http://www.codeproject.com/Articles/22769/Introduction-to-Object-Oriented-Programming-Concep

http://www.c-sharpcorner.com/UploadFile/tusharkantagarwal/objectorientedcsharp11162005070743AM/objectorientedcsharp.aspx

http://www.allinoneplace.info/Programming/CSharpOppsConceptsFAQ.aspx

---

-

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.

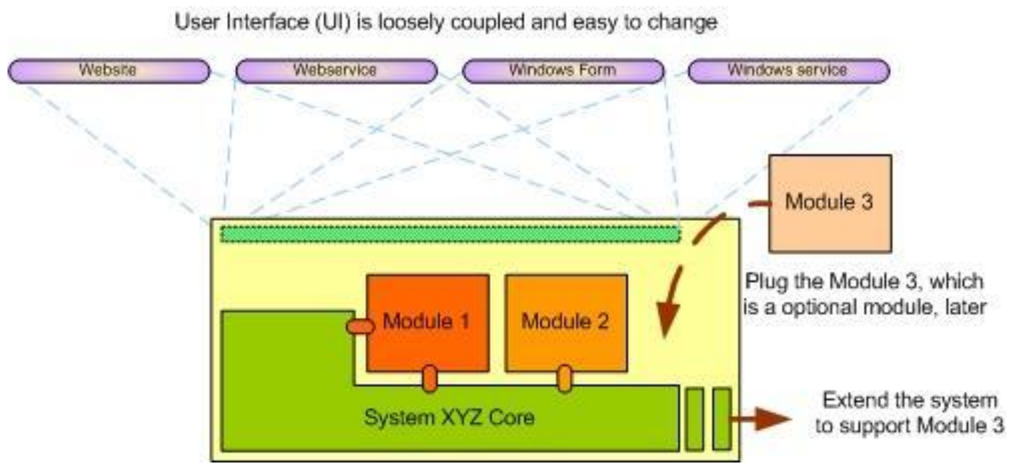
4. The Main Content

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?


User Interface (UI) is loosely coupled and easy to change
Website
Webservice
Windows Form
Windows service
Module 3
Plug the Module 3, which is a optional module, later
Module 1
Module 2
System XYZ Core
Extend the system to support Module 3

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.

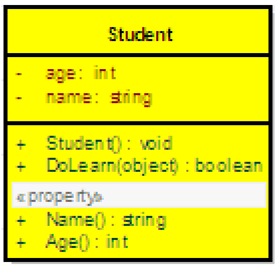
In order to clearly understand the object orientation, let's take your "*hand*" as an example. The "*hand*" is a class. Your body has two objects of type hand, named left hand and right hand. Their main functions are controlled/ managed by a set of electrical signals sent through your shoulders (through 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.

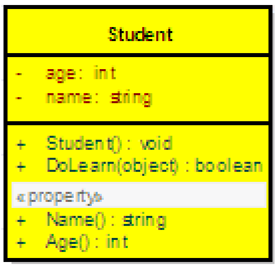
Additionally to identify a class correctly, you need to identify the full list of leaf level functions/ operations of the system (granular level use cases of the system). Then you can proceed to group each function to form classes (classes will group same types of functions/ operations). However a well 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();
```

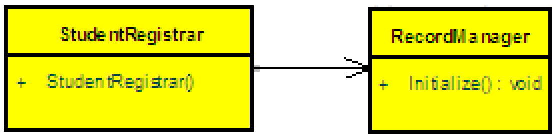
According to the sample above (let's assume that *LocalStudent* and *ForeignStudent* are implemented by the *IStudent* interface) we can see
how *LocalStudent* and *ForeignStudent*are hiding their, localize implementing information through the *IStudent* interface.

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*.

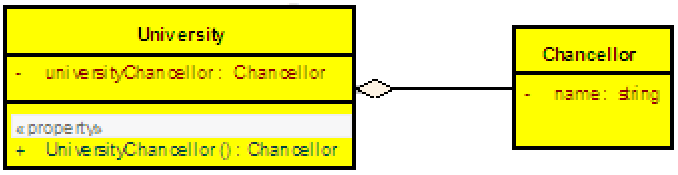


To some beginners, association is a confusing concept. The troubles created not only by the association alone, but with two other *OOP* concepts, that is association, aggregation and composition. Every one understands association, before aggregation and composition are described. The aggregation or composition 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?

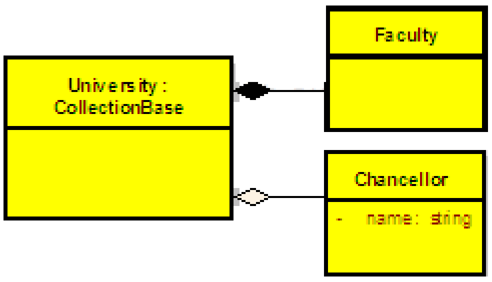
Association is a (***a***) relationship between two classes, where one class use another. But aggregation describes a special type of an association. Aggregation is the (***the***) relationship between two classes. When object of one class has an (***has***) object of another, if second is a part of 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();
}
```


University
- universityChancellor : Chancellor
«property»
+ UniversityChancellor() : Chancellor
Chancellor
- name : string

In this case I can say that *University* aggregate *Chancellor* or *University* has an (***has-**

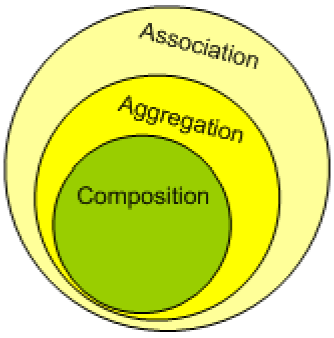
a*) *Chancellor*. But even without a *Chancellor* a *University* can exists. But the *Faculties*cannot exist without the *University*, the life time of a *Faculty* (or Faculties) attached with the life time of the *University* . If *University* is disposed the *Faculties* will not exist. In that case 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.

```
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