Object Oriented Programming

MD. MAHEDEE HASAN

MAHEDEE.NET

Object Oriented Programming

Md. Mahedee Hasan
Software Architect
Leadsoft Bangladesh Limited
Blog: ***http://mahedee.net/***
http:mahdee.blogspot.com
Email: ***mahedee.hasan@gmail.com***
Linkedin: ***http://www.linkedin.com/in/mahedee***
Facebook**:* ***http://www.facebook.com/mahedee19

Contents

Introduction

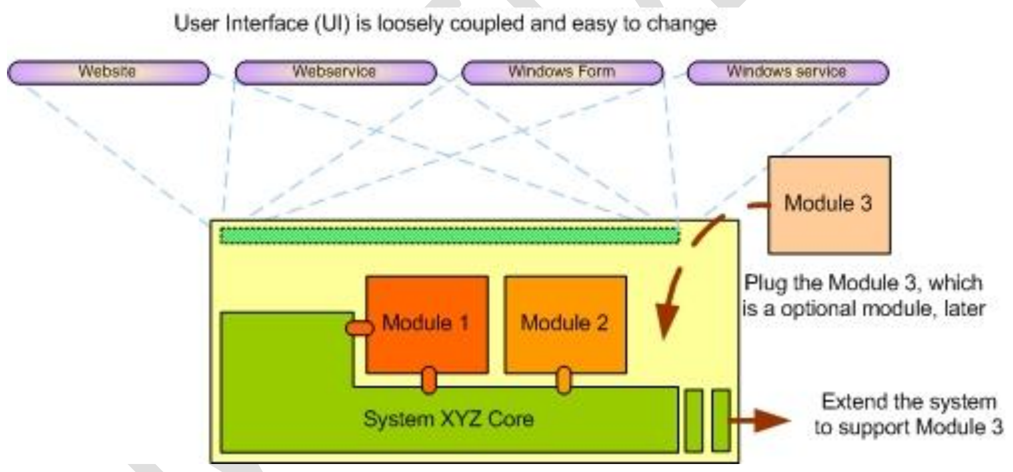
- Object-oriented programming (OOP) is a programming paradigm that uses "Objects and their interactions to design applications and computer programs".
- Based on the concepts of classes and objects that are used for modeling the real world entities
- Consist of a group of cooperating objects
- Objects exchange messages, for the purpose of achieving a common objective
- Implemented in object-oriented languages

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.

Importance of Software Architecture


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

Architecture is important because it:

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

OOP in a Nutshell

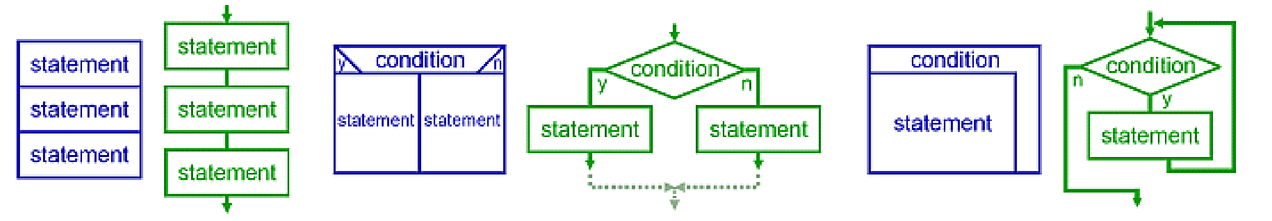
- OOP is a design philosophy
- A program models a world of interacting objects
- Gain simplicity, re-usability, extensibility, maintainibility by means of four main object-oriented programming concepts.
- Objects create other objects and “send messages” to each other (in C#, call each other’s methods)
- Each object belongs to a class
- A class defines properties of its objects
- The data type of an object is its class
- Programmers write classes (and reuse existing classes)
- Example : hands

Five basic characteristics of OOP Approach

- Everything is an object
- A program is a bunch of objects telling each other what to do by sending messages
- Each object has its own memory and might made up of other object
- Every object has a type
- All objects of a particular type can receive the same messages

Procedural Programming

- Procedural programming sometimes be used as a synonym for imperative programming
- Specifying the steps the program must take to reach the desired state
- Derived from structured programming
- Structured programming is a programming paradigm aimed on improving the clarity, quality, and development time of a computer program by making extensive use of
  - Subroutines, block structures and for and while loops—in contrast to using simple tests and jumps such as the goto statement
  - Could lead to "spaghetti code" which is both difficult to follow and to maintain.
- Based upon the concept of the procedure call
- Procedures, also known as routines, subroutines, methods, or functions
- Simply contain a series of computational steps to be carried out
- Any given procedure might be called at any point during a program's execution, including by other procedures or itself



Graphical representations of the three basic patterns. The box diagrams (blue) were invented right for the new theory, and you can see here their equivalents in the mostly used control flow charts.

Advantages of OOP

- Reusability
  - Classes and Objects
- Extensibility
  - Inheritance, aggregation and composition
- Simplicity
  - Abstraction, encapsulation and polymorphism
- Maintainability
  - All of the above three combined help us to maintain the code better.
- Better suited for team development
- Facilitates utilizing and creating reusable software components
- Makes business profitable by reusability;
  - Reduces cost of solution
  - Reduces time of development
  - Reduces effort
- Easier GUI programming
- Easier software maintenance
- All modern languages are object-oriented: Java, C#, PHP, Perl, C++ etc.

User Defined Data type

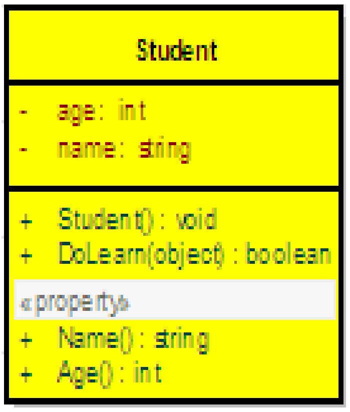
- This is the revolution of OOP
- It made almost direct translation of Problem Model into machine model
- Programmers can define problems in terms of Data Types what the problems demand
- Data type are provided with computing facilities through objects to deals with problem centric affairs-so problems are mapped here
- Platforms like Smaltalk, Java, C#.Net
    - use objects to provide solution, and
    - handle Lower level Machine Specific Mapping and Processing
- It comprises;
    - Class
    - Object

Classes and Object

What are objects?

