AUTODESK **UNIVERSITY** 2015

SD10669

Applying Design Patterns to AutoCAD® .NET application development

James Johnson,
Synergis Technologies, LLC, Sr. Application Developer

Learning Objectives

- Understanding where and why to use design patterns
- Applying design pattern code and refactoring existing code for design patterns
- How design patterns can help you build better applications
- Have sample code using design patterns that can be immediately used in your applications.

Description

Design patterns provide reusable solutions to common software issues that occur frequently when doing software design. This instructional demo is an introduction for .NET software developers on how to use design patterns in their .NET applications. There are 23 common design patterns known as the Gang of four (GOF) and are often considered to be the base for other design patterns. This instructional demo will discuss using and selecting the proper patterns in your applications. This instructional demo will demonstrate how to use common patterns with AutoCAD® plugin code samples.

Speaker

James Johnson *has worked with CAD products for more than 25 years in many positions, from being a CAD drafter to writing automation applications. In his current position he is doing CAD integration for adept document management system. In previous positions he used RealDWG® to write custom automation to create AutoCAD software drawings of industrial kitchen equipment, and he worked at Autodesk, Inc., resellers in software development groups doing custom applications for Inventor software and AutoCAD software. He has taught AutoCAD software and AutoCAD Microsoft Visual Basic software classes while working for resellers, and he was a CAD instructor at 2 different community colleges.*

Introduction

Design patterns referenced in this class and supporting code samples are defined in the book "*Design Patterns: Elements of Reusable Object- Oriented Software* (Addison-Wesley)" by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides outlines and describes 23 commonly used design patterns. The authors of that book are commonly referred to as the "Gang of Four (GOF)".

A basic description of "Design Patterns" is that they are reusable software solutions to recurring problems in application development. The (GOF) patterns are considered the foundation for other patterns and are categorized into three types:

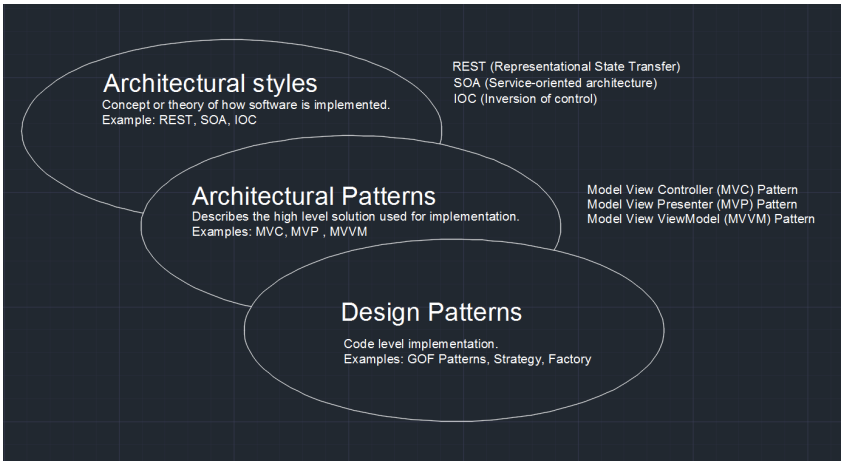
Creational Patterns: Provide ways to instantiate single objects or groups of related objects. Defined patterns are Abstract Factory, Builder, Factory Method, Prototype, and Singleton.

Structural Patterns: Provide ways to define relationships between classes or objects. Defined patterns are Adapter, Bridge, Composite, Decorator, Facade, Flyweight, and Proxy.

Behavioral Patterns: Provide communication between classes and objects. Defined patterns are Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, and Visitor.

There are many other books, websites and wiki pages that have furthered the definition of these patterns and additional variations. Another book that is highly recommended is "Head First Design Patterns" published by O'Reilly and authored by Eric Freeman and Elizabeth Freeman. It uses Java as its base development language and takes a lighter approach to learning design patterns.

