Introduction to Object Oriented Programming Concepts (OOP) and More
By Nirosh

e.

Popularity: 10.20 Rating: **4.68** out of 5

| 1 | 2 | 3 | 4 | 5 |

1. Introduction

I have noticed an increase in the number of articles published in the Architect category in code-project during the last few months. The number of readers for most of these articles is also high, though the ratings for the articles are not. This indicates that readers are interested in reading articles on Architecture, but the quality does not match their expectations. This article is a constructive attempt to group/ define/ explain all introductory concepts of software architecture for well seasoned developers who are looking to take their next step as system architects.

One day I read an article that said that the richest 2 percent own half the world's wealth. It also said that the richest 1 percent of adults owned 40 percent of global assets in the year 2000. And further, that the richest 10 percent of adults accounted for 85 percent of the world's total wealth. So there is an unbalanced distribution of wealth in the physical world. Have you ever thought of an unbalanced distribution of knowledge in the software world? According to my view point, the massive expansion of the software industry is forcing developers to use already implemented libraries, services and frameworks to develop software within ever shorter periods of time. The new developers are trained to use (I would say more often) already developed software components, to complete the development quicker. They just plug in an existing library and some how manage to achieve the requirements. But the sad part of the story is, that they never get a training to define, design the architecture for, and implement such components. As the number of years pass by, these developers become leads and also software architects. Their titles change, but the old legacy of not understanding, of not having any architectural experience continues, creating a vacuum of good architects. The bottom line is that only a small percentage of developers know how to design a truly object oriented system. The solution to this problem is getting harder every day as the aggressive nature of the software industry does not support an easy adjustment to existing processes, and also the related online teaching materials are either complex or less practical or sometimes even wrong. The most of them use impractical, irrelevant examples of shapes, animals and many other physical world entities to teach concepts of software architecture. There are only very few good business-oriented design references. Unfortunately, I myself am no exception and am a result of this very same system. I got the same education that all of you did, and also referred to the same resource set you all read.

Coming back to the initial point, I noticed that there is a knowledge gap, increasing every day, between the architects who know how to architect a system properly and the others who do not know. The ones, who know, know it right. But the ones, who do not know, know nothing. Just like the world's wealth distribution, it is an unbalanced distribution of knowledge.

2. Background

This article began after reading and hearing the questions new developers have, on basics of software architecture. There are some good articles out there, but still developers struggle to understand the basic concepts, and more importantly, the way to apply them correctly.

As I see it, newcomers will always struggle to understand a precise definition of a new concept, because it is always a new and hence unfamiliar idea. The one, who has experience, understands the meaning, but the one who doesn't, struggles to understand the very same definition. It is like that. Employers want experienced employees. So they say, you need to have experience to get a job. But how the hell is one supposed to have that experience if no one is willing to give him a job? As in the general case, the start with software architecture is no exception. It will be difficult. When you start to design your very first system, you will try to apply everything you know or learned from everywhere. You will feel that an interface needs to be defined for every class, like I did once. You will find it harder to understand when and when not to do something. Just prepare to go through a painful process. Others will criticize you, may laugh at you and say that the way you have designed it is wrong. Listen to them, and learn continuously. In this process you will also have to read and think a lot. I hope that this article will give you the right start for that long journey.

"*The knowledge of the actions of great men, acquired by long experience in contemporary affairs, and a continual study of antiquity*" – I read this phrase when I was reading the book named "*The Art of War*", seems applicable here, isn't it?

3. Prerequisites

This article is an effort to provide an accurate information pool for new developers on the basics of software architecture, focusing on **O**bject **O**riented **P**rogramming (*OOP*). If you are a developer, who has a minimum of three or more years of continuous development experience and has that hunger to learn more, to step-in to the next level to become a software architect, this article is for you.

4. The Main Content

4.1. What is Software Architecture?

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.

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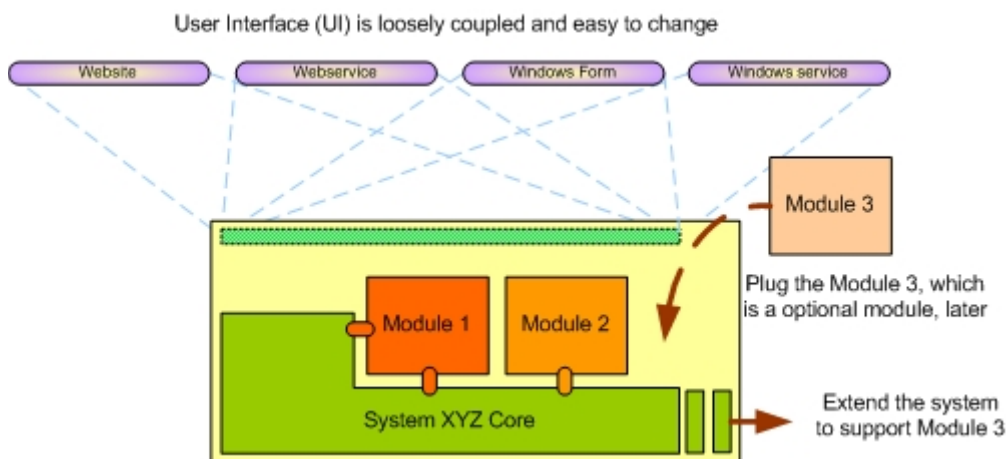
The primary goal of software architecture is to define the non-functional requirements of a system and define the environment. The detailed design is followed by a definition of how to deliver the functional behavior within the architectural rules. Architecture is important because it:

* Controls complexity
* Enforces best practices
* Gives consistency and uniformity
* Increases predictability
* Enables re-use.

4.3. What is OOP?

OOP is a design philosophy. It stands for Object Oriented Programming. **O**bject-**O**riented **P**rogramming (*OOP*) uses a different set of programming languages than old procedural programming languages (*C, Pascal*, etc.). Everything in *OOP* is grouped as self sustainable "*objects*". Hence, you gain re-usability by means of four main object-oriented programming concepts.