- An object can be considered a "*thing*" that can perform a set of **related** activities.
- Software objects model real-world objects or abstract concepts
    - E.g. dog, bicycle, queue
- Real-world objects have states and behaviors
    - Dogs' states: name, color, breed, hungry
    - Dogs' behaviors: barking, fetching, sleeping
- 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.
- So, an object can have state, behavior and identity
- This means that – an object can have
    - internal data (state),
    - Methods, functionalities or services why object exists (behavior)
    - and each object can be uniquely distinguished from every other object of same or different types (Identity)
- How do software objects implement real-world objects?
    - Use variables/data to implement states
    - Use methods/functions to implement behaviors
- In pure OOP terms an object is an instance of a class.
- An object is a software bundle of variables and related methods

What are classes?



- 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.
- Object belongs to a class
- Provides user defined data types
- Provides a basis for uniform way to solve common problem
- Collection of objects having common attributes and functionalities.
- *Class* is composed of three things:
    - A name
    - Attributes
    - Operations

Example:

```
public class Student
{
}

Student objStudent = new Student();
```

How to identify and design a Class?

According to Object Oriented Design Principles, there are five principles that you must follow when design a class. These are called **SOLID**.

- **S**RP - The Single Responsibility Principle
  - A class should have one, and only one, reason to change.
- **O**CP - The Open Closed Principle
  - You should be able to extend a classes behavior, without modifying it.
- **L**SP - The Liskov Substitution Principle
  - Derived classes must be substitutable for their base classes.
- **I**SP - The Interface Segregation Principle
  - Make fine grained interfaces that are client specific.
- **D**IP - The Dependency Inversion Principle
  - Depend on abstractions, not on concretions.

There are more principles for object oriented design and dependency management other than SOLID. They are

- DRY – Don't Repeat Yourself
- REP — The Reuse Release Equivalency Principle
- CCP — The Common Closure Principle
- CRP — The Common Reuse Principle
- ADP — The Acyclic Dependencies Principle
- SDP — The Stable Dependencies Principle
- SAP — The Stable Abstractions Principle

To identify a class correctly

- 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)
- 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
- Follow divide and conquer approach
  - Identify the module of the system
  - Then dig deep into each module separately to seek out classes.

Four Main Object Oriented Programming Concept

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

- Abstraction
- Polymorphism
- Inheritance
- and Encapsulation

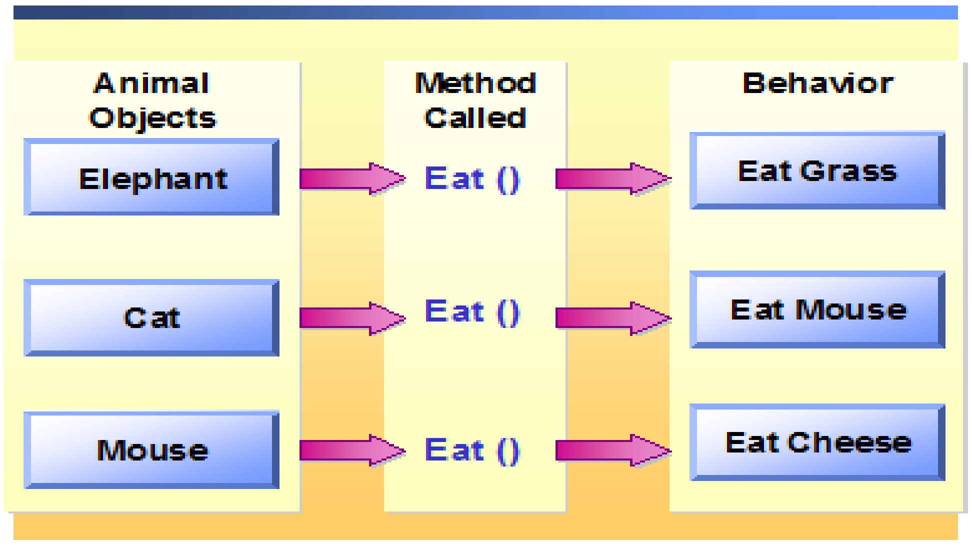
These concepts are the four main Object Oriented Programming Concepts often called **'APIE'**

Abstraction

- 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
- 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.

Polymorphism

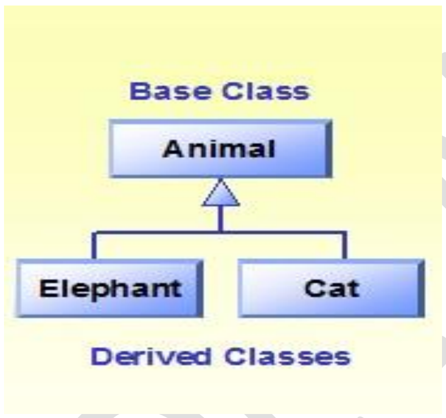
- Polymorphism is a generic term that means 'many shapes'.
- More precisely Polymorphism means the ability to request that the same operations be performed by a wide range of different types of things.
- In OOP the polymorphisms is achieved by using many different techniques named
  - Method overloading
  - Operator overloading
  - and Method overriding
- It enables you to treat your derived classes in a similar manner, even though they are different
- When you create derived classes, you provide more specialized functionality; polymorphism enables you to treat these new objects in a general way.



- It means
  - Same object acts differently to serve same service of different natures
  - Different objects of same type receive same message to do same job differently still with due accuracy

Inheritance

- Ability of a new class to be created, from an existing class by extending it, is called ***inheritance***
- Inheritance specifies an is-a (for specialization) kind of relationship
- Derived classes inherits properties and methods from base class, allowing code reuse
- Derived classes become more specialized.
- Abstraction is closely related with generalization, the inheritance is closely related with specialization.