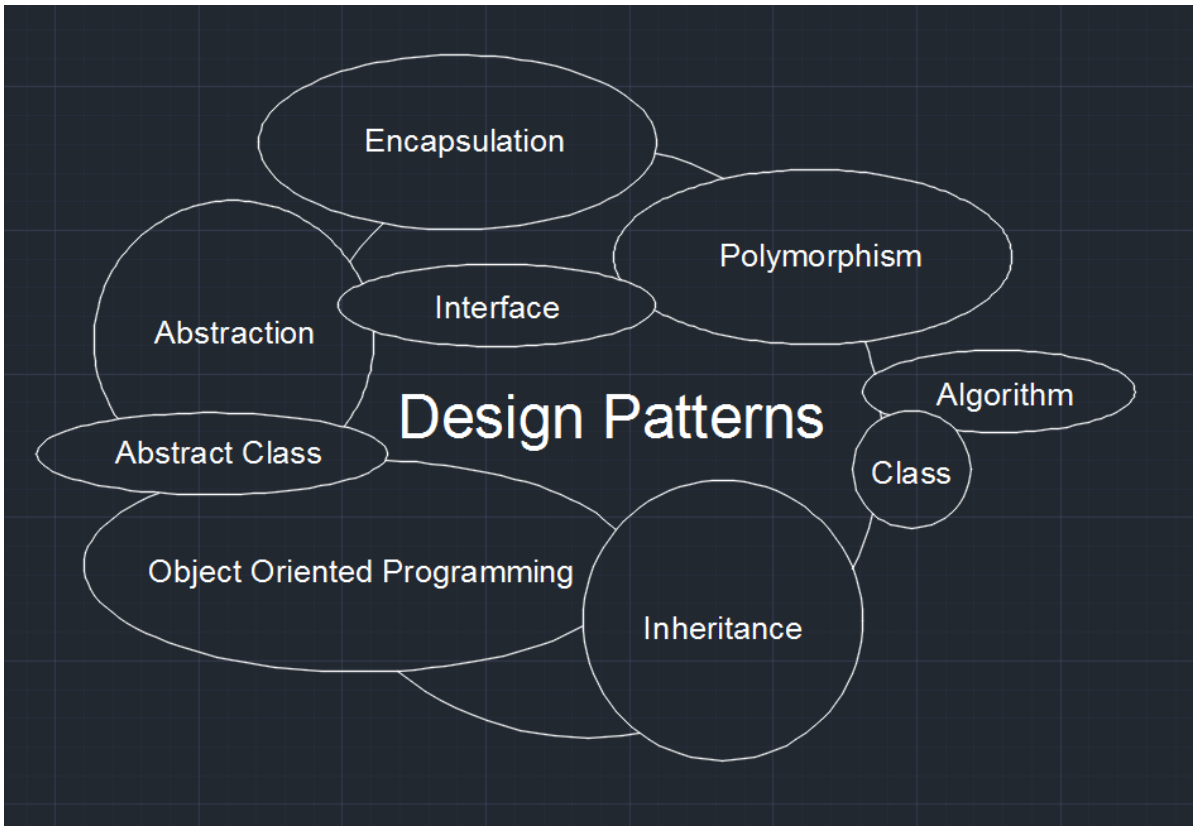
When looking into design patterns they could be considered as "CODE" level implementation patterns where Architectural patterns are at a solution level implementation. The way that I have began to look at it is in this layered level where Architectural styles (Principles) are at theory level, Architectural patterns are at overall solution level and design patterns are at the code level of implementation.



Getting Started

Using “Design Patterns” does not require any additional tools than used in day-to-day application development or any extraordinary programming capabilities.

Using “Design Patterns” will require knowledge of object-oriented programming and an understanding to definitions of terms like data types, polymorphism, interface and inheritance.



The general description of what “Design Patterns” are often refers to them as reusable software solutions to recurring problems that have been time tested. One of the main advantages of learning “Design Patterns” is that a developer needs to learn or develop a better understanding of Object oriented programming techniques.

Getting Started continued…

Data types: A programming classification identifying various types of data. Types are defined for every variable, constant and expression that evaluates to a value. Programs use built-in types from the .NET framework or referenced class libraries as well as user defined types.

Samples:

```
int iCount = 0;
bool bCheck = false;
string pathName = string.Empty;
string pathFullName = string.Empty;
RXObject objRXEnt = dbObj;


doSomething doIt = new doSomething();

public class doSomething
{
}
```

Data types are standard system types (int, bool, etc.), referenced API types(RXObject) and defined classes (dosomething).

