in architectural design and to increase flexibility and maintainability of code.

**Compare three-tier with MVC architecture.**

At first glance, the three tiers may seem similar to the MVC (Model View Controller) concept; however, topologically they are different.

| **Three-tier** | **MVC** |
| --- | --- |
| A fundamental rule in a three-tier architecture is the client tier never communicates directly with the data tier; in a three-tier model all communication must pass through the middleware tier. | It allows the client tier to communicate directly with data tier. |
| Conceptually the three-tier architecture is linear. | The MVC architecture is triangular: the View sends updates to the Controller, the Controller updates the Model, and the View gets updated directly from the Model. |
| The three-tier architecture concept emerged in the 1990s from observations of distributed systems (e.g., web applications) where the client, middleware and data tiers ran on physically separate platforms. | MVC comes from the previous decade (by work at Xerox PARC in the late 1970s and early 1980s) and is based on observations of applications that ran on a single graphical workstation; MVC was applied to distributed applications much later in its history. |

**Name some MVC frameworks available in .NET.**

* ASP.NET MVC Framework
* Maverick.NET
* MonoRail An ActionPack inspired MVC framework from the Castle Project
* [PureMVC](http://puremvc) Framework for C#
* Spring Framework.NET
* NFX Framework (.Net Framework Extension)
* Bistro Framework

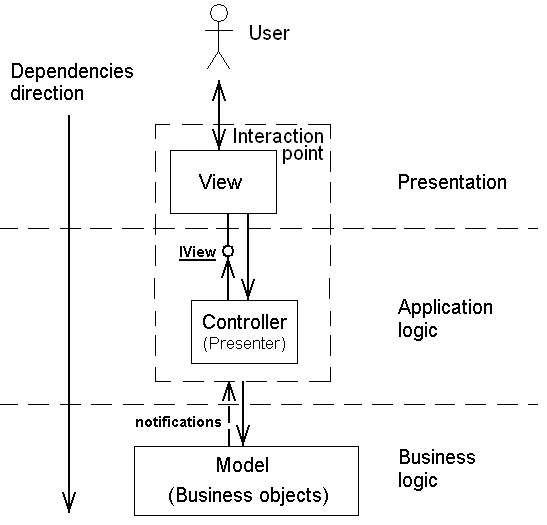
**What are the drawbacks of MVC pattern?**

* MVC has higher complexity because of the observer mechanism: in order to update the view the controller must make changes to the presentation model, which will trigger the view update. Such indirect relationship may be difficult to understand. Instead the controller could simply send a message to the view, however such direct linking is not allowed in MVC.
* MVC does not conform to the modern UI programming environments where widgets themselves handle user gestures. For example form classes (either web or Windows) in .NET applications by default contain handlers for user input. Thus it would be difficult to break the common paradigm and make controllers receive the user input instead of views.

MVP pattern is designed to eliminate the drawbacks of MVC, while preserving its separation of application logic and presentation mechanism.

**What is MVP architecture?**

The MVP pattern is a UI presentation pattern based on the concepts of the MVC pattern. Model-view-presenter (MVP) is a user interface design pattern engineered to facilitate automated unit testing and improve the separation of concerns in presentation logic.



* The *model* is an interface defining the data to be displayed or otherwise acted upon in the user interface. The model is responsible for business behaviors and state management.
* The *view* is an interface that displays data (the model) and routes user commands (events) to the presenter to act upon that data. The view is responsible for rending UI elements, the view interface is used to loosely couple the presenter from its view, the presenter is responsible for interacting between the view/model.
* The *presenter* acts upon the model and the view. It retrieves data from repositories (the model), persists it, and formats it for display in the view.

Normally, the view implementation instantiates the concrete presenter object, providing a reference to itself.

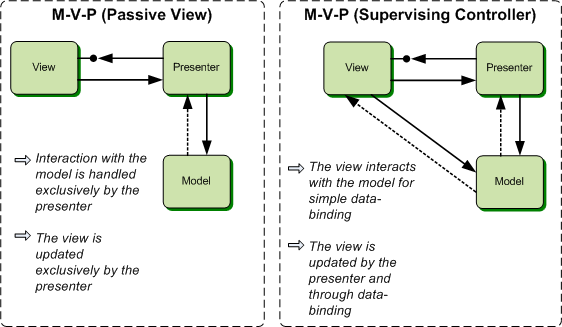
In some implementations the presenter interacts with a service (controller) layer to retrieve/persist the model. The view interface and service layer are commonly used to make writing unit tests for the presenter and the model easier.

**What are the variants of MVP pattern?**

The Model-View-Presenter variants, Passive View and Supervising Controller, specify different approaches to implementing view updates.

In *Passive View*, the presenter updates the view to reflect changes in the model. The interaction with the model is handled exclusively by the presenter; the view is not aware of changes in the model.

In *Supervising Controller*, the view interacts directly with the model to perform simple data-binding that can be defined declaratively, without presenter intervention. The presenter updates the model; it manipulates the state of the view only in cases where complex UI logic that cannot be specified declaratively is required. Examples of complex UI logic might include changing the color of a control or dynamically hiding/showing controls. Figure 1 illustrates the logical view of the Passive View and Supervising Controller variants.



**What is the difference between Supervising Controller and Passive View MVP pattern?**

The main difference between the patterns is that 'Supervising Controller' encourages coupling between the View and the Model (via [Observer-Synchronization](http://martinfowler.com/eaaDev/MediatedSynchronization.html)) while 'Passive View' forbids it.

**How do you decide which one to use –** **Supervising Controller or Passive View MVP pattern**?

The decision to use Passive View or Supervising Controller primarily depends on how testable you want your application to be. If testability is a primary concern in your application, Passive View might be more suitable because you can test all the UI logic by testing the presenter. On the other hand, if you prefer code simplicity over full testability, Supervising Controller might be a better option because, for simple UI changes, you do not have to include code in the presenter that updates the view. When choosing between Passive View and Supervising Controller, consider the following:

* Both variants allow you to increase the testability of your presentation logic.
* Passive View usually provides a larger testing surface than Supervising Controller because all the view update logic is placed in the presenter.
* Supervising Controller typically requires less code than Passive View because the presenter does not perform simple view updates.

**What are the key benefits to using either the MVC or MVP pattern?**

* *Loose coupling* – The presenter/controller are an intermediary between the UI code and the model. This allows the view and the model to evolve independently of each other.
* *Clear separation of concerns/responsibility*

UI (Form or Page) – Responsible for rending UI elements

Presenter/controller – Responsible for reacting to UI events and interacts with the model

Model – Responsible for business behaviors and state management

* *Test Driven* – By isolating each major component (UI, Presenter/controller, and model) it is easier to write unit tests. This is especially true when using the MVP pattern which only interacts with the view using an interface.
* *Code Reuse* – By using a separation of concerns/responsible design approach you will increase code reuse. This is especially true when using a full blown domain model and keeping all the business/state management logic where it belongs.
* *Hide Data Access* – Using these patterns forces you to put the data access code where it belongs in a data access layer. There a number of other patterns that typical works with the MVP/MVC pattern for data access. Two of the most common ones are repository and unit of work. (See Martin Fowler – Patterns of Enterprise Application Architecture for more details)
* *Flexibility/Adaptable* – By isolating most of your code into the presenter/controller and model components your code base is more adaptable to change. For example consider how much UI and data access technologies have changed over the years and the number of choices we have available today. A properly design solution using MVC or MVP can support multi UI and data access technologies at the same time.

**Compare MVC and MVP architectural patterns.**

Both patterns focus on separating responsibility across multi components and promote loosely coupling the UI (View) from the business layer (Model). This means that both these patterns have been use for several years and address a key OO principal namely separation of concerns between the UI and the business layers.

There are a number of frameworks is use today that based on these patterns including: JAVA Struts, ROR, Microsoft Smart Client Software Factory (CAB), Microsoft Web Client Software Factory, and the recently announced ASP.Net MVC framework.

The major differences are how the pattern is implemented and in some advanced scenarios you need both presenters and controllers. Here are the key differences between the patterns:

*MVP Pattern*

* View is more loosely coupled to the model. The presenter is responsible for binding the model to the view.
* Easier to unit test because interaction with the view is through an interface
* Usually view to presenter map one to one. Complex views may have multi presenters.

*MVC Pattern*

* Controller are based on behaviors and can be shared across views
* Can be responsible for determining which view to display (Front Controller Pattern)

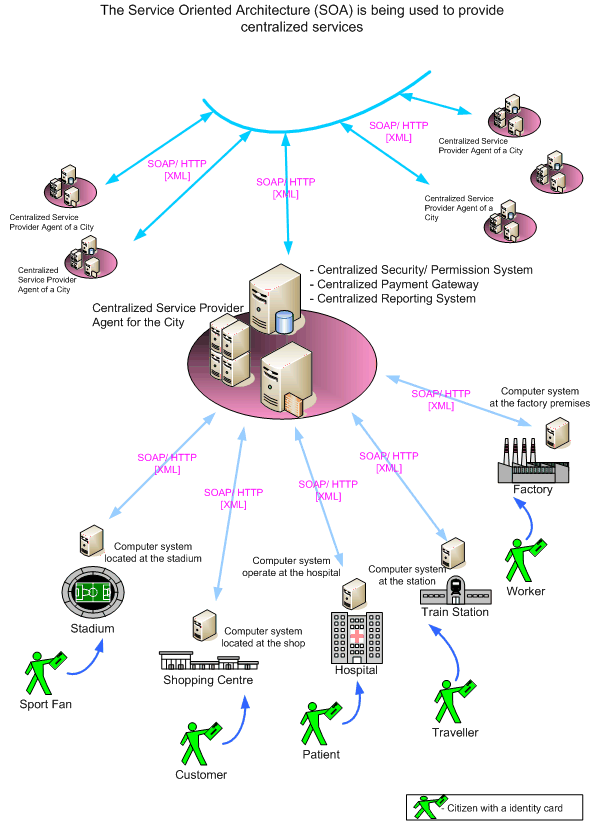
The MVP is a variation of the MVC pattern. The Presenter and Controller classes have similar roles in both patterns with one difference.

| **MVC** | **MVP** |
| --- | --- |
| In MVC, both the View and the Controller have access to the Model. | MVP essentially hides the model from the View. In MVP, the View has no direct access to the Model. It’s incorrect to say that in ‘MVP’ the view has no direct access to the model, there are two flavors of MVP, ‘Passive View’ and ‘Supervising Controller’, in ‘Passive View’ the view is passive and has no link to the model just like you said, however, in ‘Supervising Controller’ the view has link to the model. |
| It allows controllers to receive user input | It allows views to receive user input |
| A direct request from the controller to the view is not possible. | A direct request from the controller to the view is possible. Thus the controller itself may trigger the view updates instead of performing a round trip through the presentation model. In this sense the controller in MVP contains the presentation model from MVC. That is probably the reason why the controller in MVP is referred to as presenter |

**What is SOA?**

A service-oriented architecture is essentially a collection of services. These services communicate with each other. The communication can involve either simple data passing or it could involve two or more services coordinating some activity. Some means of connecting services to each other is needed.

The .Net technology introduces the SOA by mean of web services.



The SOA can be used as the concept to connect multiple systems to provide services. It has it's great share in the future of the IT world.

According to the imaginary diagram above, we can see how the Service Oriented Architecture is being used to provide a set of centralized services to the citizens of a country. The citizens are given a unique identifying card, where that card carries all personal information of each citizen. Each service centers such as shopping complex, hospital, station, and factory are equipped with a computer system where that system is connected to a central server, which is responsible of providing service to a city. As an example when a customer enter the shopping complex the regional computer system report it to the central server and obtain information about the customer before providing access to the premises. The system welcomes the customer. The customer finished the shopping and then by the time he leaves the shopping complex, he will be asked to go through a billing process, where the regional computer system will manage the process. The payment will be automatically handled with the input details obtain from the customer identifying card.

The regional system will report to the city (computer system of the city) while the city will report to the country (computer system of the country).

**What is Service-Oriented Architecture?**

SOA is an IT architecture strategy for business solution (and infrastructure solution) delivery based on the concept of service-orientation.

* It is a set of components which can be invoked, and whose interface descriptions can be published and discovered.
* It aims at building systems that are extendible, flexible and fit with legacy systems.
* It promotes the re-use of basic components called services.

**What is SOA?**

SOA stands for service oriented architecture. Before we define SOA lets first define a service. In real world service is what we pay for and we get the intended service. For instance you go to a hotel and order food. Your order first goes to the counter and then it goes to the kitchen where the food is prepared and finally the waiter serves the food.



Figure: - Hotel and services

So in order to order a item from a hotel you need the three logical departments / services to work together (counter, kitchen and waiter).   
  
In the same manner in software world these services are termed as business services. They are self contained and logical. So let's first define a business service, SOA definition will be just an extension of the same.  
  