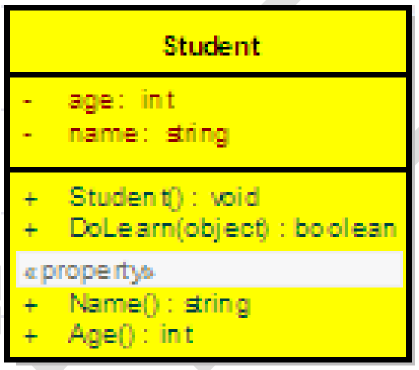
```
public class Animal
{
     public bool IsSleeping;
     public void Sleep()
     {
         Console.WriteLine("Sleeping");
     }
     public void Eat() { }
 }

 public class Antelope : Animal
 {
 }

 public class Lion : Animal
 {
     public void StalkPrey() { }
 }

 public class Elephant : Animal
 {
     public int CarryCapacity;
 }
```

```
Elephant e = new Elephant();

e.Sleep();
```

Encapsulation

- 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
- 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
- There are several other ways that an encapsulation can be used, as an example we can take the usage of an interface

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



Coupling

- The notion of coupling attempts to capture this concept of "how strongly" different modules are interconnected.
- Coupling among classes or subsystems is a measure of how interconnected those classes or subsystems are.
- Coupling increases with the complexity and obscurity of the interface between modules.
- **Tight coupling** means that related classes have to know
    - internal details of each other,
    - changes ripple through the system, and
    - the system is potentially harder to understand
    - It violates Encapsulation

- Ensure **Loose Coupling** - the goals behind achieving loose coupling between classes and modules are to:
- 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.

Cohesion

- Cohesion refers to how closely related methods and class level variables are in a class.
- Cohesion of a module represents how tightly bound the internal elements of the module are to one another
- 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/ classes to accomplish a variety of different tasks.
- Generally tight coupling gives low cohesion and loose coupling gives high cohesion.

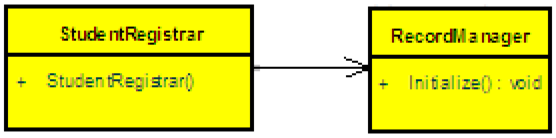
Association

- Association is a (***a***) relationship between two classes.
- It allows one object instance to cause another to perform an action on its behalf.

Example:

```
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*.

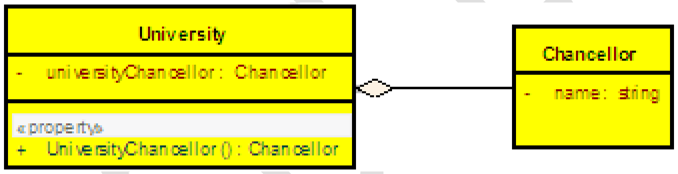


Composition

- Has-a relationship
- Decreases Degree of Coupling
- Object behaviors are easier to modify
- Enables an object dealing with multiple other objects for providing services
- Composition is preferred over Inheritance

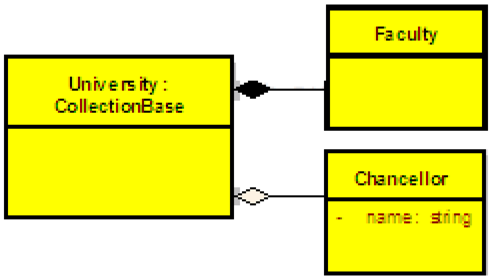
Difference between Association, Aggregation and Composition

- 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

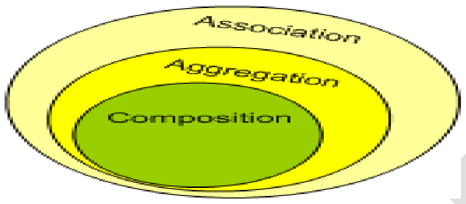

University
- universityChancellor : Chancellor
«property»
+ UniversityChancellor () : Chancellor
Chancellor
- name : string

```
public class University
{
    private Chancellor  universityChancellor = new 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*. Both depend on each other.
- .Net and Java uses the Composite relation to define their Collections
- 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)



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.
- It is used to manage complexity in large programs.
- In a sentences - 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.
- 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.

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.
- So, abstract class is generic base class
  - Contains an abstract method that must be implemented by a derived class.
- An abstract method has no implementation in the base class
- An *abstract class* is a class that can contain abstract members, although it is not required to do so.
- Any class that contains abstract members must be abstract. An abstract class can also contain non-abstract members.
- Example:

```
public abstract class Animal {
        public abstract void Eat();
        public abstract Group PolyGenicGroup{get;}
}
```

or

```
public abstract class Animal {
        public abstract void Eat();
}

public class Mouse : Animal {
        public override void Eat() {
                Console.WriteLine("Eat cheese");
        }
}
```

- When a derived class inherits an abstract method from an abstract class, it must override the abstract methods
- You can also create an abstract class that contains virtual methods, as shown in the following example:

```
public abstract class Animal {
        public virtual void Sleep() {
                Console.WriteLine("Sleeping");
        }
        public abstract void Eat();
}
```

- In this case, a derived class does not have to provide an implementation of the **Sleep** method because **Sleep** is defined as virtual.

Interface

- In summary the Interface separates the implementation and defines the structure
- Apart from that an interface is very useful when the implementation changes frequently
- Interface can be used to define a generic template and then one or more abstract classes to define partial implementations of the interface
- An interface like that of an abstract class cannot be instantiated.
- An interfaces can inherit other one or more interfaces
- Is a reference type that define contract
- Can contain method, properties, indexers, events
- Does not provide the implementations for the members
- Declare an interface

```
interface ICarnivore {
       bool   IsHungry { get; }
       Animal        Hunt();
       void   Eat(Animal victim);}
```

- Implement an interface

```
public class Lion: ICarnivore {
       private bool hungry;
       public bool IsHungry {
              get {
                     return hungry;
              }
       }

       public Animal Hunt() {
              // hunt and capture implementation
              // return animal object
       }

       public void Eat( Animal victim ) {
              // implementation
       }
}
```

- How to work with object that implement interface

```
ICarnivore objICarnivore = new Lion();

objICarnivore.Hunt();
```

What is the difference between a Class and an Interface?

- A *class* can be defined to implement an *interface*
- It also supports multiple implementations.
- When a *class* implements an *interface*, an *object* of such *class* can be encapsulated inside an *interface*.
- 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
- Example:

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

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

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

- 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 properties (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.
- Interface has no implementation; it only has the signature
- The main difference between them is that a class can implement more than one interface but can only inherit from one abstract class

Implicit and Explicit Interface 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.
- Example:
  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");
    }
}
```

Method Overloading

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

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
    }
}
```

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);
    }
}
```

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.