Polymorphism: A programming concept that provides the ability of objects to take on multiple types. Polymorphism can be static where the response to a function is determined at compile time or it can be dynamic where it is determined at runtime.

Static Overload Sample:

```
public class Overloaded
{
        public void Add(string val1, string val2)
        {
            //do something
        }

        public void Add(int val1, int val2)
        {
            //do Something
        }
}
public class mainOverload
{
        public void doOver()
        {
            Overloaded obj = new Overloaded();
            obj.Add("One", "Two");
            obj.Add(1, 2);
        }
}
```

The ADD method is overloaded, with different arguments.

Dynamic Sample (accomplished by method overriding):

```
public class Base
{
        public virtual void Show()
        {
            //do Something
        }
}
```

The Show method is overridden from the inherited Base class.

```
public class Derived : Base
{
        public override void Show()
        {
            //do something different than Base
        }
}
public class main
{
        public void useDerived()
        {
            Base obj = new Base();
            obj.Show();

            obj = new Derived();
            obj.Show();
        }
}
```

The first Show method would use the Base class and second show is from Derived class.

Inheritance: An object or class can be based on another object or class. Inheritance allows creating new classes that reuse, extend, and modify the behavior that is defined in other classes. A class that is inherited is called the base class, and the class that inherits is called the derived class.

```
public class Shape
{
        public double width {set; get;}
        public double height {set; get;}
        public double thick {set; get;}
}
public interface WeightCalc
{
        double getWeight(double volume);
}
public class Rectangle : Shape, WeightCalc
{
        public double getVolume()
        {
                return (width * height * thick);
        }
        public double getWeight(double volume)
        {
                return volume * 0.283;
        }
}
```

The Shape class and WeightCalc interface are inherited by Rectangle class.

Getting Started continued…

Interface: An interface contains the signatures of methods, properties, events or indexers. Classes that implement an interface must implement the members of the interface that are specified in the interface definition.

```
public interface WeightCalc
{
        double getWeight(double volume);
}
```

WeightCalc is an interface with a single method getWeight.

Encapsulation: The process of binding data members (variables, properties) and functions (methods) into a single unit.

```
public class Size
{
        private double _Length = 0;
        public double Length
        {
            set
            {
               _Length = value;
            }
            get
            {
               return _Length;
            }
        }
}
```

_Length variable is encapsulated in the Size class and its value is returned and set with the Length property.

Another advantage of object-oriented programming is the ability to design for cod reuse. Two ways of reusing code are with inheritance implementation (IS-A relationship) and with object composition (HAS-A relationship).

IS-A Relationship: This can be accomplished with Class inheritance or Interface inheritance.

```
public interface auEntity
{
    void drawEntity();
}

public class auCircle : auEntity
{
    public auCircle()
    {
```

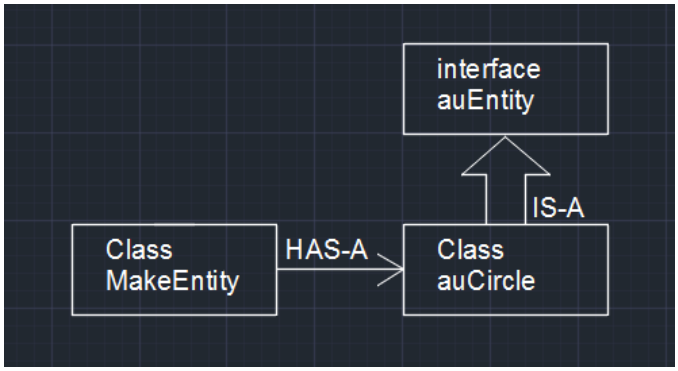
auCircle inherits from auEntity

```
            // Construct and collect data
        }
        public void drawEntity()
        {
            //Do entity creation
        }
    }
```

HAS-A Relationship: This is done by using instance variables that reference other objects.