***Definition of business service***: - It's a logical encapsulation of self contained business functionality.  
  
For instance figure 'order system' shows a simple ordering system which is achieved by different services like payment gateway, stock system and delivery system coming together. All the services are self contained and logical. They are like black boxes. In short we do not need to understand the internal details of how the business service works. For the external world it's just a black box which takes messages and serves accordingly. For instance the 'payment gateway' business service takes message 'check credit' and gives out output does the customer have credit or not. For the 'order system' business service 'payment gateway' service is a black box.

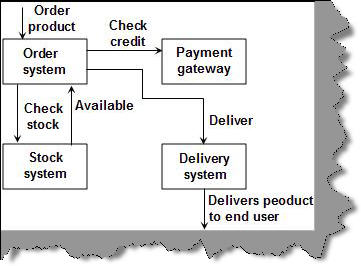


Figure: - Order system

Now let's revise some bullet points of SOA before we arrive to a definition of SOA.

* **SOA components are loosely coupled.** When we say loosely coupled means every service is self contained and exist in alone logically. For instance we take the 'payment gateway' service and attach it to a different system.
* **SOA services are black boxes.** In SOA services hide there inner complexities. They only interact using messages and send services depending on those messages. By visualizing services as black boxes services become more loosely coupled.
* **SOA service should be self defined:** - SOA services should be able to define themselves.
* **SOA Services are maintained in a listing:** - SOA services are maintained in a central repository. Applications can search the services in the central repository and use them accordingly.
* **SOA components can be orchestrated and linked to achieve a particular functionality.** SOA services can be used/orchestrated in a plug and play manner. For instance figure 'Orchestration' shows two services 'Security service' and 'Order processing service'. You can achieve two types of orchestrations from it one you can check the user first and then process order or vice-versa. Yes you guessed right using SOA we can manage work flow between services in a loosely coupled fashion.

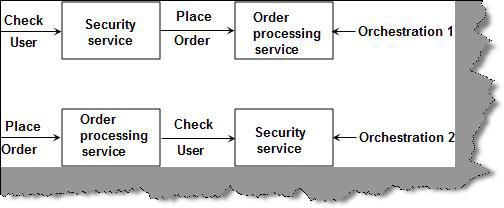


Figure: - Orchestration  
  
So let us define SOA.  
  
SOA is a architecture for building business applications using loosely coupled services which act like black boxes and can be orchestrated to achieve a specific functionality by linking together.

**(I) In SOA do we need to build systems from scratch?**

No. If you need to integrate or make an existing system as a business service, you just need to create loosely coupled wrappers which will wrap your custom systems and expose the systems functionality in generic fashion to the external world.

**(I) Can you explain business layers and plumbing layers in SOA?**  
  
In SOA we can divide any architecture in two layers. The first which has direct relevance to business as it carries out business functions. The second layer is a technical layer which talks about managing computer resources like database, web server etc. This division is needed to identify a service.

Consider the figure 'Simple order system'. It has various components which interact with each other to complete the order system functionality.

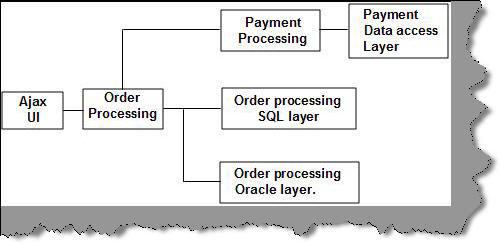


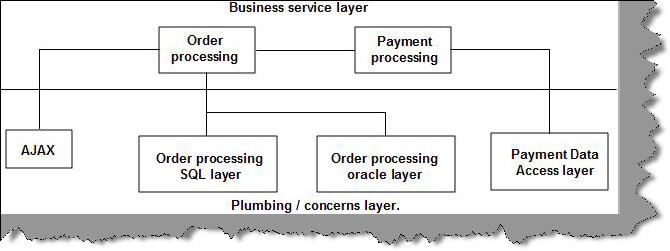
Figure: - Simple order System  
  
The simple order system can be divided in to two layers (see figure 'business and plumbing layer' one which is business related and second which is more technical related. You can see the plumbing layer consisting of data access layer , AJAX , yes more of technical stuff.  
  


Figure: - Business layer and plumbing layer  
  
**(I) what's the difference between services and components?**  
  
Services are logical grouping of components to achieve business functionality. Components are implementation approaches to make a service. The components can be in JAVA, C#, C++ but the services will be exposed in a general format like Web Services.

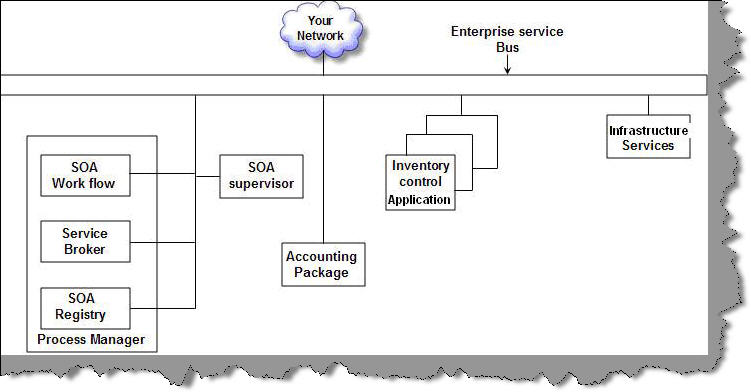
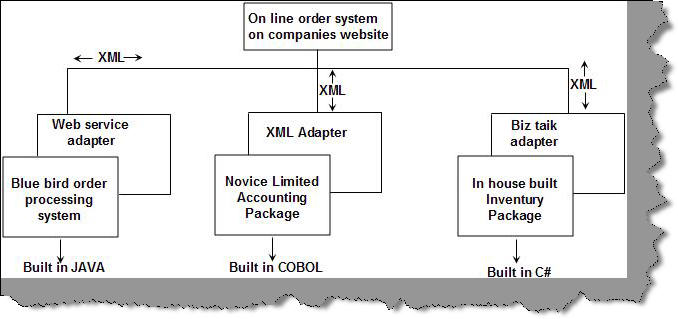
**(A) Can you describe the complete architecture of SOA?**  
  
Figure 'Architecture of SOA' shows a complete view of a SOA. Please note this architecture diagram is not tied up with implementation of Microsoft, IBM etc. It's a general architecture. Any vendor who implements SOA needs to fulfill the below SOA components. How they do it is completely their own technological implementation.  
  


Figure: - Architecture of SOA  
  
The main goal of SOA is to connect disparate systems. In order that these disparate system work they should messages to each other. ESB (Enterprise service bus) acts like a reliable post office which guarantees delivery of messages between systems in a loosely coupled manner. ESB is a special layer which delivers messages between applications. In the figure we have shown a huge plump pipe. It's not hardware or some wire etc. It's a group of components/software which helps you to send and receive messages between the disparate applications. Do not try to code your own ESB, you can think of buying one from Microsoft, IBM, Oracle, progress etc.   
  
***SOA registry*** is like a reference database of services. It describes what each services do, where are they located and how can they communicate. It's a central reference of meta-data for services.  
  
***SOA workflow*** allows us to define work flow using the services in SOA registry. We will read more about BPM in the further questions.  
  
***Service broker*** reads the work flow and takes services from the SOA registry and ties them together. Service brokers are normally middleware like EAI (Enterprise application Integration) products. You can get a list of decent EAI from Sun, Microsoft, and IBM etc.  
  
Process manager is nothing but the collection of SOA registry, SOA workflow and service broker.  
  
***SOA supervisor*** is traffic cop ensuring that services do not have issues. It deals mainly with performance issues of the system so that appropriate service levels are met. If any of the services have performance problems it sends messages to the proper infrastructure to fix the issue.  
  
**Note:** - The above explanation is of general architecture for SOA. Any vendor (Microsoft, IBM, SUN etc) who gives solution for SOA should have the above components in some or other manner. As this is a Software architecture book, we will not be covering specific vendor implementation. We would advise the reader to map the same to their vendor products for better understanding.

**(I) Can you explain a practical example in SOA?**

****  
  
**(I) What are ends, contract, address, and bindings?**  
  
These three terminologies on which SOA service stands. Every service must expose one or more ends by which the service can be available to the client. End consists of three important things where, what and how:-

* Contract (What)  
  Contract is an agreement between two or more parties. It defines the protocol how client should communicate with your service. Technically, it describes parameters and return values for a method.
* Address (Where)  
  An Address indicates where we can find this service. Address is a URL, which points to the location of the service.
* Binding (How)  
  Bindings determine how this end can be accessed. It determines how communications is done. For instance, you expose your service, which can be accessed using SOAP over HTTP or BINARY over TCP. So for each of these communications medium two bindings will be created.

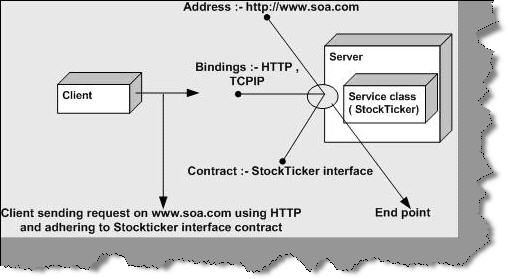
Below figure, show the three main components of end. You can see the stock ticker is the service class, which has an end hosted on www.soa.com with HTTP and TCP binding support and using Stock Ticker interface type.  
  


Figure: - Endpoint Architecture  
  
Note: - You can also remember the end point by ABC where A stands for Address, B for bindings and C for Contract.  
  
**(I) Are web-services SOA ?**  
  
SOA is a thinking, it's an architectural concept and web service is one of the technical approach to complete it. Web services are the preferred standards to achieve SOA.

* In SOA we need the services to be loosely coupled. A web service communicates using SOAP protocol which is XML based which is very loosely coupled. It answers the what part of the service.
* SOA services should be able to describe themselves. WSDL describes how we can access the service.
* SOA services are located in a directory. UDDI describes where we can get the web service. This nothing but implementation of SOA registry.

**Why SOA?**

Service is the important concept. Services can be published, discovered and used in a technology neutral, standard form by the set of protocols of the web services.

Other than being just architecture, SOA is the policies, practices, and frameworks by which it is ensure the right services are provided and consumed.

It becomes critical to implement processes that ensure that there are at least two different and separate processes— one for provider and the other for consumer, using SOA.

The Business Service Bus is starting point for developers that guide them to a coherent set that has been assembled for their domain. 

This is better than leaving developers to discover individual services and put them into context.

**What are the main benefits of SOA?**SOA helps create greater alignment between IT and line of business while generating more **flexibility** - IT flexibility to support greater business flexibility. Your business processes are changing faster and faster and global competition requires the flexibility that SOA can provide.   
SOA can help you get better reuse out of your existing IT investments as well as the new services you're developing today. SOA makes integration of your IT investments easier by making use of well-defined interfaces between services. SOA also provides an architectural model for integrating business partners’, customers’ and suppliers’ services into an enterprise’s business processes. This reduces cost and improves customer satisfaction

**Business Benefits of Service-Oriented Architecture**

SOA can help businesses respond more quickly and economically to changing market conditions.

SOA can be considered an architectural evolution. It captures many of the best practices of previous software architectures.

The goal of separating users from the service implementations is promoted by SOA.

The goals like increased interoperability, increased federation and increased business & technology domain alignment can be achieved by SOA due to its architectural and design discipline.

SOA is an architectural approach for constructing complex software-intensive systems from services.

SOA realizes its business and IT benefits through utilizing an analysis and design methodology when creating services.

**Challenges faced in SOA adoption**

One of the challenges faced by SOA is managing services metadata.

Second biggest challenge is the lack of testing in SOA space.

Another challenge is providing appropriate levels of security.

Interoperability is another important aspect in the SOA implementations.

Vendor hype concerns SOA because it can create expectations that may not be fulfilled.

**IT Benefits of Service-Oriented Architecture**

IT benefits of SOA are:

* The ability to build composite applications is provided.
* Business services are offered across the platforms.
* A self-healing infrastructure that reduces management costs is created.
* Location independence is provided
* Provides truly real-time decision-making applications.
* Reliability is enhanced
* It is not necessary that Services be at a particular system or network
* The approach is completely loosely coupled
* Hardware acquisition costs are reduced
* At every level there’s Authentication and authorization support
* Existing development skills are leveraged
* Provides a data bridge between incompatible technologies
* The search and connectivity to other services is dynamic

**What is SOA governance? What are its functions?**

Service-Oriented Architecture (SOA) governance is a concept used for activities related to exercising control over services in an SOA

Some key activities that are often mentioned as being part of SOA governance are:

* Managing the portfolio of services: This includes planning development of new services and updating current services.
* Managing the service lifecycle: This is meant to ensure that updates of services do not disturb current services to the consumers.
* Using policies to restrict behavior: Consistency of services can be ensured by having the rules applied to all the created services.
* Monitoring performance of services: The consequences of service downtime or underperformance can be severe because of service composition. Therefore action can be taken instantly when a problem occurs by monitoring service performance and availability.

**What is a reusable Service?**It is an autonomous, reusable, discoverable, stateless functionality that has the necessary granularity, and can be part of a composite application or a composite service.   
A reusable service should be identified with a business activity described by the service specifications (**design-time contract**).   
  