In order to clearly understand the object orientation, let's take your "*hand*" as an example. The "*hand*" is a class. Your body has two objects of type hand, named left hand and right hand. Their main functions are controlled/ managed by a set of electrical signals sent through your shoulders (through an interface). So the shoulder is an interface which your body uses to interact with your hands. The hand is a well architected class. The hand is being re-used to create the left hand and the right hand by slightly changing the properties of it.
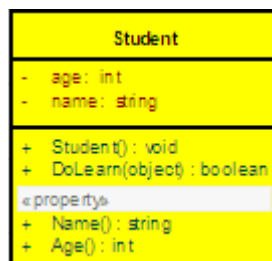
4.4. What is an Object?

An object can be considered a "*thing*" that can perform a set of **related** activities. The set of activities that the object performs defines the object's behavior. For example, the hand can grip something or a

Student (*object*) can give the name or address.

In pure *OOP* terms an object is an instance of a class.

4.5. What is a Class?

A *class* is simply a representation of a type of *object*. It is the blueprint/ plan/ template that describe the details of an *object*. A class is the blueprint from which the individual objects are created. *Class* is composed of three things: a name, attributes, and operations.

```
public class Student
{
}
```

According to the sample given below we can say that the *student* object, named *objectStudent*, has created out of the *Student* class.

```
Student objectStudent = new Student();
```

In real world, you'll often find many individual objects all of the same kind. As an example, there may be thousands of other bicycles in existence, all of the same make and model. Each bicycle has built from the same blueprint. In object-oriented terms, we say that the bicycle is an instance of the class of objects known as bicycles.

In the software world, though you may not have realized it, you have already used classes. For example, the *TextBox* control, you always used, is made out of the *TextBox* class, which defines its appearance and capabilities. Each time you drag a *TextBox* control, you are actually creating a new instance of the *TextBox* class.

4.6. 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 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.
- DIP - The Dependency Inversion Principle-
  Depend on abstractions, not on concretions.
- ISP - The Interface Segregation Principle-
  Make fine grained interfaces that are client specific.

For more information on design principles, please refer to Object Mentor.

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.

4.7. What is Encapsulation (or information hiding)?

The encapsulation is the inclusion within a program object of all the resources need for the object to function - basically, the methods and the data. In *OOP* the encapsulation is mainly achieved by creating classes, the classes expose public methods and properties. The class is kind of a container or capsule or a cell, which encapsulate the set of methods, attribute and properties to provide its indented functionalities to other classes. In that sense, encapsulation also allows a class to change its internal implementation without hurting the overall functioning of the system. That idea of encapsulation is to hide how a class does it but to allow requesting what to do.
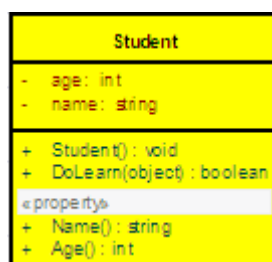
In order to modularize/ define the functionality of a one class, that class can uses functions/ properties exposed by another class in many different ways. According to Object Oriented Programming there are several techniques, classes can use to link with each other and they are named association, aggregation, and composition.

There are several other ways that an encapsulation can be used, as an example we can take the usage of an interface. The interface can be used to hide the information of an implemented class.

```
IStudent myStudent = new LocalStudent();
IStudent myStudent = new ForeignStudent();
```

According to the sample above (let's assume that *LocalStudent* and *ForeignStudent* are implemented by the *IStudent* interface) we can see how *LocalStudent* and *ForeignStudent* are hiding their, localize implementing information through the *IStudent* interface.
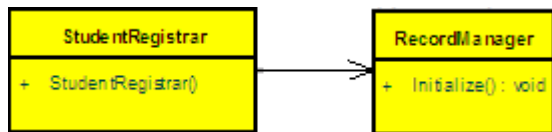
4.8. What is Association?

Association is a (*a*) relationship between two classes. It allows one object instance to cause another to perform an action on its behalf. Association is the more general term that define the relationship between two classes, where as the aggregation and composition are relatively special.

```
public class StudentRegistrar
{
    public StudentRegistrar ();
    {
        new RecordManager().Initialize();
    }
}
```

In this case we can say that there is an association between *StudentRegistrar* and *RecordManager* or there is a directional association from *StudentRegistrar* to *RecordManager* or StudentRegistrar use a (*****Use*****) *RecordManager*. Since a direction is explicitly specified, in this case the controller class is the *StudentRegistrar*.

To some beginners, association is a confusing concept. The troubles created not only by the association alone, but with two other *OOP* concepts, that is association, aggregation and composition. Every one understands association, before aggregation and composition are described. The aggregation or composition cannot be separately understood. If you understand the aggregation alone it will crack the definition given for association, and if you try to understand the composition alone it will always threaten the definition given for aggregation, all three concepts are closely related, hence must study together, by comparing one definition to another. Let's explore all three and see whether we can understand the differences between these useful concepts.

4.9. What is the difference between Association, Aggregation and Composition?

Association is a (*****a*****) relationship between two classes, where one class use another. But aggregation describes a special type of an association. Aggregation is the (*****the*****) relationship between two classes. When object of one class has an (*****has*****) object of another, if second is a part of first (containment relationship) then we called that there is an aggregation between two classes. Unlike association, aggregation always insists a direction.

```
public class University
{
    private Chancellor  universityChancellor = new Chancellor();
}
```
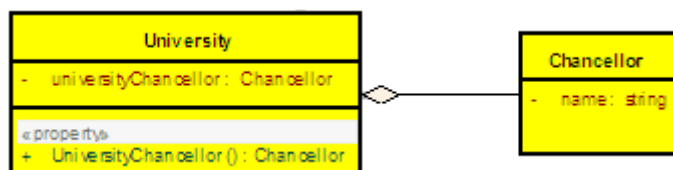

In this case I can say that *University* aggregate *Chancellor* or *University* has an (*****has-a*****) *Chancellor*. But even without a *Chancellor* a *University* can exists. But a *University* cannot exist without *Faculties*, the life time of a *University* attached with the life time of its *Faculty* (or Faculties). If *Faculties* are disposed the *University* will not exist or wise versa. In that case we called that *University* is composed of *Faculties*. So that composition can be recognized as a special type of an aggregation.
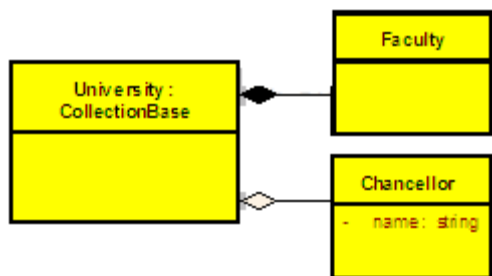
Same way, as another example, you can say that, there is a composite relationship in-between a *KeyValuePairCollection* and a *KeyValuePair*. The two mutually depend on each other.