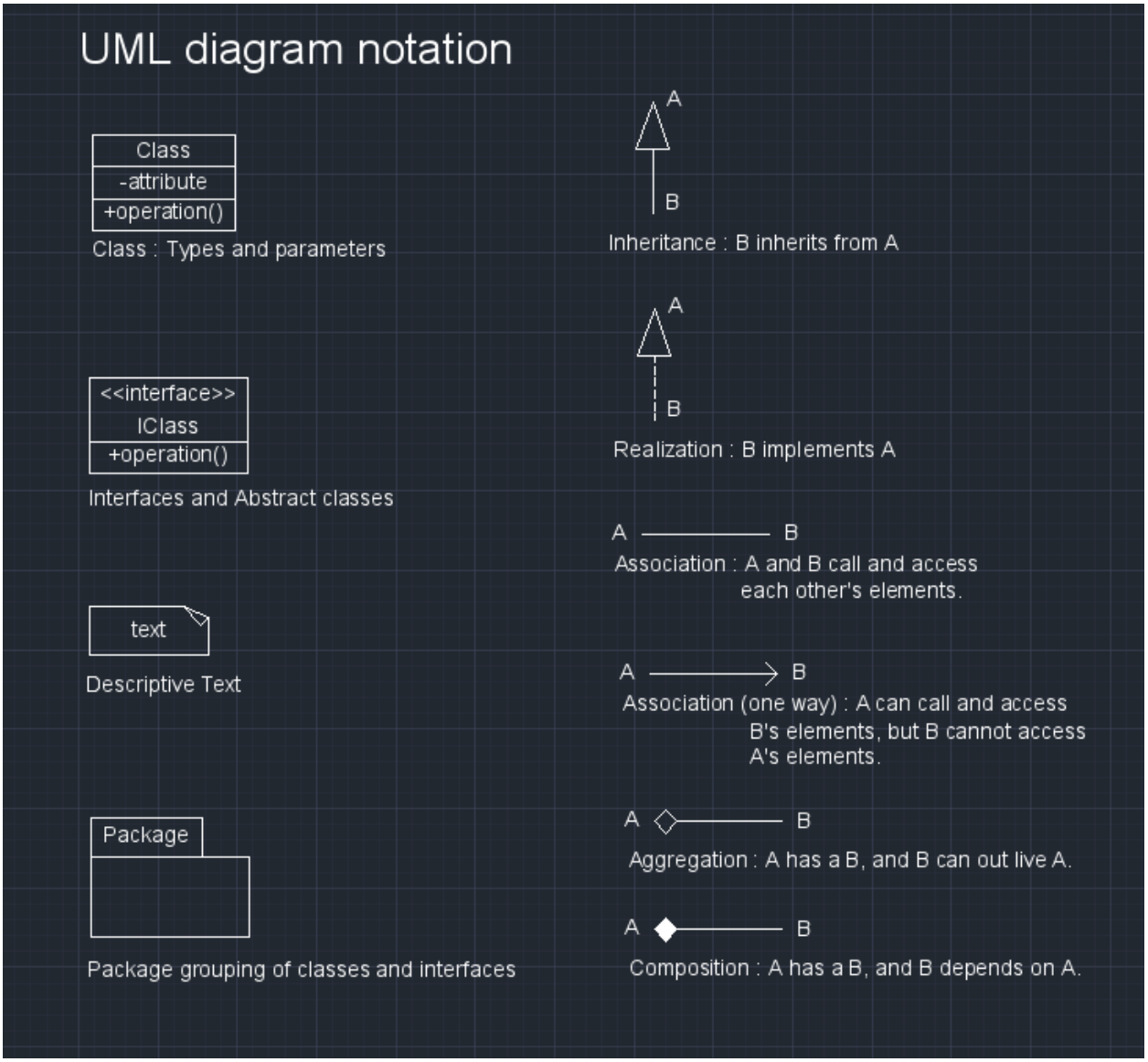
```
public class MakeEntity
{
    public auCircle thisEntity;

    public MakeEntity()
    {
        thisEntity = new auCircle();
        //do other data collection and property setting
        thisEntity.drawEntity();
    }
}
```

MakeEntity creates an instance of auCircle



Unified Modeling Language

Each of the following pattern has a Unified Modeling Language (UML) class diagram. UML is a universally accepted way of describing software in diagrammatic form. The diagrams in this handout use these UML features.



Understanding where and why to use design patterns

Using Design patterns can shorten the development process and help prevent issues that can cause major problems. Getting started with using design patterns should be to learn and experiment with implementing as many patterns as possible and never approach a software design with trying to fit in the use of a particular design pattern.