A service's constraints, including security, QoS, SLA, usage policies, may be defined by multiple run-time contracts, multiple interfaces (the WSDL for a SOAP Web Service), and multiple implementations (the code).   
  
A reusable service should be governed at the enterprise level throughout its entire lifecycle, from design-time through run-time. Its reuse should be promoted through a prescriptive process, and that reuse should be measured.

**Talking about Service identification, which approach between top-down and bottom-up methodologies encourages re-use and mantainance ?**Since the **top-down approach is business-driven** it can be practical to separate the different concerns of business and IT on different plans, providing a common ground in between. So in most situations it the most appropriate if you want to improve reuse and ROI in the medium/long term. Anyway

**How can you achieve loose coupling in a SOA?**One strategy for achieving loose coupling is to use the service interface (the WSDL for a SOAP Web Service) to limit this dependency, **hiding the service implementation from the consumer**. Loose coupling can be addressed by encapsulating the service functionalities in a manner that limits the impact of changes to the implementation on the service interface. However, at some point you will need to change the interface and manage versioning without impacting service consumers, in addition to managing multiple security constraints, multiple transports, and other considerations

**Do you recall any pattern which could be use to leverage loose coupling ?**The **Mediation** pattern, using an enterprise service bus (ESB), will help in achieving this.   
Mediation will take loose coupling to the highest level. It will establish independence between consumers and providers on all levels, including message formats, message types (including SOAP, REST, XML, binary) and transport protocols (including HTTP, HTTPS, JMS).   
Architecturally speaking this means the separation of concerns between consumers and providers on the transport, message type, and message format levels.

**The Service of a SOA should be engineered as stateless or stateful ?**Service should be **stateless**. It may have a context within its stateless execution, but it will not have an intermediary state waiting for an event or a call-back. The retention of state-related data must not extend beyond a request/response on a service. This is because state management consumes a lot of resources, and this can affect the scalability and availability that are required for a reusable service.

**What is composition of a Service ?**Composition is the process by which **services are combined to produce composite applications or composite services**. A composite application consists of the aggregation of services to produce an enterprise portal or enterprise process. A composite service consists of an aggregation of services that produces another reusable service. It's just like combining electronic components to create a computer motherboard, and then using that motherboard in a computer. Think of the motherboard as a reusable composite service that is a component of the computer, and of the computer as the composite application.

**How do I integrate my Legacy applications with SOA ?**Legacy applications are frequently at the core of your IT environment. With the right skills and tools, you need to identify discrete elements within your legacy applications and "wrap" them in standards-based interfaces and use them as services within your SOA.

**How does the ESB fits in this picture?**The Enterprise Service Bus is a core element of any SOA. ESBs provide the "any to any" connectivity between services within your own company, and beyond your business to connect to your trading partners. But SOA does not stop at just implementing an ESB. Depending on what your goals are, you may want to use an ESB to connect other services within your SOA such as information services, interaction services and business process management services. Additionally, you will need to consider development services and IT service management services. The SOA reference architecture can help you lay out an SOA environment that meets your needs and priorities. The ESB is part of this reference architecture and provides the backbone of an SOA but it should not be considered an SOA by itself.

**What are the common pitfalls of SOA ?**One of the most common pitfalls is to view SOA as an end, rather than a means to an end. Developers who focus on building an SOA solution rather than solving a specific business problem are more likely to create complex, unmanageable, and unnecessary interconnections between IT resources.

Another common pitfall is to try to solve multiple problems at once, rather than solving small pieces of the problem. Taking a top-down approach—starting with major organization-wide infrastructure investments—often fails either to show results in a relevant timeframe or to offer a compelling return on investment.

**What's the difference between services and components?**

Services are logical grouping of components to achieve business functionality. Components are implementation approaches to make a service. The components can be in JAVA, C#, C++ but the services will be exposed in a general format like Web Services.

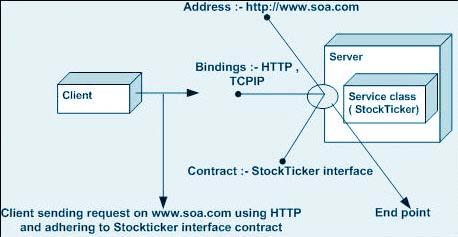
**What are ends, contract, address, and bindings?**

These three terminologies on which SOA service stands. Every service must expose one or more ends by which the service can be available to the client. End consists of three important things where, what and how:-

**Contract** is an agreement between two or more parties. It defines the protocol how client should communicate with your service. Technically, it describes parameters and return values for a method.

An **Address** indicates where we can find this service. Address is a URL, which points to the location of the service.

**Bindings** determine how this end can be accessed. It determines how communications is done. For instance, you expose your service, which can be accessed using SOAP over HTTP or BINARY over TCP. So for each of these communications medium two bindings will be created.  
Below figure, show the three main components of end. You can see the stock ticker is the service class, which has an end hosted on [**www.soa.com**](http://www.soa.com/) with HTTP and TCP binding support and using Stock Ticker interface type.



**The concept of SOA is nothing new, however why everyone started to talk about SOA only in the last years ?**Yes I agree the basic concept of SOA aren't new, however some technology technology changes in the last 10 years made service-oriented architecture more practical and applicable to more organizations than it was previously. Among this:

* Universally-accepted industry standards such as XML, its many variants, and Web-services standards have contributed to the renewed interest in SOA.
* Data governance frameworks, which are important to a successful SOA implementation, have well test and refined over the years.
* A variety of enabling technologies and tools (e.g., modeling, development, infrastructure/middleware, management, and testing) have matured.

Understanding of business and business strategies has grown, shifting attention from technology to the people, cultural changes, and process that are key business success factors.   
  
**What is the most important skill you need to adopt SOA ? technical or cultural ?  
Surely cultural**. SOA does require people to think of business and technology differently. Instead of thinking of technology first (e.g., If we implement this system, what kinds of things can we do with it?), practitioners must first think in terms of business functions, or services (e.g., My company does these business functions, so how can I set up my IT system to do those things for me most efficiently?).It is expected that adoption of SOA will change business IT departments, creating service-oriented (instead of technology-oriented) IT organizations.

**Is SOA really needed on your opinion?**

SOA is not for everyone. While SOA delivers significant benefits and cost savings, SOA does require disciplined enforcement of centralized governance principals to be successful. For some organizations, the cost of developing and enforcing these principals may be higher than the benefits realized, and therefore not a sound initiative.

**What is the Difference between coupling and cohesion?**

*Cohesion* of a single module/component is the degree to which its responsibilities form a meaningful unit; higher cohesion is better.

* Someone had vague reference to decomposability here. Clarification?
* How about: 'Cohesion is inversely proportional to the number of responsibilities a module/component has.'

*A component exhibits high cohesion when all its functions/methods are strongly related in terms of function.*

*Coupling* between modules/components is their degree of mutual interdependence; lower coupling is better.

* size: number of connections between routines
* intimacy: the directness of the connection between routines
* visibility: the prominence of the connection between routines
* flexibility: the ease of changing the connections between routines

*Two components are loosely coupled, when changes in one never or rarely necessitate a change in the other.*

A first-order principle of software architecture is to *increase cohesion* and *reduce coupling*. The higher cohesion and lower coupling a system has, in general the more its components exhibit strong data hiding, narrow but general interfaces and a high degree of flexibility.

**What are the advantages of loose coupling?**

* Make the code easier to read.
* Make our classes easier to consume by other developers by hiding the ugly inner workings of our classes behind well-designed APIs.
* Isolate potential changes to a small area of code.
* Reuse classes in completely new contexts.
* Make the code easier to set up and execute in unit tests. Loosely coupled code will make test setup much easier than code that is tightly coupled.

**What are some of the consequences of high coupling?**

* Changing code in one location can cause a ripple effect that affects code elsewhere.
* Developers / maintenance programmers need to understand potentially the whole system to be able to safely modify a single component.
* Changing requirements that affect the suitability of some component will potentially require wide ranging changes in order to accommodate a more suitable replacement component.
* More thought need to go into choices at the beginning of the lifetime of a software system in order to attempt to predict the long term requirements of the system because changes are more expensive.

**How to achieve loose coupling?**

Coupling has to do with dependencies between systems, which could be modules of code (functions, files, or classes), tools in a pipeline, server-client processes, and so forth. The less general the dependencies are, the more "tightly coupled" they become, since changing one system required changing the other systems that rely on it. The ideal situation is "loose coupling" where one system can be changed and the systems depending on it will continue to work without modification.

The general way to achieve loose coupling is through well defined interfaces. A common practice is to program to interfaces. Doing so ensures that code works against a contract, without knowing how that contract is implemented. That is abstraction in action.

Building on that, there are three ways to ensure loose coupling in your code, as listed below.

* service locator
* dependency injection
* inversion of control

If the interaction between two systems is well defined and adhered to on both sides, then it becomes easier to modify one system while ensuring that the conventions are not broken. It commonly occurs in practice that no well-defined interface is established, resulting in a sloppy design and tight coupling.

Some examples:

* Application depends on a library. Under tight coupling, app breaks on newer versions of the lib. Google for "DLL Hell".
* Client app reads data from a server. Under tight coupling, changes to the server require fixes on the client side.
* Two components are highly coupled when they depend on concrete implementation of each other.
* Two classes interact in an Object-Oriented hierarchy. Under tight coupling, changes to one class require the other class to be updated to match.
* Multiple command-line tools communicate in a pipe. If they are tightly coupled, changes to the version of one command-line tool will cause errors in the tools that read its output.

**Give an example of tight coupling from real life.**

*iPods* are a good example of tight coupling: once the battery dies you might as well buy a new iPod because the battery is soldered fixed and won't come loose, thus making replacing very expensive. A loosely coupled player would allow effortlessly changing the battery.

**What is inversion of control (IoC) principle or Hollywood Principle?**

Inversion of Control, or IoC, is an abstract principle describing an aspect of some software architecture designs in which the flow of control of a system is inverted in comparison to procedural programming.

Inversion of Control as a design guideline serves the following purposes:

* There is a decoupling of the execution of a certain task from implementation.
* Every system can focus on what it is designed for.
* Every system does not make assumptions about what other systems do or should do.
* Replacing systems will have no [side effect](http://effect) on other systems.

**What are the basic techniques to implement Inversion of Control?**

These are:

1. using a factory pattern
2. using a service locator
3. using a constructor injection
4. using a setter injection
5. using an interface injection

**What is Dependency injection** (**DI**)?

*Dependency injection (DI)* in object-oriented computer programming is a technique for supplying an external dependency (i.e. a reference) to a software component - that is, indicating to a part of a program which other parts it can use. It is a specific form of *inversion of control* where the concern being inverted is the process of obtaining the needed dependency. The term was first coined by Martin Fowler to describe the mechanism more clearly.

Fowler identifies three ways in which an object can get a reference to an external module, according to the pattern used to provide the dependency:[[3]](http://en.wikipedia.org/wiki/Dependency_injection#cite_note-2)

* *Type 1* or *interface injection*, in which the exported module provides an interface that its users must implement in order to get the dependencies at runtime.
* *Type 2* or *setter injection*, in which the dependent module exposes a setter method that the framework uses to inject the dependency.
* *Type 3* or *constructor injection*, in which the dependencies are provided through the class constructor.

The basic idea behind Dependency Injection is that you should isolate the implementation of an object from the construction of objects on which it depends. Dependency Injection is a form of the Inversion of Control Pattern where a factory object carries the responsibility for object creation and linking. The factory object ensures loose coupling between the objects and promotes seamless testability.

**What are the advantages and disadvantages of Dependency Injection?**

*Benefits*

The dependency injection pattern, also known as Inversion of Control, is one of the most popular design paradigms today. Thus, loose coupling helps to design a highly flexible application, which can be tailored to different environments with minimum re-engineering. It facilitates the design and implementation of *loosely coupled, reusable, and testable objects* in your software designs by removing dependencies that often inhibit reuse. Code becomes more testable because it abstracts and isolates class dependencies.

Dependency injection can help you design your applications so that the architecture links the components rather than the components linking themselves.

*Drawbacks*

Excessive or inappropriate use of dependency injection can make applications more complicated, harder to understand and more difficult to modify. Code that uses dependency injection can seem magical to some developers, since instantiation and initialization of objects is handled completely separately from the code that uses it. This separation can also result in problems that are hard to diagnose. Additionally, some dependency injection frameworks maintain verbose configuration files, requiring that a developer understand the configuration as well as the code in order to change it.

**Give an example of tight coupling in code where DI can be used.**  
When an object needs another object to operate properly, we say that the former is dependent on the latter. This behavior is transitive in nature. Consider three objects, namely, A, B, and C. If object A is coupled to object B, and B is in turn coupled to C, then object A is effectively coupled to object C—it is *dependent* on C. I've used the terms coupling and dependency interchangeably in this article.  
  