```
public class Animal {
    public virtual void Eat() {
        Console.WriteLine("Eat something");
    }
}

public class Cat : Animal {
    public override void Eat() {
        Console.WriteLine("Eat small animals");
    }
}
```

SOLID Object Oriented Principle

- It's an acronym of five principles introduced by **Mr. Robert Martin** (commonly known as Uncle Bob)
    - S-SRP -Single responsibility Principle
    - O-OCP - Open-closed Principle
    - L-LSP -Liskov substitution Principle
    - I-ISP - Interface segregation Principle
    - D-DIP - Dependency inversion Principle

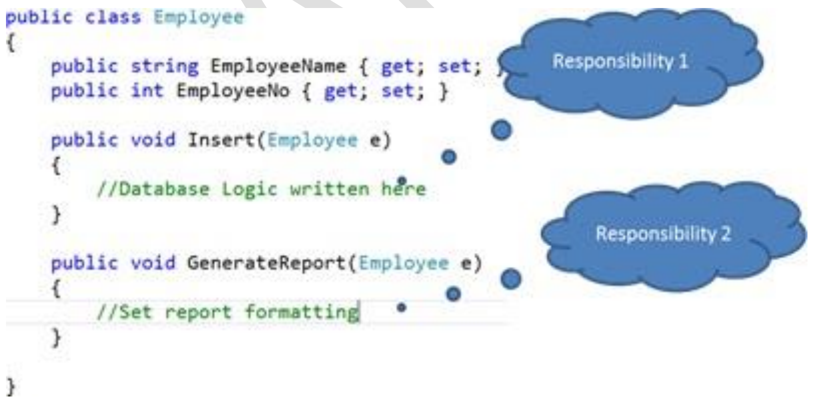
SRP -Single responsibility Principle

Real world comparison

S/W engineer
Author
News paper editor
CAN'T YOU DO ANYTHING RIGHT?
Copyright 2003 Randy Glasbergen. www.glasbergen.com

Identify Problem in Programming

Before we talk about this principle I want you take a look at following class.



```
public class Employee
{
    public string EmployeeName { get; set; }
    public int EmployeeNo { get; set; }

    public void Insert(Employee e)
    {
        //Database Logic written here
    }

    public void GenerateReport(Employee e)
    {
        //Set report formatting
    }

}
```

- Every time insert logic changes, this class will change.
- Every time report format changes, this class will changes.
- So a single change leads to double testing (or maybe more).

What is SRP?

SRP says "Every software module should have only one reason to change".

- Software Module – Class, Function etc.
- Reason to change - Responsibility

Solutions which will not Violate SRP

Now it's up to us how we achieve this. One thing we can do is create three different classes

1. `Employee` – Contains Properties (Data)
2. `EmployeeDB` – Does database operations
3. `EmplyeeReport` – Does report related tasks

```
public class Employee
{
    public string EmployeeName { get; set; }
    public int EmployeeNo { get; set; }
}
public class EmployeeDB
{
    public void Insert(Employee e)
    {
        //Database Logic written here
    }
 public Employee Select()
    {
        //Database Logic written here
    }
}
public class EmployeeReport
{
    public void GenerateReport(Employee e)
    {
        //Set report formatting
    }
}
```

Can a single class can have multiple methods?

The answer is YES. Now you might ask how it's possible that

1. A class will have single responsibility.
2. A method will have single responsibility.
3. A class may have more than one method.

```
//Method with multiple responsibilities – violating SRP
public void Insert(Employee e)
{
    string StrConnectionString = "";
    SqlConnection objCon = new SqlConnection(StrConnectionString);
    SqlParameter[] SomeParameters=null;//Create Parameter array from values
    SqlCommand objCommand = new SqlCommand("InertQuery", objCon);
    objCommand.Parameters.AddRange(SomeParameters);
    ObjCommand.ExecuteNonQuery();
}

//Method with single responsibility – follow SRP
public void Insert(Employee e)
{
    SqlConnection objCon = GetConnection();
    SqlParameter[] SomeParameters=GetParameters();
    SqlCommand ObjCommand = GetCommand(objCon,"InertQuery",SomeParameters);
    ObjCommand.ExecuteNonQuery();
}

private SqlCommand GetCommand(SqlConnection objCon, string InsertQuery, SqlParameter[]
SomeParameters)
{
    SqlCommand objCommand = new SqlCommand(InsertQuery, objCon);
    objCommand.Parameters.AddRange(SomeParameters);
    return objCommand;
}

private SqlParameter[] GetParaeters()
{
    //Create Paramter array from values
}

private SqlConnection GetConnection()
{
    string StrConnectionString = "";
    return new SqlConnection(StrConnectionString);
}
```

OCP – Open Close Principle

Real World Comparison

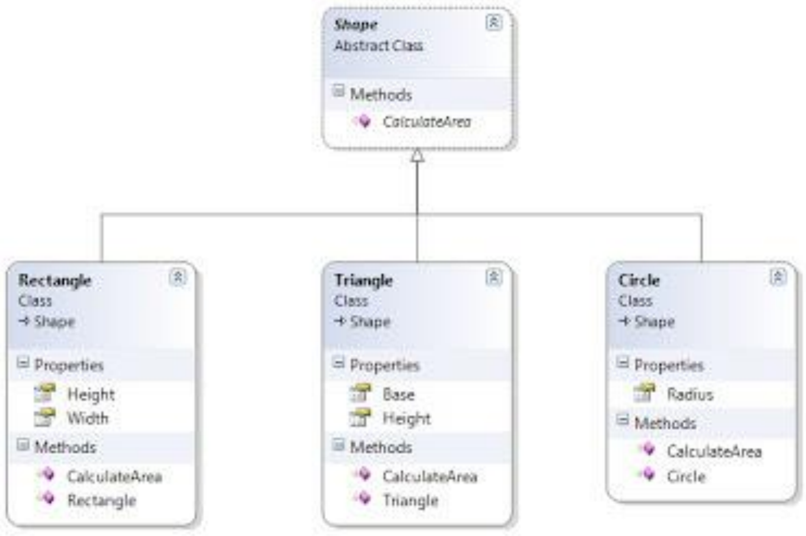
Let’s assume you want to add one more floor between the first and second floor in your two floor house. Do you think it is possible? Yes it is, but is it feasible? Here are some options:

- One thing you could have done at time you were building the house first time was make it with three floors, keeping second floor empty. Then utilize the second floor anytime you want. I don’t know how feasible that is, but it is one solution.
- Break the current second floor and build two new floors, which is not sensible.

What is OCP?

- First introduced by Betrand Meyer in 1988
- He says “Software entities (Class, module, function etc.) should be open for extension, but closed for modification”.
- An entity is “Open for extension” means that its behavior can be extended to accommodate new demand.
- The entity is “closed for modification” means that the existing source code of the module is not changed or minimum change when making enhancement
- Reduces risk – Because sometimes code changes introduce heavy risk

Example:



```
public abstract class Shape
{
 public abstract double CalculateArea();
}


//Rectangle class
public class Rectangle : Shape
{
 public double Height { get; set; }
```

```
        public double Width { get; set; }

        public Rectangle(double height, double width)
        {
            this.Height = height;
            this.Width = width;
        }

        public override double CalculateArea()
        {
            return Height * Width;
        }
       }


       //Triangle class
       public class Triangle : Shape
        {
        public double Base { get; set; }
        public double Height { get; set; }

        public Triangle(double vbase, double vheight)
        {
            this.Base = vbase;
            this.Height = vheight;
        }

        public override double CalculateArea()
        {
            return 1 / 2.0 * Base * Height;
        }
       }


       //If you wish to add Circle You don't need to modify existing class
       public class Circle : Shape
       {
        public double Radius { get; set; }


        public Circle(double radius)
        {
            this.Radius = radius;
        }

        public override double CalculateArea()
        {
            return Math.PI * Radius * Radius;
        }
    }



//Client class which uses Rectangle, Triangle and Circle class
  public class Program
    {
        public static void Main(string[] args)
```

```
 {
     Shape objShape = new Rectangle(20, 30);
     Console.WriteLine("Area of Rectangle: " + objShape.CalculateArea());

     objShape = new Triangle(20, 30);
     Console.WriteLine("Area of Triangle: " + objShape.CalculateArea());

     objShape = new Circle(4);
     Console.WriteLine("Area of Circle: " + objShape.CalculateArea());


     Console.ReadKey();
 }
}
```

LSP - Liskov substitution principle

What is LSP?

- You might be wondering why we are defining it prior to examples and problem discussions. Simply put, I thought it will be more sensible here.
- It says, "Subclasses should be substitutable for base classes." Don't you think this statement is strange? If we can always write `BaseClass b = new DerivedClass()` then why would such a principle be made?

Real World Comparison

- A father is a real estate business man whereas his son wants to be cricketer.
- A son can't replace his father in spite of the fact that they belong to same family hierarchy.

Identify Problem in Programming

- Let's talk about a very common example. Normally when we talk about geometric shapes, we call a rectangle a base class for square. Let's take a look at code snippet.

```
public class Rectangle
{
    public int Width { get; set; }
    public int Height { get; set; }
}

public class Square:Rectangle
{
    //codes specific to
    //square will be added
}
```

- One can say,

```
Rectangle o = new Rectangle();
o.Width = 5;
o.Height = 6;
```

- Perfect but as per LSP we should be able to replace Rectangle with square. Let's try to do so.

```
Rectangle o = new Square();
o.Width = 5;
o.Height = 6;
```

- **What is the matter?** Square cannot have different width and height.
- **What it means?** It means we can't replace base with derived. Means we are violating LSP.
- **Why don't we make width and height virtual in Rectangle, and override them in Square?**

```
public class Square : Rectangle
{
    public override int Width
    {
        get{return base.Width;}
        set
        {
            base.Height = value;
            base.Width = value;
        }
    }
    public override int Height
    {
        get{return base.Height;}
        set
        {
            base.Height = value;
            base.Width = value;
        }
    }
}
```

- We can't because doing so we are violating LSP, as we are changing the behavior of Width and Height properties in derived class (for Rectangle height and width cannot be equal, if they are equal it's cannot be Rectangle). (It will not be a kind of replacement).

Solution which will not violate LSP

- There should be an abstract class Shape which looks like:

```
public abstract class Shape
{
    public virtual int Width { get; set; }
    public virtual int Height { get; set; }
}
```

- Now there will be two concrete classes independent of each other, one rectangle and one square, both of which will be derived from Shape. Now the developer can say:

```
Shape o = new Rectangle();
o.Width = 5;
o.Height = 6;

Shape o = new Square();
o.Width = 5; //both height and width become 5

o.Height = 6; //both height and width become 6
```

- 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.

ISP– Interface Segregation principle

Real World Comparison

- What if every interface of the motherboard is open
    - You may confuse which port is for which purpose. Sometimes you will take wrong decision.

Identify Problem in Programming

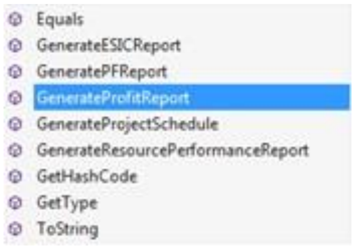
Let’s say we want to develop a Report Management System. Now, the very first task is creating a business layer which will be used by three different UIs.

1. `EmployeeUI` – Show reports related to currently logged in employee
2. `ManagerUI` – Show reports related to himself and the team for which he/manager belongs.
3. `AdminUI` – Show reports related to individual employee ,related to team and related to company like profit report.

```
public interface IReportBAL
{
    void GeneratePFReport();
    void GenerateESICReport();

    void GenerateResourcePerformanceReport();
    void GenerateProjectSchedule();

    void GenerateProfitReport();
}
public class ReportBAL : IReportBAL
{
    public void GeneratePFReport()
    {/*...............*/}

    public void GenerateESICReport()
    {/*...............*/}

    public void GenerateResourcePerformanceReport()
    {/*...............*/}

    public void GenerateProjectSchedule()
    {/*...............*/}

    public void GenerateProfitReport()
    {/*...............*/}
}
public class EmployeeUI
{
    public void DisplayUI()
    {
        IReportBAL objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
    }
}
public class ManagerUI
{
    public void DisplayUI()
    {
        IReportBAL objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
        objBal.GenerateResourcePerformanceReport ();
```

```
            objBal.GenerateProjectSchedule ();
        }
    }
    public class AdminUI
    {
        public void DisplayUI()
        {
            IReportBAL objBal = new ReportBAL();
            objBal.GenerateESICReport();
            objBal.GeneratePFReport();
            objBal.GenerateResourcePerformanceReport();
            objBal.GenerateProjectSchedule();
            objBal.GenerateProfitReport();
        }
    }
```

- Now in each UI when the developer types "`objBal`" the following intellisense will be shown:



```
Equals
GenerateESICReport
GeneratePFReport
GenerateProfitReport
GenerateProjectSchedule
GenerateResourcePerformanceReport
GetHashCode
GetType
ToString
```

What is the problem?

- The developer who is working on `EmployeeUI` gets access to all the other methods as well, which may unnecessarily cause him/her confusion.

What is ISP?

- It states that "**Clients should not be forced to implement interfaces they don’t use**."
- It can also be stated as "Many client specific interfaces are better than one general purpose interface."
- In simple words, if your interface is fat, break it into multiple interfaces.

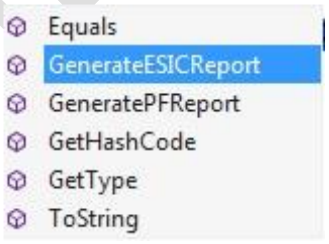
Update code to follow ISP

```
public interface IEmployeeReportBAL
{
    void GeneratePFReport();
    void GenerateESICReport();
}
public interface IManagerReportBAL : IEmployeeReportBAL
{
    void GenerateResourcePerformanceReport();
    void GenerateProjectSchedule();
}
public interface IAdminReportBAL : IManagerReportBAL
```

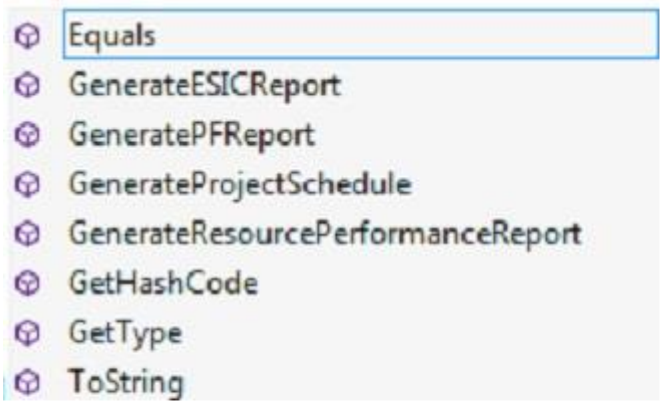
```
{
    void GenerateProfitReport();
}
public class ReportBAL  : IAdminReportBAL
{
    public void GeneratePFReport()
    {/*...............*/}

    public void GenerateESICReport()
    {/*...............*/}

    public void GenerateResourcePerformanceReport()
    {/*...............*/}

    public void GenerateProjectSchedule()
    {/*...............*/}

    public void GenerateProfitReport()
    {/*...............*/}
}
```

- Intelligence show on the following code

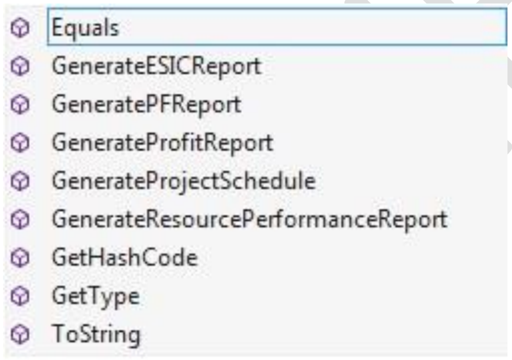
```
public class EmployeeUI
{
    public void DisplayUI()
    {
        IEmployeeReportBAL objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
    }
}
```



```
public class ManagerUI
{
    public void DisplayUI()
    {
        IManagerReportBAL  objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
        objBal.GenerateResourcePerformanceReport ();
        objBal.GenerateProjectSchedule ();
    }
}
```


Equals
GenerateESICReport
GeneratePFReport
GenerateProjectSchedule
GenerateResourcePerformanceReport
GetHashCode
GetType
ToString

```
public class AdminUI
{
    public void DisplayUI()
    {
        IAdminReportBAL  objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
        objBal.GenerateResourcePerformanceReport();
        objBal.GenerateProjectSchedule();
        objBal.GenerateProfitReport();
    }
}
```


Equals
GenerateESICReport
GeneratePFReport
GenerateProfitReport
GenerateProjectSchedule
GenerateResourcePerformanceReport
GetHashCode
GetType
ToString

DIP– Dependency Inversion principle

Real World Comparison

- Let’s talk about our desktop computers. Different parts such as RAM, a hard disk, and CD-ROM (etc.) are loosely connected to the motherboard. That means that, if, in future in any part stops working it can easily be replaced with a new one. Just imagine a situation where all parts were tightly coupled to each other, which means it would not be possible to remove any part from the motherboard. Then in that case if the RAM stops working we have to buy new motherboard which is going to be very expensive.

Identify Problem in Programming

- Look at the following code.

```
public class CustomerBAL
{
    public void Insert(Customer c)
    {
        try
        {
            //Insert logic
        }
        catch (Exception e)
        {
            FileLogger f = new FileLogger();
            f.LogError(e);
        }
    }
}

public class FileLogger
{
    public void LogError(Exception e)
    {
        //Log Error in a physical file
    }
}
```

- In the above code `CustomerBAL` is directly dependent on the `FileLogger` class which will log exceptions in physical file. Now let's assume tomorrow management decides to log exceptions in the Event Viewer. Now what? Change existing code. Oh no! My God, that might create a new error!

What is DIP?

- It says, "**High level modules should not depend upon low level modules. Rather, both should depend upon abstractions**."

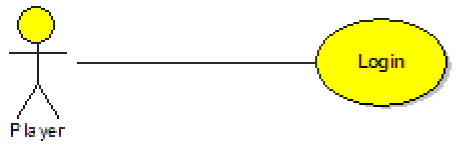
Solution with DIP