.Net and Java uses the Composite relation to define their Collections. I have seen Composition is being used in many other ways too. However the more important factor, that most people forget is the life time factor. The life time of the two classes that has bond with a composite relation mutually depend on each other. If you take the .net Collection to understand this, there you have the Collection Element define inside (it is an inner part, hence called it is composed of) the Collection, farcing the Element to get disposed with the Collection. If not, as an example, if you define the Collection and it's Element to be independent, then the relationship would be more of a type Aggregation, than a Composition. So the point is, if you want to bind two classes with Composite relation, more accurate way is to have a one define inside the other class (making it a protected or private class). This way you are allowing the outer class to fulfill its purpose, while tying the lifetime of the inner class with the outer class.

So in summary, we can say that aggregation is a special kind of an association and composition is a special kind of an aggregation. (*Association->Aggregation->Composition*)
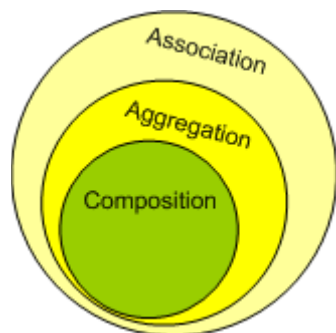
4.10. What is Abstraction and Generalization?

Abstraction is an emphasis on the idea, qualities and properties rather than the particulars (a suppression of detail). The importance of abstraction is derived from its ability to hide irrelevant details and from the use of names to reference objects. Abstraction is essential in the construction of programs. It places the emphasis on what an object is or does rather than how it is represented or how it works. Thus, it is the primary means of managing complexity in large programs.

While abstraction reduces complexity by hiding irrelevant detail, generalization reduces complexity by replacing multiple entities which perform similar functions with a single construct. Generalization is the broadening of application to encompass a larger domain of objects of the same or different type. Programming languages provide generalization through variables, parameterization, generics and *polymorphism*. It places the emphasis on the similarities between objects. Thus, it helps to manage complexity by collecting individuals into groups and providing a representative which can be used to specify any individual of the group.

Abstraction and generalization are often used together. Abstracts are generalized through parameterization to provide greater utility. In parameterization, one or more parts of an entity are replaced with a name which is new to the entity. The name is used as a parameter. When the parameterized abstract is invoked, it is invoked with a binding of the parameter to an argument.

4.11. What is an Abstract class?

Abstract classes, which declared with the abstract keyword, cannot be instantiated. It can only be used as a super-class for other classes that extend the abstract class. Abstract class is the concept and implementation gets completed when it is being realized by a subclass. In addition to this a class can inherit only from one abstract class (but a class may implement many interfaces) and must override all its abstract methods/ properties and may override virtual methods/ properties.

Abstract classes are ideal when implementing frameworks. As an example, let's study the abstract class named *LoggerBase* below. Please carefully read the comments as it will help you to understand the reasoning behind this code.

```csharp
public abstract class LoggerBase
{
    /// <summary>
    /// field is private, so it intend to use inside the class only
    /// </summary>
    private log4net.ILog logger = null;

    /// <summary>
    /// protected, so it only visible for inherited class
    /// </summary>
    protected LoggerBase()
    {
        // The private object is created inside the constructor
        logger = log4net.LogManager.GetLogger(this.LogPrefix);
        // The additional initialization is done immediately after
        log4net.Config.DOMConfigurator.Configure();
    }

    /// <summary>
    /// When you define the property as abstract,
    /// it forces the inherited class to override the LogPrefix
    /// So, with the help of this technique the log can be made,
    /// inside the abstract class itself, irrespective of it origin.
    /// If you study carefully you will find a reason for not to have "set" method here.
    /// </summary>
    protected abstract System.Type LogPrefix
    {
        get;
    }

    /// <summary>
    /// Simple log method,
    /// which is only visible for inherited classes
    /// </summary>
    /// <param name="message"></param>
    protected void LogError(string message)
    {
        if (this.logger.IsErrorEnabled)
        {
            this.logger.Error(message);
        }
    }

    /// <summary>
    /// Public properties which exposes to inherited class
    /// and all other classes that have access to inherited class
    /// </summary>
    public bool IsThisLogError
    {
        get
        {
            return this.logger.IsErrorEnabled;
        }
    }
}
```

The idea of having this class as an abstract is to define a framework for exception logging. This class will allow all subclass to gain access to a common exception logging module and will facilitate to easily replace the logging library. By the time you define the *LoggerBase*, you wouldn't have an idea about other modules of the system. But you do have a concept in mind and that is, if a class is going to log an exception, they have to inherit the *LoggerBase*. In other word the *LoggerBase* provide a framework for exception logging.

Let's try to understand each line of the above code.

Like any other class, an abstract class can contain fields, hence I used a private field named logger declare the *ILog* interface of the famous log4net library. This will allow the *Loggerbase* class to control, what to use, for logging, hence, will allow changing the source logger library easily.

The access modifier of the constructor of the *LoggerBase* is protected. The public constructor has no use when the class is of type abstract. The abstract classes are not allowed to instantiate the class. So I went for the protected constructor.

The abstract property named LogPrefix is an important one. It enforces and guarantees to have a value for *LogPrefix* (*LogPrefix* uses to obtain the detail of the source class, which the exception has occurred) for every subclass, before they invoke a method to log an error.

The method named *LogError* is protected, hence exposed to all subclasses. You are not allowed or rather you cannot make it public, as any class, without inheriting the *LoggerBase* cannot use it meaningfully.

Let's find out why the property named *IsThisLogError* is public. It may be important/ useful for other associated classes of an inherited class to know whether the associated member logs its errors or not.

Apart from these you can also have virtual methods defined in an abstract class. The virtual method may have its default implementation, where a subclass can override it when required.