Objects can be coupled in two ways: tight coupling and loose coupling. When an object is loosely coupled with another object, you can change the coupling with ease; when the coupling is tight, the objects are not independently reusable and hence are difficult to use effectively in unit test scenarios.  
  
Here's an example of tight coupling. Consider two classes, C1 and C2, where C1 is tightly coupled with C2 and requires it to operate. In this case C1 is dependent on C2, as shown below:

public class C2

{

//Some code

}

public class C1

{

C2 bObject = new C2();

//Some code

}

The tight coupling between the two classes shown above occurs because C1 (which is dependent on C2) creates and contains an instance of the class C2. It's "tight" because you can eliminate or change the dependency only by modifying the container class (C1). This is where dependency injection fits in.

**What are the three ways to implement Dependency Injection in .NET Applications?**There are three common forms of dependency injection:

1. Constructor Injection
2. Setter Injection
3. Interface-based injection

Constructor injection uses parameters to inject dependencies. In setter injection, you use setter methods to inject the object's dependencies. Finally, in interface-based injection, you design an interface to inject dependencies. The following section shows how to implement each of these dependency injection forms and discusses the pros and cons of each.

|  |
| --- |
|  |

*Implementing Constructor Injection*I'll begin this discussion by implementing the first type of dependency injection mentioned in the preceding section—constructor injection. Consider a design with two layers; a BusinessFacade layer and the BusinessLogic layer. The BusinessFacade layer of the application depends on the BusinessLogic layer to operate properly. All the business logic classes implement an IBusinessLogic interface.  
  
With constructor injection, you'd create an instance of the BusinessFacade class using its argument or parameterized constructor and pass the required BusinessLogic type to inject the dependency. The following code snippet illustrates the concept, showing the BusinessLogic and BusinessFacade classes.

interface IBusinessLogic

{

//Some code

}

class ProductBL : IBusinessLogic

{

//Some code

}

class CustomerBL : IBusinessLogic

{

//Some code

}

public class BusinessFacade

{

private IBusinessLogic businessLogic;

public BusinessFacade(IBusinessLogic businessLogic)

{

this.businessLogic = businessLogic;

}

}

You'd instantiate the BusinessLogic classes (ProductBL or CustomerBL) as shown below:

IBusinessLogic productBL = new ProductBL();

Then you can pass the appropriate type to the BusinessFacade class when you instantiate it:

BusinessFacade businessFacade = new BusinessFacade(productBL);

Note that you can pass an instance of either BusinessLogic class to the BusinessFacade class constructor. The constructor does not accept a concrete object; instead, it accepts any class that implements the IBusinessLogic interface.  
  
Even though it is flexible and promotes loose coupling, the major drawback of constructor injection is that once the class is instantiated, you can no longer change the object's dependency. Further, because you can't inherit constructors, any derived classes call a base class constructor to apply the dependencies properly. Fortunately, you can overcome this drawback using the setter injection technique.

*Implementing Setter Injection*  
Setter injection uses properties to inject the dependencies, which lets you create and use resources as late as possible. It's more flexible than constructor injection because you can use it to change the dependency of one object on another without having to create a new instance of the class or making any changes to its constructor. Further, the setters can have meaningful, self-descriptive names that simplify understanding and using them. Here's an example that adds a property to the BusinessFacade class which you can use to inject the dependency.

|  |
| --- |
|  |

The following is now our BusinessFacade class with the said property.

public class BusinessFacade

{

private IBusinessLogic businessLogic;

public IBusinessLogic BusinessLogic

{

get

{

return businessLogic;

}

set

{

businessLogic = value;

}

}

}

The following code snippet illustrates to implement setter injection using the BusinessFacade class shown above.

IBusinessLogic productBL = new ProductBL();

BusinessFacade businessFacade = new BusinessFacade();

businessFacade.BusinessLogic = productBL;

The preceding code snippet uses the BusinessLogic property of the BusinessFacade class to set its dependency on the BusinessLogic type. The primary advantage of this design is that you can change the dependency between the BusinessFacade and the instance of BusinessLogic even after instantiating the BusinessFacade class.  
  
Even though setter injection is a good choice, its primary drawback is that an object with setters cannot be immutable—and it can be difficult to identify which dependencies are needed, and when. You should normally choose constructor injection over setter injection unless you need to change the dependency after instantiating an object instance, or cannot change constructors and recompile.  
  
*Implementing Interface Injection*You accomplish the last type of dependency injection technique, interface injection, by using a common interface that other classes need to implement to inject dependencies. The following code shows an example in which the classes use the IBusinessLogic interface as a base contract to inject an instance of any of the business logic classes (ProductBL or CustomerBL) into the BusinessFacade class. Both the business logic classes ProductBL and CustomerBL implement the IBusinessLogic interface:

interface IBusinessLogic

{

//Some code

}

class ProductBL : IBusinessLogic

{

//Some code

}

class CustomerBL : IBusinessLogic

{

//Some code

}

class BusinessFacade : IBusinessFacade

{

private IBusinessLogic businessLogic;

public void SetBLObject(IBusinessLogic businessLogic)

{

this.businessLogic = businessLogic;

}

}

In the code snippet above, the SetBLObject method of the BusinessFacade class accepts a parameter of type IBusinessLogic. The following code shows how you'd call the SetBLObject() method to inject a dependency for either type of BusinessLogic class:

IBusinessLogic businessLogic = new ProductBL();

BusinessFacade businessFacade = new BusinessFacade();

businessFacade.SetBLObject(businessLogic);

Or:

IBusinessLogic businessLogic = new CustomerBL();

BusinessFacade businessFacade = new BusinessFacade();

businessFacade.SetBLObject(businessLogic);

All three forms of dependency injection discussed in this article passed a *reference* to a BusinssLogic type rather than an *instance* of the type by using interfaces. According to Jeremy Weiskotten, a senior software engineer for Kronos:

"Coding to well-defined interfaces, particularly when using the dependency injection pattern, is the key to achieving loose coupling. By coupling an object to an interface instead of a specific implementation, you have the ability to use any implementation with minimal change and risk."

Dependency Injection can reduce the coupling between software components and it promises to become the paradigm of choice for designing loosely coupled, maintainable and testable objects. It can be used to abstract the dependencies of an object outside of it and make such objects loosely coupled with each other.

**How to achieve loose coupling through inversion of control principle?**

The Inversion of Control (IoC) pattern, also known as Dependency Injection, has recently become popular in the J2EE community. Several open source projects, including [Spring](http://new), [PicoContainer](http://new), and [HiveMind](http://new), use the IoC pattern to develop lightweight J2EE Containers.   
IoC is not a new concept, however. It has been around for several years now. Using object-oriented design principles and features such as interface, inheritance, and polymorphism, the IoC pattern enables better software design that facilitates *reuse, loose coupling, and easy testing* of software components.

Assume Class A has a relationship with Class B: it wants to use the services of Class B. The usual way to establish this relationship is to instantiate Class B inside Class A. Though this approach works, it creates tight coupling between the classes. You can't easily change Class B without modifying Class A. To eliminate the coupling, you can have a Configurator inject the instance of Class B (Object "b") to the instance of Class A (Object "a"). If you want to change the implementation of Class B later on, you simply change the Configurator object. *So, the control of how Object "a" gets the reference of Object "b" is inverted*. Object "a" is not responsible for getting the reference to Object "b". Instead, the Configurator is responsible for it. This is the basis for the IoC design pattern.

|  |  |
| --- | --- |
| Figure 1. Object "a" Directly Creates Object "b" |  |

To demonstrate the benefits of the Configurator object in this case, the following examples show a simple design without IoC and one with IoC.

Listing 1 and Figure 1 are simple examples in which Class A uses Class B:

public class A{

private B b;

public A(){

b=new B();

}

**Listing 1.** Class A Directly Refers Class B

Listing 1 assumes the following design decisions:

1. Class A needs a reference to Class B.
2. Class B is a concrete class that has a default constructor.
3. Class A owns the instance of class B.
4. No other class can access the instance of Class B.

|  |  |
| --- | --- |
| Figure 2. Object "a" First Creates Object "c" and Then Object "b" by Passing Object "c" |  |

If any one of the above design decisions change, then the code in Listing 1 must be modified. For example, if Class B changed to have a non-default constructor, which takes Class C (see Figure 2), then Listing 1 would change to Listing 2.

public class A{

private B b;

public A(){

C c=new C();

b=new B(c);

}

**Listing 2.** Class A Directly Refers Class B and Class C

Once again, Listing 2 assumes certain design decisions. Now Object "a" owns both Object "b" and Object "c". If Class B or Class C changes at all, then Class A needs to change as well. In essence, a simple design of a simple class with implicit design assumptions becomes a maintenance nightmare in the future. Consider how difficult making changes would be if you had this scenario in a typical application with several classes.

Alternatively, if you use a framework that uses the IoC pattern, the responsibility of creating Object "b" moves from Object "a" to the IoC framework, which creates and injects Object "b" into Object "a". This insulates Class A from the changes in Class B. Before Object "a" is used, it needs the reference to Object "b". The IoC framework will inject Object "b" into Object "a".

|  |  |
| --- | --- |
| Figure 3. IoC Framework Creates Object "b" and Sets It to Object "a" |  |

Listing 3 shows Class A from the previous listings modified to use the IoC pattern.

public class A{

private B b;

public A(){

}

public setB(B b){

this.b=b;

}

}

**Listing 3.** Class A Gets the Reference to Class B via setB

Listing 3 assumes the following design decisions:

1. A needs a reference to B, and it doesn't need to know how B is instantiated.
2. B can be an interface, an abstract class, or a concrete class.
3. Before the instance of Class A is used, it needs a reference to the instance of Class B.

From the above design decisions, you can see that there is no tight coupling between Class A and Class B. Both can be changed independently without affecting each other. Of course, if there is any change in the public methods of Class B, Class A needs to change as well. But how Object "b" is created and managed is not decided in the implementation of Object "a". Instead, the IoC framework uses the setB() method in Object "a" to inject Object "b" (see Figure 3).

**Give an example of cohesion in code.**

Cohesion is often mentioned with Coupling since they usually go hand-in-hand.  Cohesion refers to how closely related methods and class level variables are in a class.  A class with high cohesion would be one where all the methods and class level variables are used together to accomplish a specific task.  On the other end, a class with low cohesion is one where functions are randomly inserted into a class and used to accomplish a variety of different tasks.  Generally tight coupling gives low cohesion and loose coupling gives high cohesion.

The code below is of an EmailMessage class that has high cohesion.  All of the methods and class level variables are very closely related and work together to accomplish a single task.

*class EmailMessage*

*{*

*private string sendTo;*

*private string subject;*

*private string message;*

*public EmailMessage(string to, string subject, string message)*

*{*

*this.sendTo = to;*

*this.subject = subject;*

*this.message = message;*

*}*

*public void SendMessage()*

*{*

*// send message using sendTo, subject and message*

*}*

*}*

Now here is an example of the same class but this time as a low cohesive class.  This class was originally designed to send an email message but sometime in the future the user needed to be logged in to send an email so the Login method was added to the EmailMessage class.

*class EmailMessage*

*{*

*private string sendTo;*

*private string subject;*

*private string message;*

*private string username;*

*public EmailMessage(string to, string subject, string message)*

*{*

*this.sendTo = to;*

*this.subject = subject;*

*this.message = message;*

*}*

*public void SendMessage()*

*{*

*// send message using sendTo, subject and message*

*}*

*public void Login(string username, string password)*

*{*

*this.username = username;*

*// code to login*

*}*

*}*

The Login method and username class variable really have nothing to do with the EmailMessage class and its main purpose.  This class now has low cohesion and is probably not a good example to follow.

**What are the principles of class design?**

There are five principles of class design (**Remember: SOLID**). They are:

| **SRP** | [The Single Responsibility Principle](http://www.objectmentor.com/resources/articles/srp.pdf) | *A class should have one, and only one, reason to change.* |
| --- | --- | --- |
| **OCP** | [The Open Closed Principle](http://www.objectmentor.com/resources/articles/ocp.pdf) | *You should be able to extend a classes behavior, without modifying it.* |
| **LSP** | [The Liskov Substitution Principle](http://www.objectmentor.com/resources/articles/lsp.pdf) | *Derived classes must be substitutable for their base classes.* |
| **ISP** | [The Interface Segregation Principle](http://www.objectmentor.com/resources/articles/isp.pdf) | *Make fine grained interfaces that are client specific.* |
| **DIP** | [The Dependency Inversion Principle](http://www.objectmentor.com/resources/articles/dip.pdf) | *Depend on abstractions, not on concretions.* |

**Single Responsibility Principle**

**Define Single Responsibility Principle.**

In object-oriented programming, the **single responsibility principle** states that every object should have a single responsibility, and that responsibility should be entirely encapsulated by the class. All its services should be narrowly aligned with that responsibility.

It is based on the principle of cohesion.

**What is the motivation and intent of SRP?**

*Motivation*

In this context a responsibility is considered to be one reason to change. This principle states that if we have 2 reasons to change for a class, we have to split the functionality in two classes. Each class will handle only one responsibility and on future if we need to make one change we are going to make it in the class which handles it. When we need to make a change in a class having more responsibilities the change might affect the other functionality of the classes.