```
public interface ILogger
{
    void LogError(Exception e);
}

public class FileLogger:ILogger
```

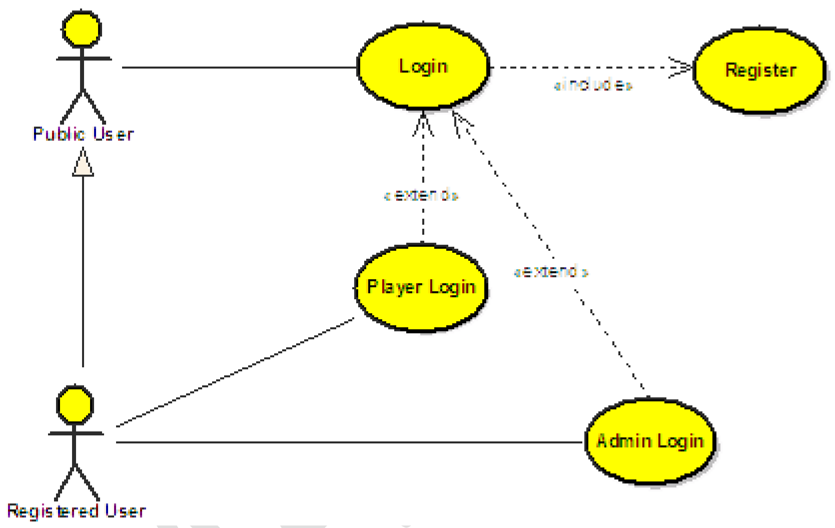
```
{
    public void LogError(Exception e)
    {
        //Log Error in a physical file
    }
}
public class EventViewerLogger : ILogger
{
    public void LogError(Exception e)
    {
        //Log Error in a physical file
    }
}
public class CustomerBAL
{
    private ILogger _objLogger;
    public CustomerBAL(ILogger objLogger)
    {
        _objLogger = objLogger;
    }