All and all, the important factor here is that all *OOP* concepts should be used carefully with reasons, you should be able to logically explain, why you make a property a public or a field a private or a class an abstract. Additionally, when architecting frameworks, the *OOP* concepts can be used to forcefully guide the system to be developed in the way framework architect's wanted it to be architected initially.

4.12. What is an Interface?

In summary the Interface separates the implementation and defines the structure, and this concept is very useful in cases where you need the implementation to be interchangeable. Apart from that an interface is very useful when the implementation changes frequently. Some say you should define all classes in terms of interfaces, but I think recommendation seems a bit extreme.

Interface can be used to define a generic template and then one or more abstract classes to define partial implementations of the interface. Interfaces just specify the method declaration (implicitly public and abstract) and can contain properties (which are also implicitly public and abstract). Interface definition begins with the keyword interface. An interface like that of an abstract class cannot be instantiated.

If a class that implements an interface does not define all the methods of the interface, then it must be declared abstract and the method definitions must be provided by the subclass that extends the abstract class. In addition to this an interfaces can inherit other interfaces.

The sample below will provide an interface for our *LoggerBase* abstract class.

```
public interface ILogger
{
    bool IsThisLogError { get; }
}
```

4.13. What is the difference between a Class and an Interface?

In .Net/ C# a *class* can be defined to implement an *interface* and also it supports multiple implementations. When a *class* implements an *interface*, an *object* of such *class* can be encapsulated inside an *interface*.

If *MyLogger* is a class, which implements *ILogger,* there we can write

```
ILogger log = new MyLogger();
```

A *class* and an *interface* are two different types (conceptually). Theoretically a *class* emphasis the idea of encapsulation, while an *interface* emphasis the idea of abstraction (by suppressing the details of the implementation). The two poses a clear separation from one to another. Therefore it is very difficult or rather impossible to have an effective meaningful comparison between the two, but it is very useful and also meaningful to have a comparison between an interface and an abstract class.

4.14. What is the difference between an Interface and an Abstract class?

There are quite a big difference between an *interface* and an *abstract class*, even though both look similar.

- Interface definition begins with a keyword interface so it is of type interface
- Abstract classes are declared with the abstract keyword so it is of type class

- Interface has no implementation, but they have to be implemented.
- Abstract class's methods can have implementations and they have to be extended.

- Interfaces can only have method declaration (implicitly public and abstract) and fields (implicitly public static)
- Abstract class's methods can't have implementation only when declared abstract.

- Interface can inherit more than one interfaces
- Abstract class can implement more than one interfaces, but can inherit only one class

- Abstract class must override all abstract method and may override virtual methods
- Interface can be used when the implementation is changing
- Abstract class can be used to provide some default behavior for a base class.
- Interface makes implementation interchangeable
- Interface increase security by hiding the implementation
- Abstract class can be used when implementing framework
- Abstract classes are an excellent way to create planned inheritance hierarchies and also to use as non-leaf classes in class hierarchies.

Abstract classes let you define some behaviors; they force your subclasses to provide others. For example, if you have an application framework, an abstract class can be used to provide the default implementation of the services and all mandatory modules such as event logging and message handling etc. This approach allows the developers to develop the application within the guided help provided by the framework.

However, in practice when you come across with some application-specific functionality that only your application can perform, such as startup and shutdown tasks etc. The abstract base class can declare virtual shutdown and startup methods. The base class knows that it needs those methods, but an abstract class lets your class admit that it doesn't know how to perform those actions; it only knows that it must initiate the actions. When it is time to start up, the abstract class can call the startup method. When the base class calls this method, it can execute the method defined by the child class.

4.15. What is Implicit and Explicit Interface Implementations?

As mentioned before .Net support multiple implementations, the concept of implicit and explicit implementation provide safe way to implement methods of multiple interfaces by hiding, exposing or preserving identities of each of interface methods, even when the method signatures are the same.

Let's consider the interfaces defined below.

```
interface IDisposable
{
    void Dispose();
}
```

Here you can see that the class *Student* has implicitly and explicitly implemented the method named *Dispose()* via *Dispose* and *IDisposable.Dispose*.

```
class Student : IDisposable
{
    public void Dispose()
    {
        Console.WriteLine("Student.Dispose");
    }

    void IDisposable.Dispose()
    {
        Console.WriteLine("IDisposable.Dispose");
    }
}
```

4.16. What is Inheritance?

Ability of a new class to be created, from an existing class by extending it, is called *inheritance*.
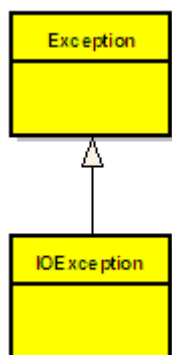

```
public class Exception
{
}
```

```
public class IOException : Exception
{
}
```

According to the above example the new class (*IOException*), which is called the derived class or subclass, inherits the members of an existing class (*Exception*), which is called the base class or super-class. The class *IOException* can extend the functionality of the class Exception by adding new types and methods and by overriding existing ones.

Just like abstraction is closely related with generalization, the inheritance is closely related with specialization. It is important to discuss those two concepts together with generalization to better understand and to reduce the complexity.

One of the most important relationships among objects in the real world is specialization, which can be described as the "**is-a**" relationship. When we say that a dog is a mammal, we mean that the dog is a specialized kind of mammal. It has all the characteristics of any mammal (it bears live young, nurses with milk, has hair), but it specializes these characteristics to the familiar characteristics of *canis domesticus*. A cat is also a mammal. As such, we expect it to share certain characteristics with the dog that are generalized in Mammal, but to differ in those characteristics that are specialized in cats.

The specialization and generalization relationships are both reciprocal and hierarchical. Specialization is just the other side of the generalization coin: Mammal generalizes what is common between dogs and cats, and dogs and cats specialize mammals to their own specific subtypes.

Similarly, as an example you can say that both *IOException* and *SecurityException* are of type Exception. They have all characteristics and behaviors of an Exception, That mean the *IOException* is a specialized kind of Exception. A *SecurityException* is also an *Exception*. As such, we expect it to share certain characteristic with *IOException* that are generalized in Exception, but to differ in those characteristics that are specialized in *SecurityExceptions*. In other words, Exception generalizes the shared characteristics of both *IOException* and *SecurityException*, while *IOException* and *SecurityException* specialize with their characteristics and behaviors.