**The Single Responsibility Principle** is a simple and intuitive principle, but in practice it is sometimes hard to get it right.

*Intent*

A class should have only one reason to change.

**Give an example of SRP.**

Let's assume we need an object to keep an email message. We are going to use the IEmail interface from the below sample. At the first sight everything looks just fine. At a closer look we can see that our IEmail interface and Email class have 2 responsibilities (reasons to change). One would be the use of the class in some email protocols such as pop3 or imap. If other protocols must be supported the objects should be serialized in another manner and code should be added to support new protocols. Another one would be for the Content field. Even if content is a string maybe we want in the future to support HTML or other formats.

If we keep only one class, each change for a responsibility might affect the other one:

* Adding a new protocol will create the need to add code for parsing and serializing the content for each type of field.
* Adding a new content type (like html) make us to add code for each protocol implemented.

// single responsibility principle - bad example  
  
interface IEmail {  
 public void setSender(String sender);  
 public void setReceiver(String receiver);  
 public void setContent(String content);  
}  
  
class Email implements IEmail {  
 public void setSender(String sender) {// set sender; }  
 public void setReceiver(String receiver) {// set receiver; }  
 public void setContent(String content) {// set content; }  
}

We can create a new interface and class called IContent and Content to split the responsibilities. Having only one responsibility for each class give us a more flexible design:

* adding a new protocol causes changes only in the Email class.
* adding a new type of content supported causes changes only in Content class.

// single responsibility principle - good example  
interface IEmail {  
 public void setSender(String sender);  
 public void setReceiver(String receiver);  
 public void setContent(IContent content);  
}  
  
interface Content {  
 public String getAsString(); // used for serialization  
}  
  
class Email implements IEmail {  
 public void setSender(String sender) {// set sender; }  
 public void setReceiver(String receiver) {// set receiver; }  
 public void setContent(IContent content) {// set content; }  
}

*Conclusion*

The Single Responsibility Principle represents a good way of identifying classes during the design phase of an application and it reminds you to think of all the ways a class can evolve. A good separation of responsibilities is done only when the full picture of how the application should work is well understood.

**Give an example where SRP is violated and show it can be corrected?**

Given a DataAccess class that looks something like:

1. public class DataAccess
2. {
3. public class GetUser(string userName, string password)
4. {
5. using (var connection = new SqlConnection(...))
6. using (var command = new SqlCommand())
7. {
8. command.CommandTex = "SELECT Id, Name, UserName, Password, DateOfBirth FROM Users where UserName = @UserName and Password = @Password";
9. command.Parameters.Add("@UserName", SqlDbType.VarChar).Value = userName;
10. command.Parameters.Add("@Password", SqlDbType.VarChar).Value = password;
11. command.Connection = connection;
12. connection.Open();
13. using (var reader = command.ExecuteReader())
14. {
15. return dr.Read() ? MapUser(dr) : null;
16. }
17. }
18. }
20. private User MapUser(IDataReader dr)
21. {
22. return new User
23. {
24. Id = Convert.ToInt32(dr["Id"]),
25. Name = dr["Name"].ToString(),
26. UserName = dr["UserName"].ToString(),
27. Password = dr["Password"].ToString(),
28. DateOfBirth = Convert.ToDateTime(dr["DateOfBirth"]),
29. };
30. }
31. }

public class DataAccess

{

public class GetUser(string userName, string password)

{

using (var connection = new SqlConnection(...))

using (var command = new SqlCommand())

{

command.CommandTex = "SELECT Id, Name, UserName, Password, DateOfBirth FROM Users where UserName = @UserName and Password = @Password";

command.Parameters.Add("@UserName", SqlDbType.VarChar).Value = userName;

command.Parameters.Add("@Password", SqlDbType.VarChar).Value = password;

command.Connection = connection;

connection.Open();

using (var reader = command.ExecuteReader())

{

return dr.Read() ? MapUser(dr) : null;

}

}

}

private User MapUser(IDataReader dr)

{

return new User

{

Id = Convert.ToInt32(dr["Id"]),

Name = dr["Name"].ToString(),

UserName = dr["UserName"].ToString(),

Password = dr["Password"].ToString(),

DateOfBirth = Convert.ToDateTime(dr["DateOfBirth"]),

};

}

}

We have a violation of our single responsibility principle – DataAccess has two clear and distinct purposes: data access and data mapping. Now, it's true that data access and data mapping go hand-in-hand, but the unnecessary coupling of logic within a single class makes both functions (access and mapping) unnecessary brittle. By being in the same class, GetUser is tightly coupled to the mapping logic (which makes it more likely to be negatively effected by any changes).

The other problem I see people having when it comes to applying SRP is an irrational dislike for having multiple classes and a fear of object instantiation. At the very least, the solution to our above design problem is to create a new class, named DataMapper, and create an instance of it from within our DataAccess class. This would provide logical separation (which is what SRP is concerned about). However, we'd still have tight coupling and would likely consider introducing an IDataMapper interface and leverage dependency injection (other parts of SOLID focus on those aspects, so we'll skip the details for now). The point is that we'd introduce a new type (possibly two), and an extra object instantiation:

1. using (var reader = command.ExecuteReader())
2. {
3. return dr.Read() ? new DataMapper().MapUser(dr) : null;
4. }
6. //or
8. using (var reader = command.ExecuteReader())
9. {
10. return dr.Read() ? Factory.Create<IDataMapper>().MapUser(dr) : null;
11. }

using (var reader = command.ExecuteReader())

{

return dr.Read() ? new DataMapper().MapUser(dr) : null;

}

//or

using (var reader = command.ExecuteReader())

{

return dr.Read() ? Factory.Create<IDataMapper>().MapUser(dr) : null;

}

I consider both side effects extremely insignificant (non-existent really) in comparison to what is gained (readability, maintainability and testability).

In the end, the lesson here is that you should put thought into each and every one of your classes. That starts by first defining a very specific purpose for each class – regardless or how many .cs files you end up with. But don't think that SRP is something you can apply at the start and naturally maintain. As your system grows so too will your classes. At some point you'll notice that a class that once had a defined purpose seems to be a little murky. Do not hesitate to immediately refactor your code.

The Single Responsibility Principle represents a good way of identifying classes during the design phase of an application and it reminds you to think of all the ways a class can evolve. A good separation of responsibilities is done only when the full picture of how the application should work is well understand.

**Open Close Principle**

*Motivation*

A clever application design and the code writing part should take care of the frequent changes that are done during the development and the maintaining phase of an application. Usually, many changes are involved when a new functionality is added to an application. Those changes in the existing code should be minimized, since it's assumed that the existing code is already unit tested and changes in already written code might affect the existing functionality in an unwanted manner.

The Open Close Principle states that the design and writing of the code should be done in a way that new functionality should be added with minimum changes in the existing code. The design should be done in a way to allow the adding of new functionality as new classes, keeping as much as possible existing code unchanged.

*Intent*

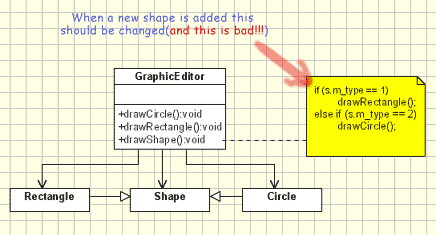
Software entities like classes, modules and functions should be open for extension but closed for modifications.

Example

Bellow is an example which violates the Open Close Principle. It implements a graphic editor which handles the drawing of different shapes. It's obviously that it does not follow the Open Close Principle since the GraphicEditor class has to be modified for every new shape class that has to be added. There are several disadvantages:

* for each new shape added the unit testing of the GraphicEditor should be redone.
* when a new type of shape is added the time for adding it will be high since the developer who add it should understand the logic of the GraphicEditor.
* adding a new shape might affect the existing functionality in an undesired way, even if the new shape works perfectly

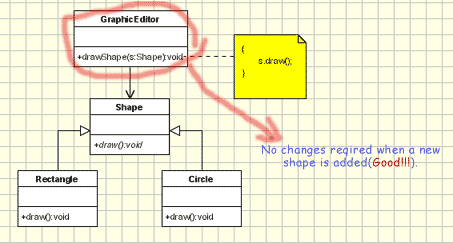
In order to have more dramatic effect, just imagine that the Graphic Editor is a big class, with a lot of functionality inside, written and changed by many developers, while the shape might be a class implemented only by one developer. In this case it would be great improvement to allow the adding of a new shape without changing the GraphicEditor class.



// Open-Close Principle - Bad example  
 class GraphicEditor {  
   
 public void drawShape(Shape s) {  
 if (s.m\_type==1)  
 drawRectangle(s);  
 else if (s.m\_type==2)  
 drawCircle(s);  
 }  
 public void drawCircle(Circle r) {....}  
 public void drawRectangle(Rectangle r) {....}  
 }  
   
 class Shape {  
 int m\_type;  
 }  
   
 class Rectangle extends Shape {  
 Rectangle() {  
 super.m\_type=1;  
 }  
 }  
   
 class Circle extends Shape {  
 Circle() {  
 super.m\_type=2;  
 }  
 }

Bellow is a example which supports the Open Close Principle. In the new design we use abstract draw() method in GraphicEditor for drawing objects, while moving the implementation in the concrete shape objects. Using the Open Close Principle the problems from the previous design are avoided, because GraphicEditor is not changed when a new shape class is added:

* no unit testing required.
* no need to understand the sourcecode from GraphicEditor.
* since the drawing code is moved to the concrete shape classes, it's a reduced risk to affect old functionallity when new functionallity is added.



// Open-Close Principle - Good example  
 class GraphicEditor {  
 public void drawShape(Shape s) {  
 s.draw();  
 }  
 }  
   
 class Shape {  
 abstract void draw();  
 }  
   
 class Rectangle extends Shape {  
 public void draw() {  
 // draw the rectangle  
 }  
 }

*Conclusion*

Like every principle OCP is only a principle. Making a flexible design involves additional time and effort spent for it and it introduce new level of abstraction increasing the complexity of the code. So this principle should be applied in those area which are most likely to be changed.

There are many design patterns that help us to extend code without changing it. For instance the Decorator pattern help us to follow Open Close principle. Also the Factory Method or the Observer pattern might be used to design an application easy to change with minimum changes in the existing code.

**Liskov's Substitution Principle(LSP)**

Motivation

All the time we design a program module and we create some class hierarchies. Then we extend some classes creating some derived classes.

We must make sure that the new derived classes just extend without replacing the functionality of old classes. Otherwise the new classes can produce undesired effects when they are used in existing program modules.

Likov's Substitution Principle states that if a program module is using a Base class, then the reference to the Base class can be replaced with a Derived class without affecting the functionality of the program module.

Intent

Derived types must be completely substitutable for their base types.

Example

Below is the classic example for which the Likov's Substitution Principle is violated. In the example 2 classes are used: Rectangle and Square. Let's assume that the Rectangle object is used somewhere in the application. We extend the application and add the Square class. The square class is returned by a factory pattern, based on some conditions and we don't know the exact what type of object will be returned. But we know it's a Rectangle. We get the rectangle object, set the width to 5 and height to 10 and get the area. For a rectangle with width 5 and height 10 the area should be 50. Instead the result will be 100

// Violation of Likov's Substitution Principle  
class Rectangle  
{  
 protected int m\_width;  
 protected int m\_height;  
  
 public void setWidth(int width){  
 m\_width = width;  
 }  
  
 public void setHeight(int height){  
 m\_height = height;  
 }  
  
  
 public int getWidth(){  
 return m\_width;  
 }  
  
 public int getHeight(){  
 return m\_height;  
 }  
  
 public int getArea(){  
 return m\_width \* m\_height;  
 }   
}  
  
class Square extends Rectangle   
{  
 public void setWidth(int width){  
 m\_width = width;  
 m\_height = width;  
 }  
  
 public void setHeight(int height){  
 m\_width = height;  
 m\_height = height;  
 }  
  
}  
  
class LspTest  
{  
 private static Rectangle getNewRectangle()  
 {  
 // it can be an object returned by some factory ...   
 return new Square();  
 }  
  
 public static void main (String args[])  
 {  
 Rectangle r = LspTest.getNewRectangle();  
   
 r.setWidth(5);  
 r.setHeight(10);  
 // user knows that r it's a rectangle. It assumes that he's able to set the width and height as for the base class  
  
 System.out.println(r.getArea());  
 // now he's surprised to see that the area is 100 instead of 50.  
 }  
}

Conclusion

This principle is just an extension of the Open Close Principle and it means that we must make sure that new derived classes are extending the base classes without changing their behavior.

**Interface Segregation Principle (ISP)**

Motivation

When we design an application we should take care how we are going to make abstract a module which contains several submodules. Considering the module implemented by a class, we can have an abstraction of the system done in an interface. But if we want to extend our application adding another module that contains only some of the submodules of the original system, we are forced to implement the full interface and to write some dummy methods. Such an interface is named fat interface or polluted interface. Having an interface pollution is not a good solution and might induce inappropriate behavior in the system.