    public void Insert(Customer c)
    {
        try
        {
            //Insert logic
        }
        catch (Exception e)
        {
            _objLogger.LogError(e);
        }
    }
}
```

Basic Idea about 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 communicates with another system.

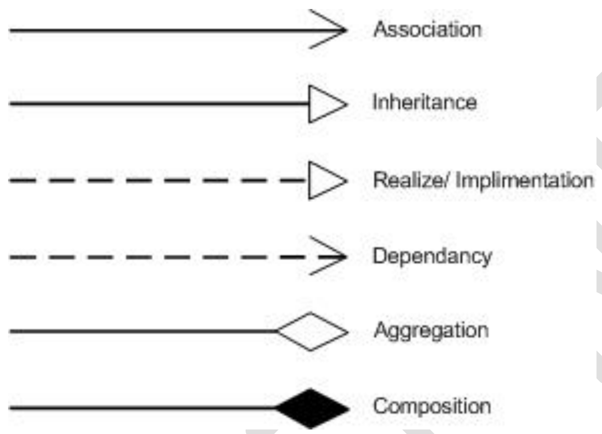


- In another angle a use case encodes a typical user interaction with the system. In particular, it:
    - Captures some user-visible function.
    - Achieves some concrete goal for the user.
- A complete set of use cases largely defines the requirements for your system: everything the user can see, and would like to do.



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



Different Diagram

Package Diagram

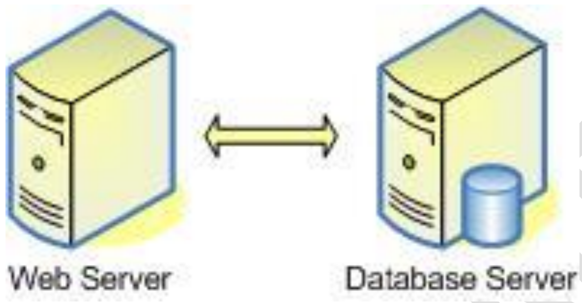
- Package diagrams are used to reflect the organization of packages and their elements

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.

What is two-tier architecture?

- The two-tier architecture is refers to client/ server architectures as well
- 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.

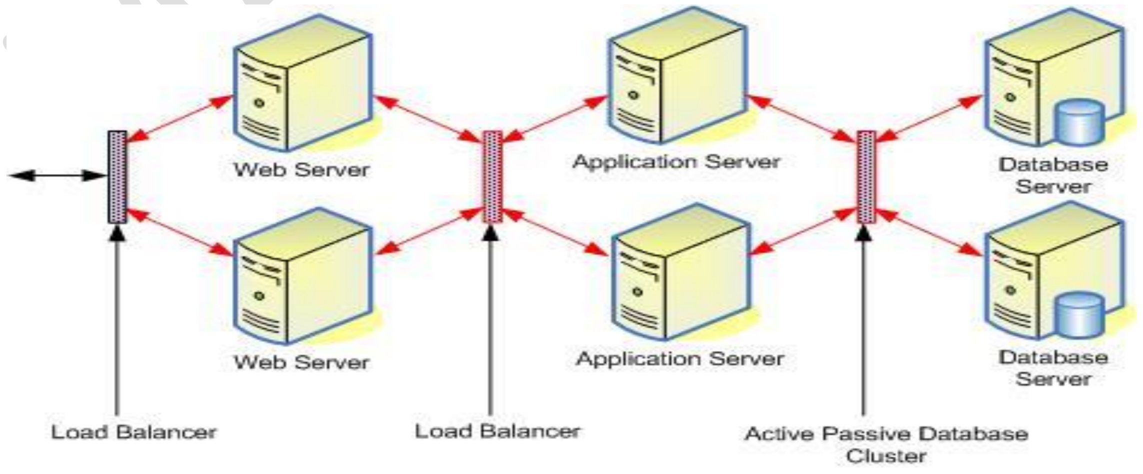


What is three-tier architecture?

- The three tier software architecture (also known as three layer architectures) emerged in the 1990s
    - Overcome the limitations of the two tier architecture.

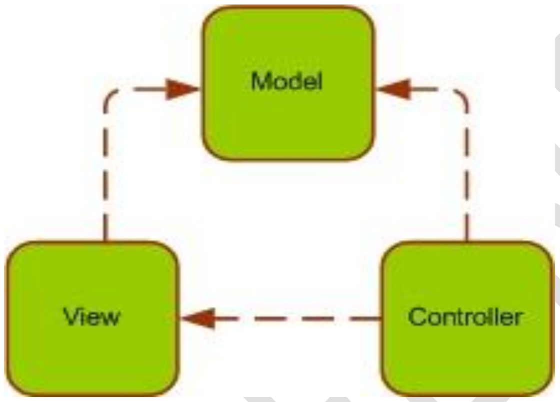
The 3-Tier architecture has the following three tiers.

- **Presentation Tier or Web Server**: User Interface, displaying/ accepting data/ input to/ from the user
- **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
- **Data Tier or Database server**: Simple reading and writing method to database or any other storage, connection, command, stored procedures etc



What is MVC architecture?

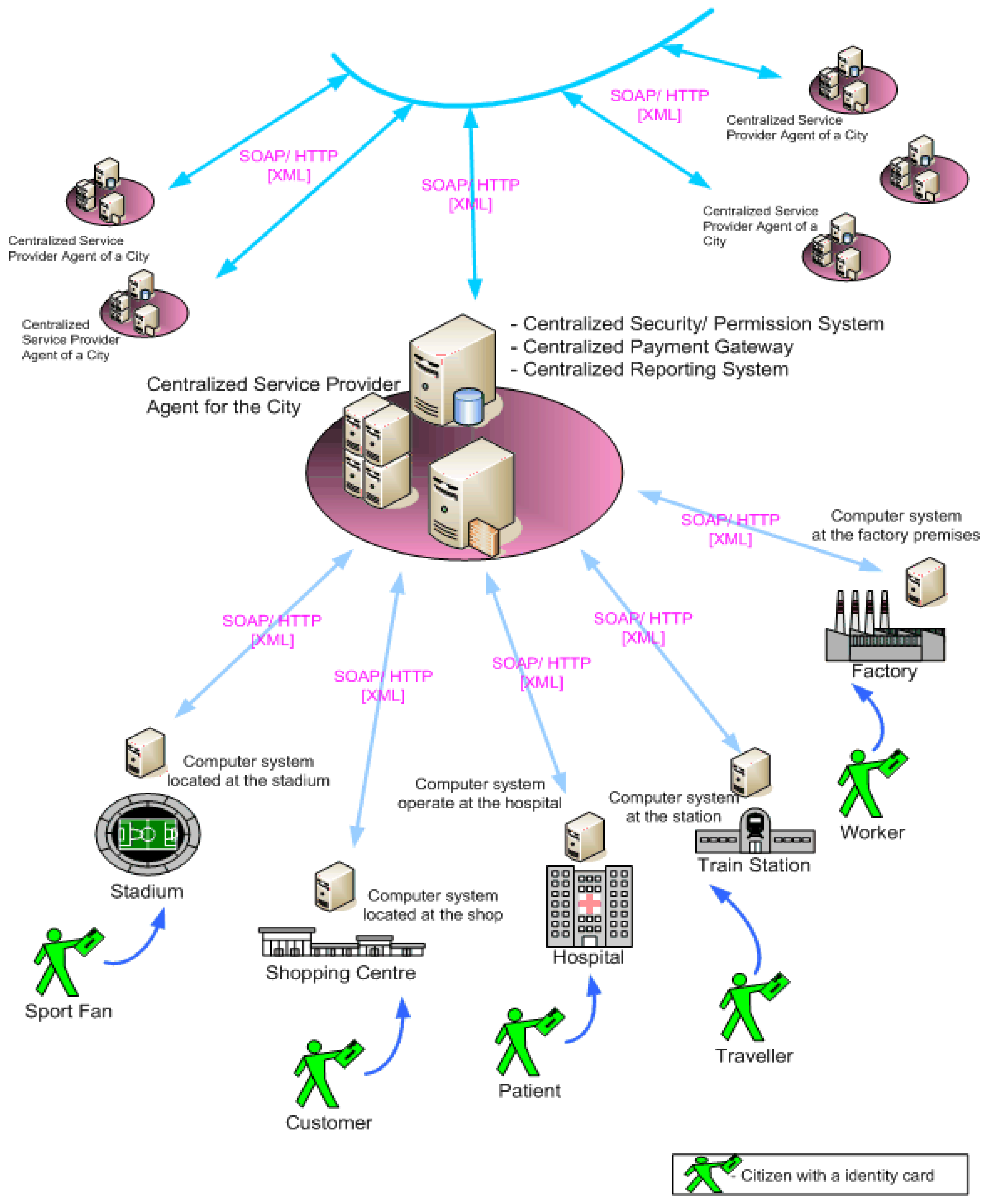
- The **M**odel-**V**iew-**C**ontroller (*MVC*) architecture separates the modeling of the domain
- The definitions given below are the closes possible ones I found for ASP.NET version of *MVC*.
    - **Model**: *DataSet* and typed *DataSet* (sometimes business object, object collection, XML etc) are the most common use of the model.
    - **View**: The *ASPX* and *ASCX* files generally handle the responsibilities of the view.
    - **Controllers**: The handling of events or the controlling is usually done in the code-behind class.



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.

The Service Oriented Architecture (SOA) is being used to provide centralized services


SOAP/ HTTP [XML]
SOAP/ HTTP [XML]
SOAP/ HTTP [XML]
Centralized Service Provider Agent of a City
Centralized Service Provider Agent of a City
Centralized Service Provider Agent of a City
Centralized Service Provider Agent of a City
- Centralized Security/ Permission System
- Centralized Payment Gateway
- Centralized Reporting System
Centralized Service Provider Agent for the City
SOAP/ HTTP [XML]
Computer system at the factory premises
SOAP/ HTTP [XML]
SOAP/ HTTP [XML]
SOAP/ HTTP [XML]
SOAP/ HTTP [XML]
Factory
Computer system located at the stadium
Computer system operate at the hospital
Computer system at the station
Worker
Stadium
Train Station
Computer system located at the shop
Hospital
Shopping Centre
Sport Fan
Traveller
Patient
Customer
- Citizen with a identity card

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.

What is the Business Logic Layer?

- 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.

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.

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

History Card

Version No	Modification History	Update Date	Published Date
1	Created	5 December 2013	22 December 2013
2	Update Cover	24 November 2014	24 November 2014

References:

1. http://www.oodesign.com/liskov-s-substitution-principle.html (link)
2. http://mahedee.blogspot.com/2012/12/open-close-principle.html (link)
3. http://www.objectmentor.com/omSolutions/oops_what.html (Links)
4. http://en.wikibooks.org/wiki/Object_Oriented_Programming (Link)
5. http://oopsconcepts.blogspot.com/ (Link)
6. http://www.codeproject.com/Articles/22769/Introduction-to-Object-Oriented-Programming-Concep (Link)
7. Basic Object-Oriented Concepts - Edward V. Berard (The Object Agency, Inc.) (Slide)
8. Qualities of a class (Slide)
9. http://www.virtuosimedia.com/dev/php/procedural-vs-object-oriented-programming-oop (Link)
10. http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Procedural_programming.html (Link)
11. http://en.wikipedia.org/wiki/Structured_programming (Link)
12. A Concise Introduction to Software Engineering - Pankaj Jalote (Book)
13. MSDN C# Training Materials (Tutorial)
14. http://www.codeproject.com/Articles/567768/Object-Oriented-Design-Principles (Link)
15. http://www.codeproject.com/Articles/495019/Dependency-Inversion-Principle-and-the-Dependency(Link)