In *OOP*, the specialization relationship is implemented using the principle called inheritance. This is the most common and most natural and widely accepted way of implement this relationship.

4.17. What is Polymorphisms?

Polymorphisms is a generic term that means 'many shapes'. More precisely *Polymorphisms* means the ability to request that the same operations be performed by a wide range of different types of things.

At times, I used to think that understanding Object Oriented Programming concepts have made it difficult since they have grouped under four main concepts, while each concept is closely related with one another. Hence one has to be extremely careful to correctly understand each concept separately, while understanding the way each related with other concepts.

In *OOP* the *polymorphisms* is achieved by using many different techniques named method overloading, operator overloading and method overriding,

4.18. What is Method Overloading?

The method overloading is the ability to define several methods all with the same name.

```
public class MyLogger
{
    public void LogError(Exception e)
    {
        // Implementation goes here
    }

    public bool LogError(Exception e, string message)
    {
        // Implementation goes here
    }
}
```

4.19. What is Operator Overloading?

The operator overloading (less commonly known as ad-hoc *polymorphisms*) is a specific case of *polymorphisms* in which some or all of operators like +, - or == are treated as polymorphic functions and as such have different behaviors depending on the types of its arguments.

```
public class Complex
{
    private int real;
    public int Real
    { get { return real; } }

    private int imaginary;
    public int Imaginary
    { get { return imaginary; } }

    public Complex(int real, int imaginary)
    {
        this.real = real;
        this.imaginary = imaginary;
    }

    public static Complex operator +(Complex c1, Complex c2)
    {
        return new Complex(c1.Real + c2.Real, c1.Imaginary + c2.Imaginary);
    }
}
```

I above example I have overloaded the plus operator for adding two complex numbers. There the two properties named Real and Imaginary has been declared exposing only the required "*get*" method, while the object's constructor is demanding for mandatory real and imaginary values with the user defined constructor of the class.

4.20. What is Method Overriding?

Method overriding is a language feature that allows a subclass to override a specific implementation of a method that is already provided by one of its super-classes.

A subclass can give its own definition of methods but need to have the same signature as the method in its super-class. This means that when overriding a method the subclass's method has to have the same name and parameter list as the super-class's overridden method.

```
using System;
public class Complex
{
    private int real;
    public int Real
    { get { return real; } }

    private int imaginary;
    public int Imaginary
    { get { return imaginary; } }

    public Complex(int real, int imaginary)
    {
        this.real = real;
        this.imaginary = imaginary;
    }

    public static Complex operator +(Complex c1, Complex c2)
    {
        return new Complex(c1.Real + c2.Real, c1.Imaginary + c2.Imaginary);
    }

    public override string ToString()
    {
        return (String.Format("{0} + {1}i", real, imaginary));
    }
}
```

In above example I have extended the implementation of the sample Complex class given under operator overloading section. This class has one overridden method named *"ToString"*, which override

the default implementation of the standard *"ToString"* method to support the correct string conversion of a complex number.

```
Complex num1 = new Complex(5, 7);
Complex num2 = new Complex(3, 8);

// Add two Complex numbers using the
// overloaded plus operator
Complex sum = num1 + num2;

// Print the numbers and the sum
// using the overriden ToString method
Console.WriteLine("({0}) + ({1}) = {2}", num1, num2, sum);
Console.ReadLine();
```

4.21. What is a Use case?

A use case is a thing an actor perceives from the system. A use case maps actors with functions. Importantly, the actors need not be people. As an example a system can perform the role of an actor, when it communicate with another system.
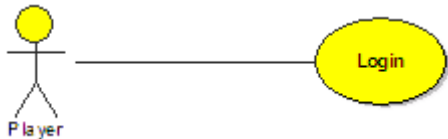
In another angle a use case encodes a typical user interaction with the system. In particular, it:

- Captures some user-visible function.
- Achieves some concrete goal for the user.

A complete set of use cases largely defines the requirements for your system: everything the user can see, and would like to do. The below diagram contains a set of use cases that describes a simple login module of a gaming website.
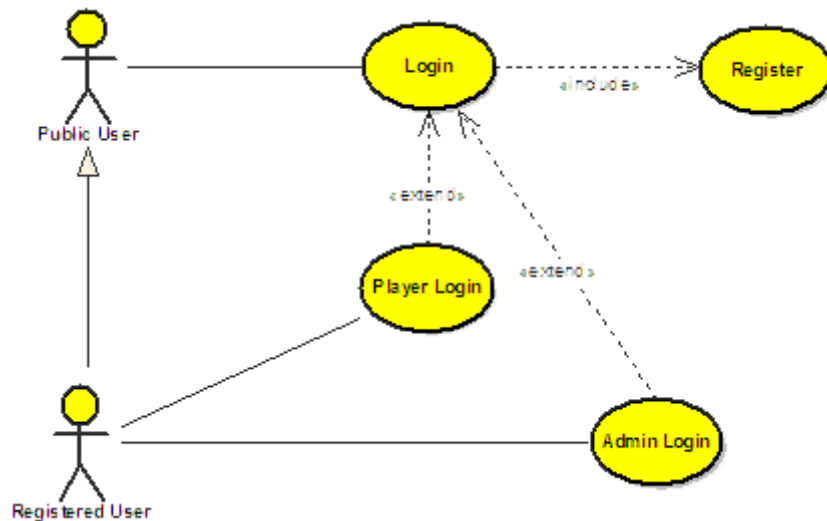

4.22. What is a Class Diagram?

A class diagrams are widely used to describe the types of objects in a system and their relationships. Class diagrams model class structure and contents using design elements such as classes, packages and objects. Class diagrams describe three different perspectives when designing a system, conceptual, specification, and implementation. These perspectives become evident as the diagram is created and help solidify the design.

The Class diagrams, physical data models, along with the system overview diagram are in my opinion

the most important diagrams that suite the current day rapid application development requirements.