The **Interface Segregation Principle** states that clients should not be forced to implement interfaces they don't use. Instead of one fat interface many small interfaces are preferred based on groups of methods, each one serving one submodule.

Intent

Clients should not be forced to depend upon interfaces that they don't use.

Example

Below is an example which violates the Interface Segregation Principle. We have a Manager class which represent the person which manages the workers. And we have 2 types of workers some average and some very efficient workers. Both types of workers works and they need a daily launch break to eat. But now some robots came in the company they work as well , but they don't eat so they don't need a launch break. One on side the new Robot class need to implement the IWorker interface because robots works. On the other side, the don't have to implement it because they don't eat.

This is why in this case the IWorker is considered a polluted interface.

If we keep the present design, the new Robot class is forced to implement the eat method. We can write a dummy class which does nothing(let's say a launch break of 1 second daily), and can have undesired effects in the application(For example the reports seen by managers will report more lunches taken than the number of people).

According to the Interface Segregation Principle, a flexible design will not have polluted interfaces. In our case the IWorker interface should be split in 2 different interfaces.

// interface segregation principle - bad example  
interface IWorker {  
 public void work();  
 public void eat();  
}  
  
class Worker implements IWorker{  
 public void work() {  
 // ....working  
 }  
 public void eat() {  
 // ...... eating in launch break  
 }  
}  
  
class SuperWorker implements IWorker{  
 public void work() {  
 //.... working much more  
 }  
  
 public void eat() {  
 //.... eating in launch break  
 }  
}  
  
class Manager {  
 IWorker worker;  
  
 public void setWorker(IWorker w) {  
 worker=w;  
 }  
  
 public void manage() {  
 worker.work();  
 }  
}

Following it's the code supporting the Interface Segregation Principle. By splitting the IWorker interface in 2 different interfaces the new Robot class is no longer forced to implement the eat method. Also if we need another functionality for the robot like recharging we create another interface IRechargeble with a method recharge.

// interface segregation principle - good example  
interface IWorker extends Feedable, Workable {  
}  
  
interface IWorkable {  
 public void work();  
}  
  
interface IFeedable{  
 public void eat();  
}  
  
class Worker implements IWorkable, IFeedable{  
 public void work() {  
 // ....working  
 }  
  
 public void eat() {  
 //.... eating in launch break  
 }  
}  
  
class Robot implements IWorkable{  
 public void work() {  
 // ....working  
 }  
}  
  
class SuperWorker implements IWorkable, IFeedable{  
 public void work() {  
 //.... working much more  
 }  
  
 public void eat() {  
 //.... eating in launch break  
 }  
}  
  
class Manager {  
 Workable worker;  
  
 public void setWorker(Workable w) {  
 worker=w;  
 }  
  
 public void manage() {  
 worker.work();  
 }  
}

## Conclusion

If the design is already done fat interfaces can be segregated using the Adapter pattern.

Like every principle Interface Segregation Principle is one principle which require additional time and effort spent to apply it during the design time and increase the complexity of code. But it produce a flexible design. If we are going to apply it more than is necessary it will result a code containing a lot of interfaces with single methods, so applying should be done based on experience and common sense in identifying the areas where extension of code are more likely to happens in the future.

**Dependency Inversion Principle**

Motivation

In an application we have low level classes which implement basic and primary operations and high level classes which encapsulate complex logic and rely on the low level classes. A natural way of implementing such structures would be to write low level classes and once we have them to write the complex high level classes. Since the high level classes are defined in terms of others this seems the logical way to do it. But this is not a flexible design. What happens if we need to replace a low level class?

Let's take the classical example of a copy module which read characters from keyboard and write them to the printer device. The high level class containing the logic is the Copy class. The low level classes are KeyboardReader and PrinterWriter.

In a bad design the high level class uses directly the low level classes. In this case if we want to change the design to direct the output to a new FileWriter class we have to change the Copy class. (Let's assume that it is a very complex class, with a lot of logic and realy hard to test).

In order to avoid such problems we can introduce an abstraction layer between the high level classes and low level classes. Since the high level modules contains the complex logic they should not depend on the low level modules and that the new abstraction layer should not be created based on low level modules. The low level modules are created based on the abstraction layer.

According to this principle the way of designing a class structure is to start from high level modules to the low level modules:  
High Level Classes --> Abstraction Layer --> Low Level Classes

Intent

* High-level modules should not depend on low-level modules. Both should depend on abstractions.
* Abstractions should not depend on details. Details should depend on abstractions.

Example

Below is an example which violates the Dependency Inversion Principle. We have the manager class which is a high level class, and the low level class Worker. We need to add a new module to our application because in the company there are some new specialized workers employed. We created a new class SuperWorker for this.

Let's assume that the Manager class is a complex one containing a very complex logic. And now we have to change it in order to introduce the new SuperWorker. Let's see the disadvantages:

* we have to change the Manager class (remember it is a complex one and this will involve some time and effort).
* some present functionality from the manager class might be affected.
* the unit testing should be redone.

All those problems will take a lot of time to solve. Now it would be very simple if the application was designed following the Dependency Inversion Principle. That means that we design the manager class, an IWorker interface and the Worker class implementing the IWorker interface. When we need to add the SuperWorker class all we have to do is implement the IWorker interface for it.

In order to have more dramatic effect, just imagine that the Graphic Editor is a big class, with a lot of functionallity inside, written and changed by many developpers, while the a shape might be a class implemented only by one developer. In this case it would be great improvment to allow the adding of a new shape without changing the GraphicEditor class.

// Dependency Inversion Principle - Bad example  
class Worker {  
 public void work() {  
 // ....working  
 }  
}  
  
class Manager {  
 Worker m\_worker;  
  
 public void setWorker(Worker w) {  
 m\_worker=w;  
 }  
  
 public void manage() {  
 m\_worker.work();  
 }  
}  
  
class SuperWorker {  
 public void work() {  
 //.... working much more  
 }  
}

Below is the code which supports the Dependency Inversion Principle. In this new design a new abstraction layer is added through the IWorker Interface. Now the problems from the above code are solved:

* Manager class should not be changed.
* minimized risk to affect old funtionallity present in Manager class.
* no need to redone the unit testing for Manager class.

// Dependency Inversion Principle - Good example  
interface IWorker {  
 public void work();  
}  
  
class Worker implements IWorker{  
 public void work() {  
 // ....working  
 }  
}  
  
class SuperWorker implements IWorker{  
 public void work() {  
 //.... working much more  
 }  
}  
  
class Manager {  
 IWorker m\_worker;  
  
 public void setWorker(IWorker w) {  
 m\_worker=w;  
 }  
  
 public void manage() {  
 m\_worker.work();  
 }  
}

Conclusion

When this principle is applied it means that the high level classes are not working directly with low level classes, they are using interfaces as an abstract layer. In that case the creation of new low level objects inside the high level classes(if necessary) can not be done using the operator new. Instead, some of the Creational design patterns can be used, such as Factory Method, Abstract Factory, Prototype.

The Template Design Pattern is an example where the DIP principle is applied.

Of course, using this principle implies an increased effort and a more complex code, but more flexible. This principle can not be applied for every class or every module. If we have a class functionality that is more likely to remain unchanged in the future there is not need to apply this principle.

**What are the principles of package design?**

There are about six principles that are about packages. In this context a package is a binary deliverable like a .jar file, or a dll as opposed to a namespace like a java package or a C++ namespace.  
  
The first three package principles are about package *cohesion*, they tell us what to put inside packages:

| **REP** | [The Release Reuse Equivalency Principle](http://www.objectmentor.com/resources/articles/granularity.pdf) | *The granule of reuse is the granule of release.* |
| --- | --- | --- |
| **CCP** | [The Common Closure Principle](http://www.objectmentor.com/resources/articles/granularity.pdf) | *Classes that change together are packaged together.* |
| **CRP** | [The Common Reuse Principle](http://www.objectmentor.com/resources/articles/granularity.pdf) | *Classes that are used together are packaged together.* |

The last three principles are about the couplings between packages, and talk about metrics that evaluate the package structure of a system.

| **ADP** | [The Acyclic Dependencies Principle](http://www.objectmentor.com/resources/articles/granularity.pdf) | *The dependency graph of packages must have no cycles.* |
| --- | --- | --- |
| **SDP** | [The Stable Dependencies Principle](http://www.objectmentor.com/resources/articles/stability.pdf) | *Depend in the direction of stability.* |
| **SAP** | [The Stable Abstractions Principle](http://www.objectmentor.com/resources/articles/stability.pdf) | *Abstractness increases with stability.* |

**What is the Reuse/Release Equivalence Principle and why is it important?**

The Reuse/Release Equivalence Principle (REP) says:

The unit of reuse is the unit of release. Effective reuse requires tracking of releases from a change

control system. The package is the effective unit of reuse and release.

*The unit of reuse is the unit of release*

Code should not be reused by copying it from one class and pasting it into another. If the original author fixes any bugs in the code, or adds any features, you will not automatically get the benefit. You will have

to find out what's changed, then alter your copy. Your code and the original code will gradually diverge.

Instead, code should be reused by including a released library in your code. The original author retains

responsibility for maintaining it; you should not even need to see the source code.

*Effective reuse requires tracking of releases from a change control system*

The author of a library needs to identify releases with numbers or names of some sort. This allows users of the library to identify different versions. This requires the use of some kind of release tracking system.

*The package is the effective unit of reuse and release*

It might be possible to use a class as the unit of reuse and release, however there are so many classes in a typical application, it would be burdensome for the release tracking system to keep track of them all. A

larger-scale entity is required, and the package fits this need well.

See also Robert Martin's article on [Granularity](http://www.objectmentor.com/resources/articles/granularity.pdf).

In general, sets of collaborating classes are reused

\_ package as the unit of reuse

• Only packages that are tested and released through a tracking system

can be effectively reused

– introducing changes for re-users in a controlled way

• If a package contains classes that should be reused,

then it should not contain classes that are not designed for reuse.

– either all classes in a package are reusable or none of them

– group classes in packages from the perspective of their reusers

Consequences

• only changes in classes interesting to the reuser will lead to a new

release of the package

• avoiding accidental reuse of classes not designed for re-use

• reduced effort for

– making releases

– upgrades at the reuser side

**The Common-Reuse Principle**

*“The classes in a package are reused together. If you reuse one of the*

*classes in a package, you reuse them all.”*

• If the user is only interested in a part of a package

– its code still depends on the whole package

– own code has to be revalidated on any new release of the used package

(even the change affects a class that is actually not used)

• Classes that tend to be reused together belong in the same package

(similar to Single-Responsibility Principle (SRP) for packages)

• Classes that are not tightly bound to each other with class

relationships should not be in the same package

• We want to make sure that the classes in a single package are

inseparable, i.e., it is impossible to depend on some and not the others

\_ high cohesion

The [Reuse/Release Equivalency Principle (REP)](http://ifacethoughts.net/2006/04/04/reuserelease-equivalency-principle/) stresses on the importance of a package (a set of cohesive classes) to be used for releasing reusable software. It also highlights the importance of releasing software that can be tracked, so that the users get a guarantee of interface and behaviour from the author.

However, it is very important to decide which classes should be bound together by a package. Improper selection of classes can create unwanted dependencies and turn usage of that package into a nightmare. Common Reuse Principle (CRP) lays down a good foundation to determine which classes should be packaged together:

The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all.

It says that only cohesive classes should be packaged together. As with everything, the packaging of classes and the cohesiveness should be defined from the user’s perspective. If a user uses a package, all the classes in that package should be reusable in the same context. A rule that can be a derivative of this is that all the classes related to a functionality should be packaged together. For example, if I am using a package for matrix calculations, it better have only those classes related to matrices and not trigonometric functions.

As specified by [Reuse/Release Equivalency Principle (REP)](http://ifacethoughts.net/2006/04/04/reuserelease-equivalency-principle/), even if one or two classes are being used, the granularity to be considered is the entire package. This dependency means that with every revision or release of that package, the software has to be tested and revalidated, irrespective of whether the modified class is being used or not. Hence, only those classes should be packaged together which will be reused together.