Creational Design Patterns

Provide ways to instantiate single objects or groups of related objects.

- **Abstract Factory**

  The abstract factory pattern is used to provide an interface for a group of similar factories or dependent objects without specifying the concrete classes. The client creates a concrete implementation of the abstract factory and then uses the interface of the factory to create the concrete objects. The client doesn't know which concrete objects it gets from the factories, because it uses only the interfaces of the products.

  Using an abstract factory pattern is a “super” factory that produces objects that follow a general pattern and at runtime the factory is paired with concrete factories to produce objects that follow the pattern of the abstract factory.

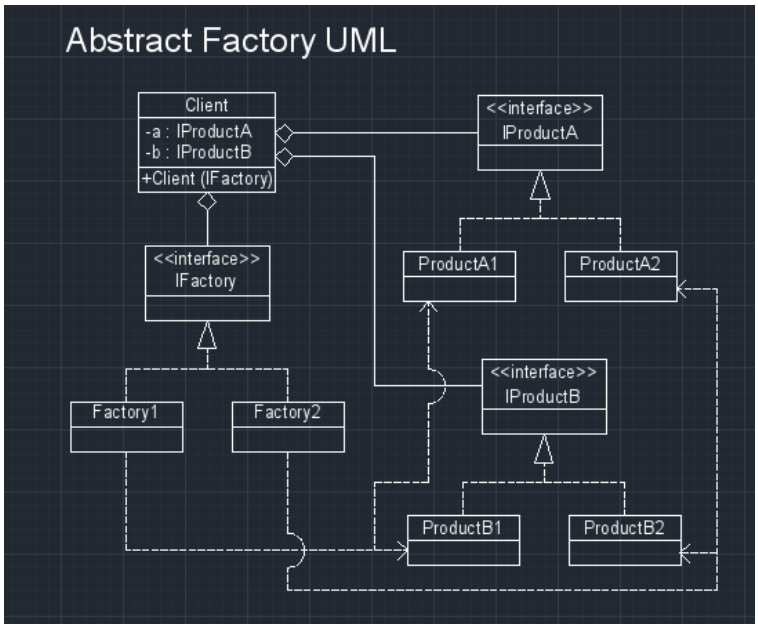
  The abstract factory pattern extends the factory design pattern that allows creating objects without being concerned about the actual classes of the objects being created.

  Use:

  - Creation of independent system for the representation of its products.
  - Configured system with multiple family of products.
  - Family of related product objects to be used together.
  - To expose the interface, but not the implementation of a class library of products.

  Participants:

  - Abstract Factory
  - Concrete Factory
  - Abstract Product
  - Concrete Product
  - Client



- **Builder**

  The builder pattern is used to separate the creation of complex objects from its representation to enable the construction process to create different representations. A director class is used to control the construction procedure.

  The builder pattern builds the complex objects with a step by step process, an interface defines the steps for the Builder object and a director class controls the object creation process.

  When an application needs to create an object which has to be constructed using many different objects, the client code gets cluttered with the details of the objects that need to be assembled to create the resulting object.

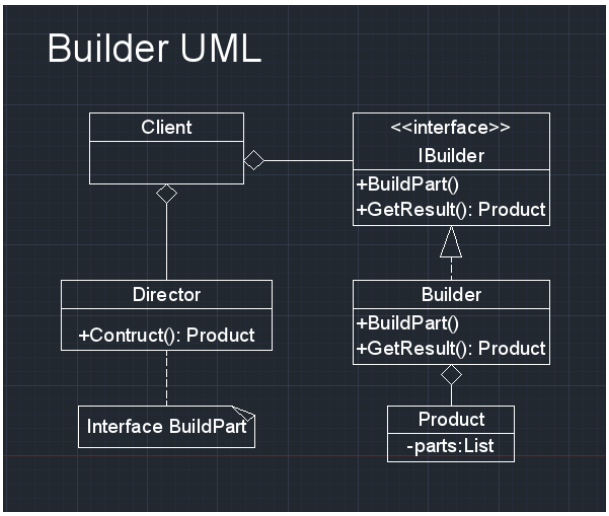
  The builder pattern defines a way to separate the builder object from its construction. The same construction method can create different representations of the object.