UML Notations:
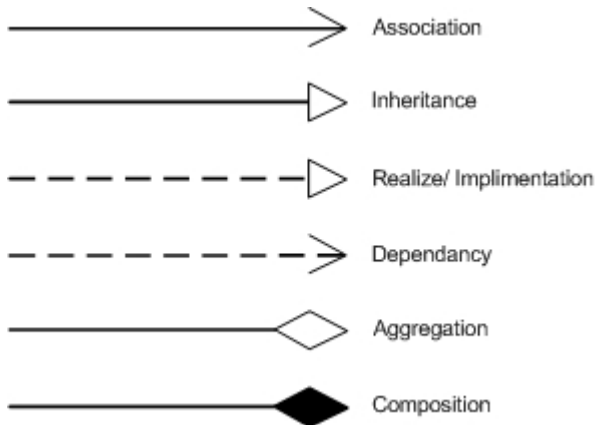
4.23. What is a Package Diagram?

Package diagrams are used to reflect the organization of packages and their elements. When used to represent class elements, package diagrams provide a visualization of the name-spaces. In my designs, I use the package diagrams to organize classes in to different modules of the system.

4.24. What is a Sequence Diagram?

A sequence diagrams model the flow of logic within a system in a visual manner, it enable both to document and validate your logic, and are used for both analysis and design purposes. Sequence diagrams are the most popular UML artifact for dynamic modeling, which focuses on identifying the behavior within your system.

4.25. 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.
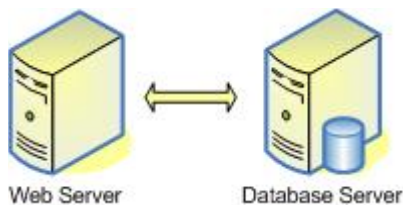
4.26. What is three-tier architecture?

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.
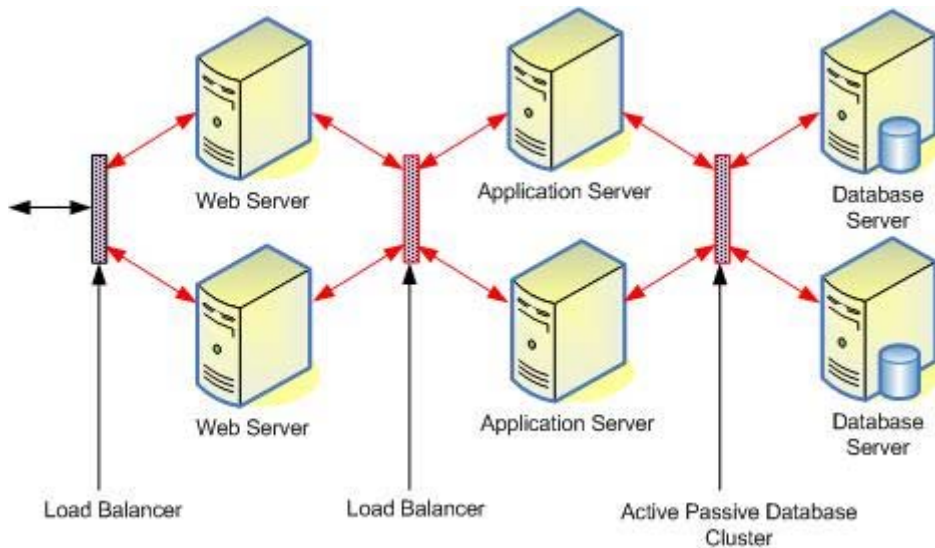

The 3-Tier architecture has the following three tiers.

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

4.27. What is MVC architecture?

The **M**odel-**V**iew-**C**ontroller (*MVC*) architecture separates the modeling of the domain, the presentation, and the actions based on user input into three separate classes.

Unfortunately, the popularity of this pattern has resulted in a number of faulty usages; each technology (*Java*, *ASP.NET* etc) has defined it in their own way making it difficult to understand. In particular, the term "controller" has been used to mean different things in different contexts. The definitions given bellow are the closes possible ones I found for ASP.NET version of *MVC*.
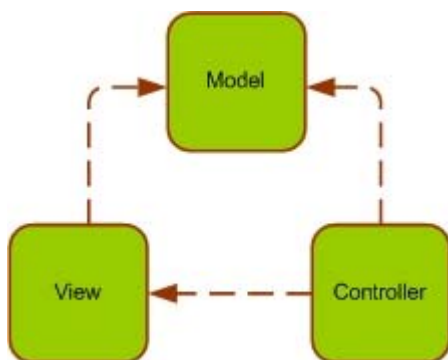
1. **Model**: *DataSet* and typed *DataSet* (some times business object, object collection, XML etc) are the most common use of the model.
2. **View**: The *ASPX* and *ASCX* files generally handle the responsibilities of the view.
3. **Controllers**: The handling of events or the controlling is usually done in the code-behind class.

In a complex n-tier distributed system the *MVC* architecture place the vital role of organizing the presentation tier of the system.