**The Common-Closure Principle**

*“The classes in a package should be closed together against the same*

*kinds of changes. A change that affects a package affects all the*

*classes in that package an no other packages.”*

• The Single-Responsibility Principle says that a **class** should not

contain multiple reasons to change

• Analogously, the Common-Closure principle says that a **package**

should not contain multiple reasons to change

\_ All classes that are likely to change for the same reason should be

packaged together

• Note: The Open-Closure Principle states that classes should be closed

for modification but open for extension

• Full closure is not attainable; but, the common-closure principle makes

the closure strategic by designing systems so that they are closed to

the **most common kinds of changes**

**Scenario Based Design Problems**

**We have two types of employees. Regular and contract employee. Regular employees will be paid on a fixed basis at the end of the month. Contract employees will be paid weekly based on the number of hours they worked.**

**Managers will be assigned to these employees for supervision. A manager may have regular and contract employees under him.**

**This application will calculate the payroll for these employees.**

**They asked me to come up with the class design for this situation.**

**What answer is the interviewer expecting from me?. Pointers in this direction will be highly appreciated.**

They are testing that you understand some basic tenets of good OO design. Specifically, they seem to be looking for:

1. An understanding of Polymorphism (by using functionality of each Employee at abstract base level)
2. An understanding of Specialisation (by extending the functionality of the base Employee class to produce different behaviour)

Design 1.

public class Employee  
{

   public abstract float CalCulatePayroll();     
}  
  
  
public class FullTimeEmp : Employee  
{  
   public override float CalCulatePayroll()  
   {  
   }  
}  
  
public class ContractEmp : Employee  
{  
  public int NoofHR  
      {get; set;}  
  
  public override float CalCulatePayroll()  
   {  
       sal = nohr\*money;  
   }  
}

Design 2.

public class Employee  
{  
  public bool isContractEmployee  
  { get; set;}  
  
  public int NoofHR  
  {get; set;}  
  
  
  public  float CalCulatePayroll()  
  {  
    if(this.isContractEmployee)  
    {  
      //calculate sal on based hr  
    }  
    else  
    {  
      //calculate regurlare sal  
    }  
  }  
}

Design 1 is correct solution in terms of maintainability and reusability.

**Give an example where you would use abstract class and not interface.**

**Class A has 10 methods. There are 5 methods which are related and there are other 5 which are also related. Class B and Class C inherits from class A. How do you re-factor the code to reduce coupling and increase cohesion in the design.**

**There is a software design principle that states “*Favour composition over inheritance*”. Explain with example.**

<http://blogs.msdn.com/b/steverowe/archive/2008/04/28/prefer-composition-over-inheritance.aspx>

*Composition* - Functionality of an object is made up of an aggregate of different classes.  In practice, this means holding a pointer to another class to which work is deferred.

*Inheritance* - Functionality of an object is made up of it's own functionality plus functionality from its parent classes.

For most non-trivial problems, there will be similar code needed by multiple classes.  It is not a wise idea to put the same code in more than one place (a topic for another day).  There are two strategies in object-oriented programming which attempt to solve the problem of duplicate code.  The one most popular in the early days was inheritance.  Shared functionality was implemented in a base class which allowed each child class to inherit that functionality.  A child would just not implement foo() and the parent would do the work.  This works, but it is not very flexible.

Suppose that the shared functionality is some kind of encryption algorithm.  Each child class will only inherit from one base class.  What if there is a for different encryption algorithms?  It would be possible to have multiple base classes, say AESEncryptionBase and DESEncryptionBase, but this necessitates multiple copies of the child classes--one for each base class.  With more than 2 base classes, this become untenable.  It also becomes very difficult to change out the encryption routine at runtime.  Doing so means creating a new object and copying the contents of the old object to it.

Another difficulty is the distortion of otherwise clean class hierarchies.  Each child should have an "is-a" relationship with its parent.  Is a music file and AESEncryptionBase?  No.  Here is a particularly telling examples from Smalltalk.  In [Squeak](http://www.squeak.org/) (the dominant open-source Smalltalk implementation), Semaphore inherits from LinkedList.  Is Semaphore a linked list?  No.  A linked list is used in the implementation, but a sempahore is not a specialization of linked lists.

A better approach is to contain the new functionality via composition.  A class should contain instances of objects it needs to utilize functionality from.  In the music file case, it would have a pointer to an EncryptionImpl class which might be AES, DES, or ROT13.  The class hierarchy will stay smaller and the music file implementation does not even need to be aware of which encryption method it is using.  In the Semaphore case, Semaphore would contain a LinkedList object which it would use to do the work.  Clients of Semaphore would not be expecting LinkedList functionality.  Extraneous methods would not need to be disabled.  Composition would also allow for more flexibility later.  If an implementation based on a heap or a prioritized queue were found to be advantageous, they could be without clients of Semaphore knowing.

Think twice before inheriting functionality.  There are times when it is a good idea such as when there is a logical default behavior and only some child classes need to over-ride it, but if the intent is to utilize the functionality rather than expose it to child class callers, composition is almost always the right decision.

**Why prefer composition over inheritance? What trade-offs are there for each approach? When should you choose inheritance over composition?**

*Prefer composition over inheritance as it is more malleable / easy to modify later, but do not use a compose-always approach.* With composition, it's easy to change behavior on the fly with Dependency Injection / Setters. Inheritance is more rigid as most languages do not allow you to derive from more than one type.. So the goose is more or less cooked once you derive from Class A.

My acid test for the above is:

* Does TypeB want to expose the complete interface (all public methods no less) of TypeA such that TypeB can be used where TypeA is expected? Indicates **Inheritance**.

e.g. A Cessna biplane will expose the complete interface of an airplane, if not more. So that makes it fit to derive from Airplane.

* Does TypeB only want only some/part of the behavior exposed by TypeA? Indicates need for**Composition.**

e.g. A Bird may need only the fly behavior of an Airplane. In this case, it makes sense to extract it out as an interface / class / both and make it a member of both classes.

**Update:** Just came back to my answer and it seems now that it is incomplete without a specific mention of Barbara Liskov's [Liskov Substitution Principle](http://en.wikipedia.org/wiki/Liskov_substitution_principle) as a test for 'Should I be inheriting from this type?'

**We had often heard that composition is better than inheritance. Why is it so? First of all how one is different from the other and what are the similarities in them?**  
  
Let’s say we are writing simulation software for Rocket Launching systems which are to be supplied to different countries. Now, these different countries can use them as they want it.  
The code for our launching system is below:

public class Launcher

{  
    public bool LaunchMissile()  
    {  
        Console.WriteLine("Missile launched");  
        return true;  
    }  
}

public class SufraceToAirMissileLauncher: Launcher

{

}

Now, country A uses this code to launch missile as follows:  
static void Main(string[] args)  
{  
        SufraceToAirMissileLauncher staLauncher = new SufraceToAirMissileLauncher();  
        bool isLaunched =   staLauncher.LaunchMissile();

}

This is how Inheritance is used. The various launchers can reuse the base Launcher class code to launch missile.  
  
The same thing can be achieved by using Composition where base class functionality is encapsulated inside the main concrete class. The code for that is below:

public class SufraceToAirMissileLauncher

{  
    private Launcher launcher = new Launcher();  
    public bool LaunchMissile()  
    {  
        return launcher.LaunchMissile();  
    }

}

The client UI code remains the same.  
  
Now, due to our superb code, our patented launching software had become famous and another country B wants to use it. But they had a condition that instead of launching the missile through base class they would want to get an instance of a missile. Now it's up to them what they want to do with it. They might add some nuclear material on it or modify it to increase its range or do whatever they might like. So another Missile object comes into the picture.

public class Missile

{  
    private bool isLaunched;  
    public bool IsLaunched  
    {  
        get { return isLaunched; }  
        set { isLaunched = value; }  
    }

    public Missile(bool isLaunched)  
    {  
        IsLaunched = IsLaunched;  
    }

}

And the base class function has changed to:

public class Launcher

{  
    public Missile LaunchMissile()  
    {  
        Console.WriteLine("Missile returned");  
        return new Missile(true);  
    }

}

Now, it returns a missile instead of launching it. So now if we rely on inheritance, the client code of country A would break since the method signature has changed from what is being used in its UI.  
  
However, if the country A had used composition instead, the code will not break. Only the derived class function would need to accommodate the new changed behaviour of the base class. To accommodate this, we need to change our derived class code function "LaunchMissile" as:

public class SufraceToAirMissileLauncher

{  
    private Launcher launcher = new Launcher();  
    public bool LaunchMissile()  
    {  
        Missile missile = launcher.LaunchMissile();  
        return missile.IsLaunched;  
    }

}

Hence, the client code of country A would still work:

static void Main(string[] args)  
{  
        SufraceToAirMissileLauncher staLauncher = new SufraceToAirMissileLauncher();  
        bool isLaunched =   staLauncher.LaunchMissile();

}

On the other hand country B which was insisting on getting a missile would still get missile from the base class.  
  
So through this simple example we see how the composition is favoured over inheritance to maintain compatibility and where there is a possibility that the functionality might change in future.

**Give another example to justify using composition over inheritance.**

A friend of mine recently asked me if I thought that having an inheritance hierarchy with a depth of 10 classes was acceptable or not. I don't think it is. If you take a class at the bottom, any change to the 9 classes above can introduce a breaking change. It can quickly become a maintenance nightmare!

Most of the time, if you have that kind of hierarchy in your application, it means that you chose inheritance over composition. It's probably also a code**-**smell indicating that your class is doing too much. To avoid that, let me show you how it is possible to extend a class without using inheritance.

Let's assume that we start with the following code. I know it's very basic, however it's still enough for our demonstration.

1. abstract class TripBase
2. {
3. private readonly string \_from;
4. private readonly string \_to;
6. public string From
7. {
8. get { return \_from; }
9. }
11. public string To
12. {
13. get { return \_to; }
14. }
16. protected TripBase(string from, string to)
17. {
18. \_from = from;
19. \_to = to;
20. }
22. public abstract DateTime CalculateEstimatedArrivalTime();
23. }
25. class CarTrip : TripBase
26. {
27. public CarTrip(string from, string to)
28. : base(from, to)
29. {
30. }
32. public override DateTime CalculateEstimatedArrivalTime()
33. {
34. return DateTime.Now.AddHours(15);
35. }
36. }
38. class PlaneTrip : TripBase
39. {
40. public PlaneTrip(string from, string to)
41. : base(from, to)
42. {
43. }
45. public override DateTime CalculateEstimatedArrivalTime()
46. {
47. return DateTime.Now.AddHours(2);
48. }
49. }

We have a BaseTrip abstract class and we want to extend it by having 2 sub classes: CarTrip and PlaneTrip. In a real application, we would have some kind of algorithm to calculate or fetch the transportation time and it would probably be very different for each class. In this example, to simplify, we just return a hard coded value.

This code is not bad, however it's not perfect either. For instance, are we sure that calculating the transportation time between 2 city is the responsibility of the CarTrip and the PlaneTrip class? As a matter of fact, if we want to add a new MotocycleTrip class that would have different properties but the same algorithm than CarTrip to calculate the transportation time, we would need to extract a new abstract super class that we would probably call RoadTrip... It doesn't look good...

Here is a re-factored version of this class, using composition instead of inheritance:

1. class Trip
2. {
3. private readonly string \_from;
4. private readonly string \_to;
5. private readonly ITransportationMode \_transportationMode;
7. public string From
8. {
9. get { return \_from; }
10. }
12. public string To
13. {
14. get { return \_to; }
15. }
17. public Trip(string from, string to, ITransportationMode transportationMode)
18. {
19. \_from = from;
20. \_to = to;
21. \_transportationMode = transportationMode;
22. }
24. public DateTime CalculateEstimatedArrivalTime()
25. {
26. return DateTime.Now.Add(\_transportationMode.CalculateTransportationTimeBetween(\_from, \_to));
27. }
28. }
30. internal interface ITransportationMode
31. {
32. TimeSpan CalculateTransportationTimeBetween(string from, string to);
33. }
35. class CarTransportationMode : ITransportationMode
36. {
37. public TimeSpan CalculateTransportationTimeBetween(string from, string to)
38. {
39. return [new](http://www.google.com/search?q=new+msdn.microsoft.com) TimeSpan(15, 0, 0);
40. }
41. }
43. class PlaneTransportationMode : ITransportationMode
44. {
45. public TimeSpan CalculateTransportationTimeBetween(string from, string to)
46. {
47. return [new](http://www.google.com/search?q=new+msdn.microsoft.com) TimeSpan(2, 0, 0);
48. }
49. }

In a few words, I've extracted the calculations in their own classes that implement an ITransportationMode interface. They are injected in the Trip class through the constructor (but we could also use a setter injection or a method injection).

This implementation improves the code in several ways:

* The calculation of the transportation time is done in a new class, it's not polluting our Trip class any more. It's a lot easier to add new algorithms; we just need to add a new implementation of ITransportationMode. It can evolve independently of the Trip class. It's called the separation of concern principle.
* We can more easily configure our Trip class. Before, wa had to create a new instance of a TripBase sub class to change the way the calculation was done, now we can just change the instance of ITransportationMode while keeping the same Trip object.
* We can have several subclasses of Trip using the same algorithm without introducing a new abstract class (such as RoadTrip). Therefore, composition helps us to keep our hierarchy of class flat.
* This is strategy pattern.

**I understand what composition is in OOP ,but I am not able to get a clear idea of what Aggregation is . Can someone explain?**

Simple rules:

1. A "owns" B = Composition : B has no meaning or purpose in the system without A
2. A "uses" B = Aggregation : B exists independently (conceptually) from A

Example 1:

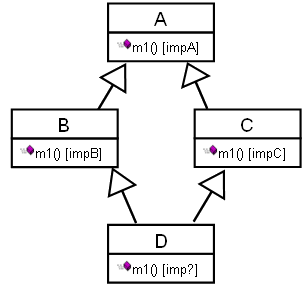
A Company is an aggregation of People. A Company is a composition of Accounts. When a Company ceases to do business its Accounts cease to exist but its People continue to exist.