4.28. 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 Service Oriented Architecture (SOA) is being used to provide
centralized services
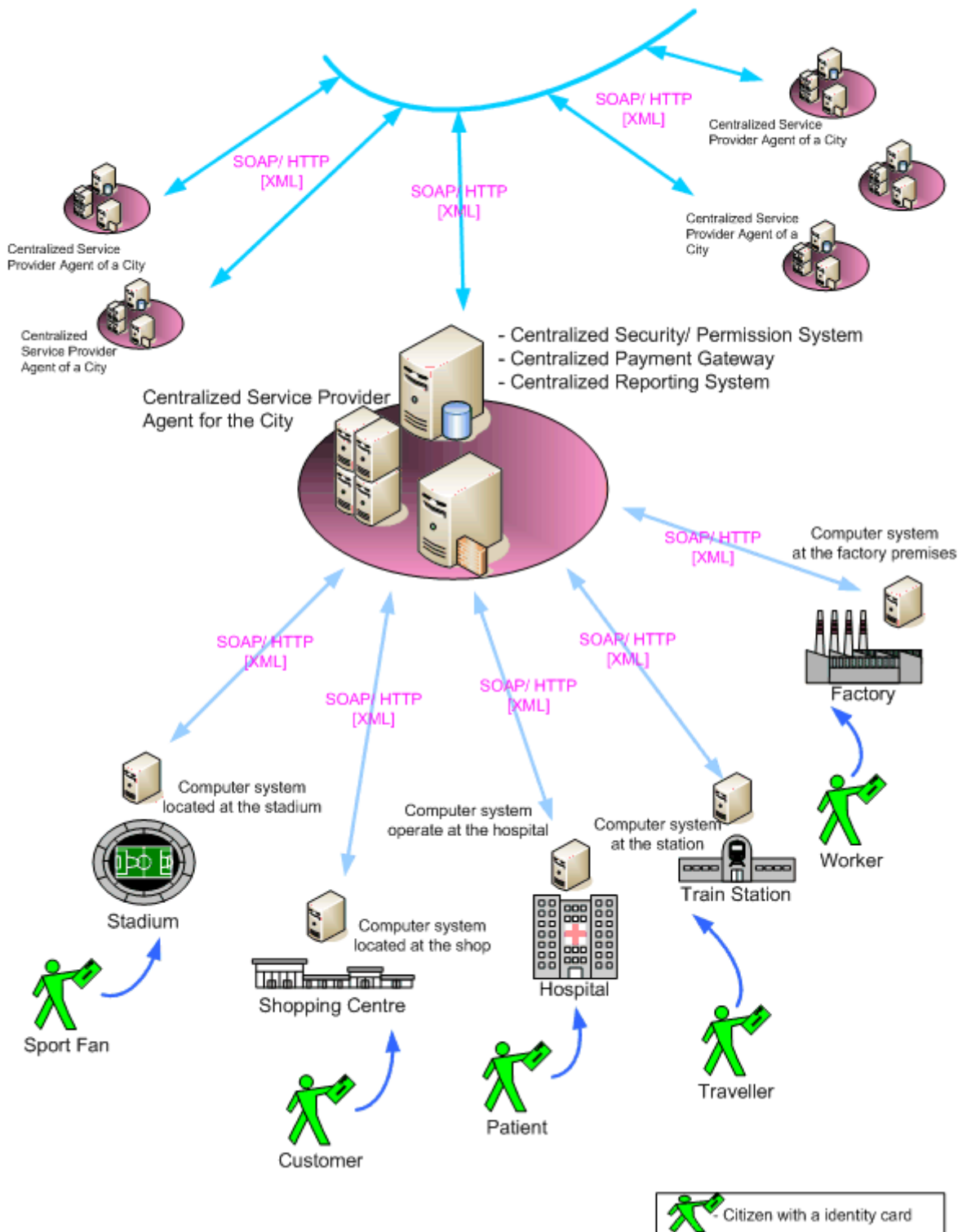
The SOA can be used as the concept to connect multiple systems to provide services. It has it's great
share in the future of the IT world.

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).

4.29. 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.

4.30. 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.

4.31. What is Gang of Four (GoF) Design Patterns?

The Gang of Four (GoF) patterns are generally considered the foundation for all other patterns. They are categorized in three groups: Creational, Structural, and Behavioral. Here you will find information on these important patterns.

Creational Patterns
- Abstract Factory Creates an instance of several families of classes
- Builder Separates object construction from its representation
- Factory Method Creates an instance of several derived classes
- Prototype A fully initialized instance to be copied or cloned
- Singleton A class of which only a single instance can exist

Structural Patterns
- Adapter Match interfaces of different classes
- Bridge Separates an object's interface from its implementation

- Composite A tree structure of simple and composite objects
- Decorator Add responsibilities to objects dynamically
- Facade A single class that represents an entire subsystem
- Flyweight A fine-grained instance used for efficient sharing
- Proxy An object representing another object

Behavioral Patterns

- Chain of Resp. A way of passing a request between a chain of objects
- Command Encapsulate a command request as an object
- Interpreter A way to include language elements in a program
- Iterator Sequentially access the elements of a collection
- Mediator Defines simplified communication between classes
- Memento Capture and restore an object's internal state
- Observer A way of notifying change to a number of classes
- State Alter an object's behavior when its state changes
- Strategy Encapsulates an algorithm inside a class
- Template Method Defer the exact steps of an algorithm to a subclass
- Visitor Defines a new operation to a class without change

4.32. What is the difference between Abstract Factory and Builder design patterns?

The two design patterns are fundamentally different. However, when you learn them for the first time, you will see a confusing similarity. So that it will make harder for you to understand them. But if you continue to study eventually, you will get afraid of design patterns too. It is like infant phobia, once you get afraid at your early age, it stays with you forever. So the result would be that you never look back at design patterns again. Let me see whether I can solve this brain teaser for you.

In the image below, you have both design pattern listed in. I am trying to compare the two one on one to identify the similarities. If you observe the figure carefully, you will see an easily understandable color pattern (same color is used to mark the classes that are of similar kind).
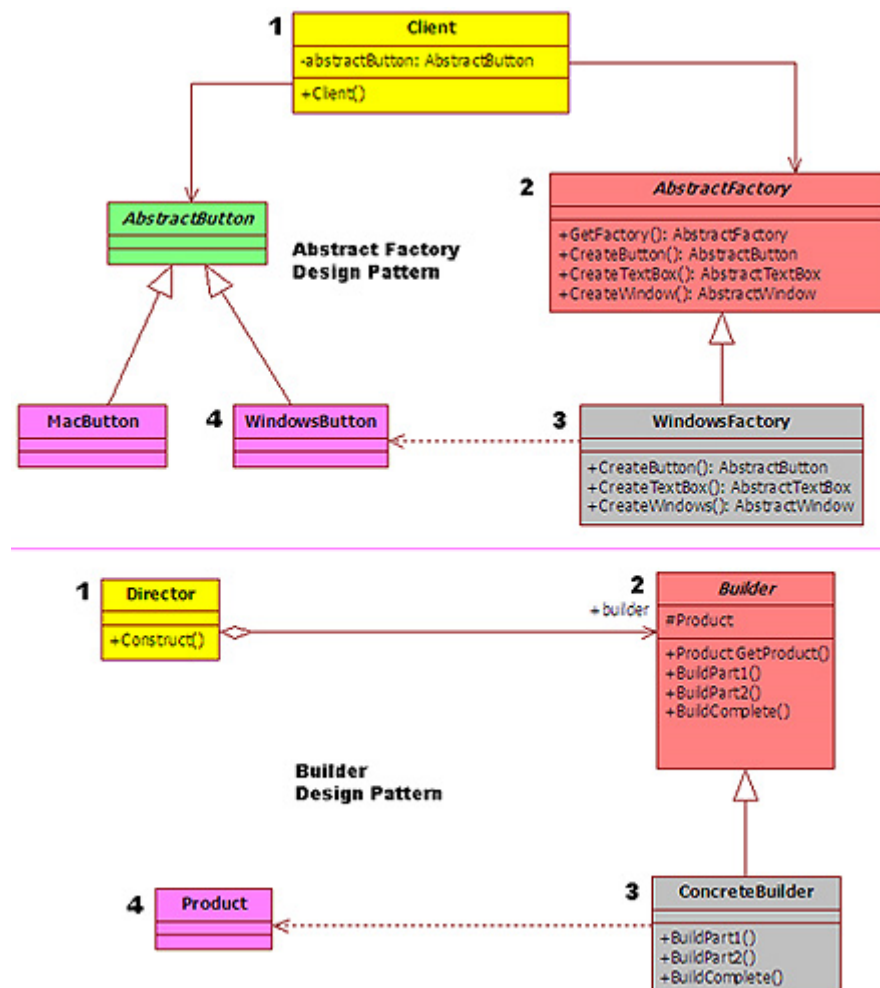
Please follow up with the numbers in the image when reading the listing below.

Mark **#1**: Both patterns have used a generic class as the entry-class. The only difference is the name of the class. One pattern has named it as "Client", while the other named it as "Director".
Mark **#2**: Here again the difference is the class name. It is "AbstractFactory" for one and "Builder" for the other. Additionally both classes are of type abstract.
Mark **#3**: Once again both patterns have defined two generic (WindowsFactory & ConcreteBuilder) classes. They both have created by inheriting their respective abstract class.
Mark **#4**: Finally, both seem to produce some kind of a generic output.

Now, where are we? Aren't they looking almost identical? So then why are we having two different patterns here?

Let's compare the two again side by side for one last time, but this time, focusing on the differences.

- *Abstract Factory*: Emphasizes a family of product objects (either simple or complex)
- *Builder*: Focuses on constructing a complex object step by step

- *Abstract Factory*: Focus on *what* is made
- *Builder*: Focus on *how* it is made

- *Abstract Factory*: Focus on defining many different types of *factories* to build many *products*, and it is not a one builder for just one product
- *Builder*: Focus on building a one complex but one single *product*

- *Abstract Factory*: Defers the choice of what concrete type of object to make until run time
- *Builder*: Hide the logic/ operation of how to compile that complex object

- *Abstract Factory*: *Every* method call creates and returns different objects
- *Builder*: Only the *last* method call returns the object, while other calls partially build the object

Sometimes creational patterns are complementary: So you can join one or many patterns when you design your system. As an example builder can use one of the other patterns to implement which components get built or in another case Abstract Factory, Builder, and Prototype can use Singleton in their implementations. So the conclusion would be that the two design patterns exist to resolve two type of business problems, so even though they look similar, they are not.

I hope that this shed some light to resolve the puzzle. If you still don't understand it, then this time it is not you, it has to be me and it is since that I don't know how to explain it.

5. What is the Conclusion?

I don't think, that it is realistic trying to make a programming language be everything to everybody. The language becomes bloated, hard to learn, and hard to read if everything plus the kitchen sink is thrown in. In another word every language has their limitations. As system architect and designer we should be able to fully and more importantly correctly (this also mean that you shouldn't use a ballistic missile to kill a fly or hire FBI to catch the fly) utilize the available tools and features to build usable, sustainable, maintainable and also very importantly expandable software systems, that fully utilize the feature of the language to bring a competitively advance system to their customers. In order to do it, the foundation of a system places a vital role. The design or the architecture of a software system is **the** foundation. It hold the system together, hence designing a system properly (this never mean an *over* desinging) is the key to the success. When you talk about designing a software system, the correct handling of OOP concept is very important. I have made the above article richer with idea but still kept it short so that one can learn/ remind all of important concept at a glance. Hope you all will enjoy reading it.

6. What I Referred?

- MSDN Library: http://msdn2.microsoft.com/en-us/library/default.aspx

- Practical Approach to Computer Systems Design and Architecture: http://www.codeproject.com /useritems/System_Design.asp

- Introduction: What is Object-Oriented Programming?: http://www.inf.ufsc.br/poo/smalltalk /ibm/tutorial/oop.html

- Basic Object-Oriented Concepts: http://www.toa.com/pub/oobasics/oobasics.htm

- Inheritance and Polymorphism—Specialization and Generalization: http://en.csharp-online.net /Inheritance_and_Polymorphism%E2%80%94Specialization_and_Generalization

- Abstraction and Generalization: http://cs.wwc.edu/~aabyan/PLBook/HTML/AbsGen.html

- Object Oriented Analysis and Design Team: http://atlas.kennesaw.edu/~dbraun/csis4650 /A&D/index.htm

- Client/Server Software Architectures--An Overview: http://www.sei.cmu.edu/str/descriptions /clientserver.html

- Service-oriented architecture (SOA) definition: http://www.service-architecture.com /web-services/articles/service-oriented_architecture_soa_definition.html

- http://www.dofactory.com/Patterns/Patterns.aspx
- http://www.objectmentor.com
- Builder Design Pattern [Wiki]

7. History

- 28/01/2008
    - - Added more content base on design principles.
    - - Added History section
- 04/02/2008
    - - Added more details to explain the Composition.
- 15/04/2008
    - - Added Comparison between Abstract Factory and Builder
    - - Updated some of the wording slightly

License

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

About the Author

Nirosh   In-depth coverage of Microsoft .Net Technologies.

- If it is good for both good and bad people then it is bad for a good society..

Contact Nirosh via c_nir*o*sh@hotmail.com (Remove * to use)
View Nirosh's profile on Linked-In

Occupation: Architect

Location: Sri Lanka

Member

Discussions and Feedback

78 messages have been posted for this article. Visit **http://www.codeproject.com /KB/architecture/OOP_Concepts_and_manymore.aspx** to post and view comments on this article, or click **here** to get a print view with messages.