Example 2: (very simplified)

A Text Editor owns a Buffer (composition). A Text Editor uses a File (aggregation). When the Text Editor is closed, the Buffer is destroyed but the File itself is not destroyed.

**Describe the classical problem with multiple inheritance.**

One of the most important problems of **multiple** **inheritance** is caused by this situation:



Class A has the m1() method. The [ImpA] means that method m1 has been implemented in A. Classes B and C inherit from A, and both of them redefine method m1. So class B has its own implementation of m1, denoted by [ImpB], and class C has its own implementation of m1 too, denoted by [ImpC].

Now class D inherits from B and C. The problem is... what implementation of m1 is used while running this code?

D d = new D();

d.m1();

...this code?

B d = new D();

d.m1();

...and this code?

C d = new D();

d.m1();

Languages and compilers supporting **multiple** **inheritance** solve this in one way or the other. But this makes compiling, debugging and understanding the code more difficult. Often the version of the implementation of m1 to be run is nearly unpredictable.

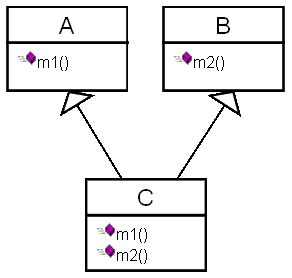
However, this pattern solves to certain degree that problem, because we can choose what implementation will be run. Assuming our D class uses our pattern for simulating **multiple** **inheritance**, there will be five different situations:

* D redefines m1 and we always want its implementation to be used: We just have to make D.BPart.m1() andD.CPart.m1() to call D.m1().
* We always want B's implementation: Then we make D.m1() and D.CPart.m1() call D.BPart.m1().
* We always want C's implementation: Then we make D.m1() and D.BPart.m1() call D.CPart.m1().
* We want B's implementation when m1 is called from a B type variable, C's implementation when m1 is called from a C type variable, and D's new implementation when called from a D type variable: Just redefine m1 in D class and let D.BPart and D.CPart alone. (This option is not recommendable except when we know what we are doing.)
* We want three new different implementations depending on the type of the variable that references our Dobject at call time: We write different implementations of m1 in D.m1(), D.BPart.m1(), and D.CPart.m1(). (This is even less recommendable than the previous one. But it could be useful for someone.)

**Most recent languages like C#, Java, Delphi, have simple inheritance. They don't have multiple inheritancebecause their designers had to choose between have it in and have all the problems it comes with, or get it out of the language putting away all those problems, and introduce a versatile and less problematic substitute like interfaces and interface inheritance.**

**How do you emulate multiple inheritances?**

Let's suppose we have two classes, A and B, and we want C to inherit from both of them.



 Collapse | [Copy Code](http://www.codeproject.com/Articles/10072/Simulated-Multiple-Inheritance-Pattern-for-C)

class A

{

m1();

}

class B

{

m2();

}

class C : A, B

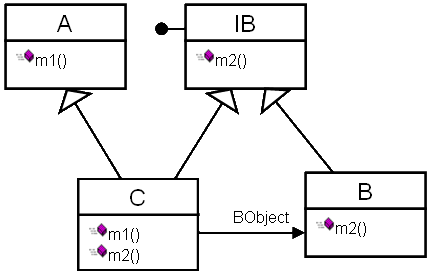
{

m1();

m2();

}

That code is impossible in C#. The following is the classical workaround to deal with this:



 Collapse | [Copy Code](http://www.codeproject.com/Articles/10072/Simulated-Multiple-Inheritance-Pattern-for-C)

class A

{

m1();

}

interface IB

{

m2();

}

class B : IB

{

m2();

}

class C : A, IB

{

B BObject;

m1();

m2() { BObject.m2(); }

}

In this code, we make class B to implement a new interface named IB that has identical methods as those of B. Then class C inherits from A and IB, and uses an internal B object that replicates B's m2 method implementation. So C.m2 in fact calls its BObject.m2 method. Then we can say C now has A's implementation of m1() and B's implementation of m2(). And we can use C object wherever we could use A objects or IB objects.

**But this solution has several problems.** One of them is the fact that we can't use C objects where explicitly B objects are expected, but only where IB objects are. So maybe we will have to change all the other code in the project, replacing references to B objects with references to IB objects. That will not be a big problem if we are designing the model and can take such decisions. But if the project is dependent on third party code or the standard library (the Framework) then we will be unable to make those modifications. Even more important, if class B is not ours, but it is from the standard library or from third party code, we can't make it implement our IB interface. We can't touch it.

**Describe the simulated multiple inheritance pattern for C# without using interfaces.**

**Or you have two classes A, B and class C want to inherit them without touching them. How do you achieve it?**

We have seen that we cannot **simulate** **multiple** **inheritance** completely using only interfaces and simple **inheritance**. We need something more, and C# happens to have that. Let's see.

We have these objectives:

* We want class C to inherit from classes A and B, being able to call their implementation of their methods with no need to rewrite them.
* We want to be able to use C objects wherever an A object or B object is expected.
* We don't want to modify A or B, for they are untouchable for one or another reason, or we simply don't care.
* We want to instantiate, reference, and use C objects just like normal objects.
* We work with parent classes that don't expose public fields, but use properties instead. However, this pattern will work even if one of the parent classes (at most) exposes public fields.
* Of course, this pattern works for three or more parents, not only two.

Here is the basic idea:

We will create two auxiliary classes, Aaux and Baux that inherit from A and B respectively.

class A

{

m1();

}

class B

{

m2();

}

class Aaux : A

{

m1();

}

class Baux : B

{

m2();

}

Our new class C won't inherit from A or B but have the same methods of both of them. Besides, it will contain two objects: one of type Aaux and the other of type Baux. We will call them C.APart and C.BPart respectively. C will use their implementations of m1 and m2 instead of rewriting them.

 Collapse | [Copy Code](http://www.codeproject.com/Articles/10072/Simulated-Multiple-Inheritance-Pattern-for-C)

class C

{

Aaux APart;

Baux BPart;

m1()

{

APart.m1();

}

m2()

{

BPart.m2();

}

}

So every C object has a pair of A and B objects inside. Let's make those objects know who is containing them by adding a reference to the C object that contains them. We will modify classes Aaux and Baux for this purpose:

 Collapse | [Copy Code](http://www.codeproject.com/Articles/10072/Simulated-Multiple-Inheritance-Pattern-for-C)

class Aaux : A

{

C CPart;

m1();

}

class Baux : B

{

C CPart;

m2();

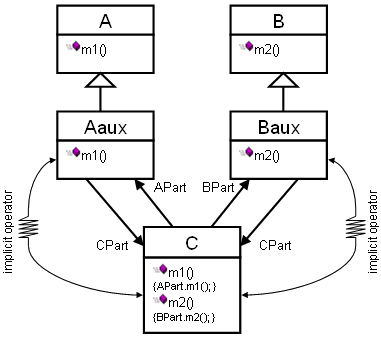
}

And finally we arrive at the final trick. We will redefine the implicit casting operator for class C, so:

* Whenever an A object is expected and a C object is found, C.APart is returned.
* Whenever a B object is expected and a C object is found, C.BPart is returned.

Again, we will redefine the implicit casting operator for class Aaux so whenever a C object is expected and an Aauxobject is found, Aaux.CPart is returned. Identically, we will redefine the implicit casting operator for class Baux so whenever a C object is expected and a Baux object is found, Baux.CPart is returned.

This is the final look:



 Collapse | [Copy Code](http://www.codeproject.com/Articles/10072/Simulated-Multiple-Inheritance-Pattern-for-C)

class Aaux : A

{

C CPart;

m1();

static implicit operator C(Aaux a)

{

return a.CPart;

}

}

class Baux : B

{

C CPart;

m2();

static implicit operator C(Baux b)

{

return b.CPart;

}

}

class C

{

Aaux APart;

Baux BPart;

m1()

{

APart.m1();

}

m2()

{

BPart.m2();

}

static implicit operator A(C c)

{

return c.APart;

}

static implicit operator B(C c)

{

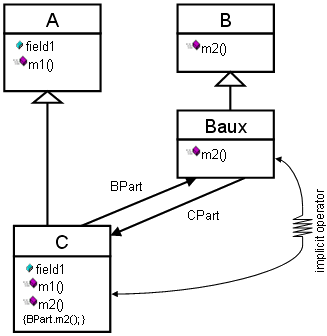
return c.BPart;

}

}

Now given that code, we can use C objects wherever an A or B object is expected, in addition to where a C object is expected. The only cost has been adding two extra classes and requiring the parent classes not to expose publicfields.

However, there is another step we can take that will allow us to reduce in on the number of extra classes required and will let one of the parent classes have public fields. In fact, the class diagram will be even simpler.



We only have to make C inherit directly from A, the class which exposes public fields.

Of course, properties are fully compatible with this pattern, since they behave like methods. So parent classes may have as many public properties as they wish.

## Using the code

Let's see an example. We are a computer dealer that buys computer hardware from major vendors and sells it to end users. However, we often get out of stock and we avoid losing customers by buying from rival computer shops and re-selling to our customers. We are not the only ones with that policy, so other shops often purchase our goods for selling them later. Our program has two arrays - in one of them we keep all our vendors, and in the other we keep all our customers. Rival shops are both vendors and customers. This would be the idea:

 Collapse | [Copy Code](http://www.codeproject.com/Articles/10072/Simulated-Multiple-Inheritance-Pattern-for-C)

class Vendor

{...}

class Customer

{...}

class Shop : Vendor, Customer

{...}

And this could be the final code (it's available in the download):

 Collapse | [Copy Code](http://www.codeproject.com/Articles/10072/Simulated-Multiple-Inheritance-Pattern-for-C)

/// *<summary>*

/// *A computer manufacturer. They resupply us.*

/// *</summary>*

public class Vendor

{

string id;

public string VendorId

{

get

{

return id;

}

set

{

id = value;

}

}

public Vendor(string vendorId)

{

id = vendorId;

}

public virtual void AskForRessuply()

{

Console.WriteLine("Please ressuply me, vendor "+id+".");

}

}

/// *<summary>*

/// *A customer. We send them their purchased goods.*

/// *</summary>*

public class Customer

{

string name;

public string Name

{

get

{

return name;

}

set

{

name = value;

}

}

public Customer(string customerName)

{

name = customerName;

}

public virtual void SendOrder()

{

Console.WriteLine("Dear "+name+": We are sending your goods.");

}

}

/// *<summary>*

/// *The auxiliary class that redefines Customer.*

/// *</summary>*

internal class CustomerAux : Customer

{

*// <--- It has a link to the Shop object that contains it.*

internal Shop shopPart;

internal CustomerAux(string customerName) : base (customerName)

{

}

*// We declare the implicit casting operator for returning*

*// shopPart when a Shop object is expected.*

static public implicit operator Shop(CustomerAux c)

{

return c.shopPart;

}

}

/// *<summary>*

/// *We consider a shop like a Vendor and a Customer,*

/// *for we both ask them to resupply us,*

/// *or they purchase our goods.*

/// *</summary>*

public class Shop : Vendor *// <-- It inheirs only from Vendor...*

{

CustomerAux customerPart; *// ...but has a CustomerAux object inside.*

*// Shops have an address in addition to the vendor id*

*// and the customer name inherited from Vendor and Customer.*

string address;

public string Address

{

get

{

return address;

}

set

{

address = value;

}

}

*// Here we are 'redirecting' property Name to the customerPart object.*

public string Name

{

get

{

return customerPart.Name;

}

set

{

customerPart.Name = value;

}

}

*// The Shop constructor.*

public Shop(string vendorId, string customerName, string shopAddress) :

base (vendorId)

{

*// We create and bind the CustomerAux object to this one.*

customerPart = new CustomerAux(customerName);

customerPart.shopPart = this;

address = shopAddress;

}

*// Here we are redirecting Customer.SendOrder to the customerPart object.*

public virtual void SendOrder()

{

customerPart.SendOrder();

}

*// We redefine the implicit casting operator for returning*

*// customerPart when a Customer object is expected.*

static public implicit operator Customer(Shop s)

{

return s.customerPart;

}

}

*// An example of use of the Vendor, Customer and Shop classes.*

class EntryPoint

{

static void Main(string[] args)

{

Vendor ibm = new Vendor("32FK-IBM");

Vendor hp = new Vendor("1138-HP");

Customer mrSimpson = new Customer("Mr. Simpson");

Customer mrGates = new Customer("Mr. Gates");

Shop joys =

new Shop("1979-JCS", "Joy's Computer Shop", "123, Fake St.");

Vendor[] vendors = {ibm, hp, joys};

foreach(Vendor ven in vendors)

ven.AskForRessuply();

Customer[] customers = {mrSimpson, mrGates, joys};

foreach(Customer cus in customers)

cus.SendOrder();

Console.ReadLine();

}

}