  Use:

  - Creation of an object that is independent of its parts.
  - Construction to allow for different representations.

  Participants:

  - Builder
  - Concrete builder
  - Director
  - Product



- **Factory Method**

  The factory method pattern is used to defer instantiation with class constructors replaced into subclasses. The process of object generation is abstracted so that the type of the object instantiated is determined at runtime. An interface is defined for creating an object, but the subclasses decide which class to instantiate.

  In the factory method pattern, the object is created without exposing the creation logic to the client and uses a common interface for reference to created objects.

  The factory method pattern uses inheritance, as object creation is delegated to subclasses that implement the factory method to create objects.

The factory method pattern can make a design a little more complicated, but also makes it more customizable.

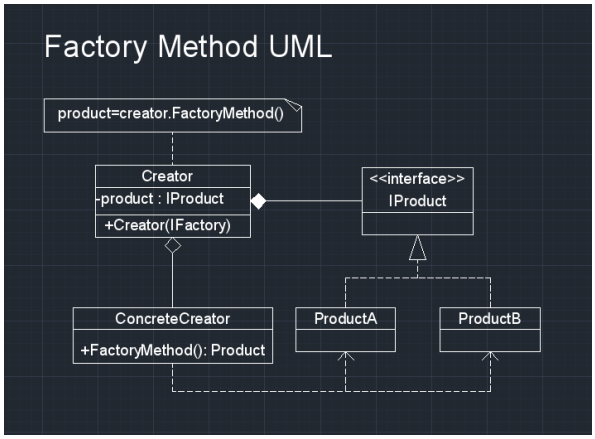
The factory method pattern is similar to abstract factory pattern but without the emphasis on groups and families of factories.

Use:

- The class needs to create objects that it cannot anticipate.
- The class needs its subclasses to specify the objects it creates.
- The class needs to delegate the responsibility to subclasses.

Participants:

- Product
- Concrete Product
- Creator
- Concrete Creator



- **Prototype**

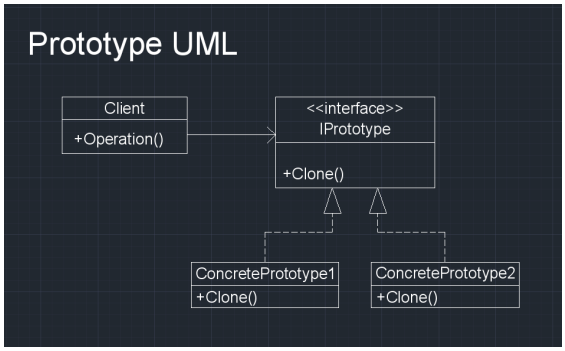
  The prototype pattern is used to instantiate new object by copying the properties of an existing object which creates an independent copy. This is used when the creating a new object is prohibitive for the application.

  Use:

  - To create Instances of objects that are specified at run-time.
  - To avoid creation of subclasses in the client.
  - To avoid creating a new object.
  - To create instances of a class to manage the state of the class.

  Participants:

  - Prototype
  - Concrete Prototype
  - Client



- **Singleton**

  The singleton pattern is used to ensure that only one instance of a particular class is ever created and provides access to the object. All further references to objects of the singleton class refer to the same underlying instance.

  This is useful when only one object is needed in the application or where multiple objects could contain mismatched property data.

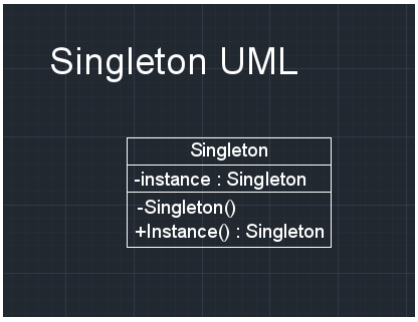
  The singleton pattern is often used in scenarios where it is not beneficial which introduces unnecessary restrictions in situations where a single instance of a class is not needed.

  Use:

  - There must be only one instance of a class.
  - The instance needs to be accessible from a known point in the application.
  - The instance needs to be extended and clients can access without code change.

  Participants:

  - Singleton



Structural Patterns

Provide ways to define relationships between classes or objects.

- **Adapter**

  The adapter pattern is used to convert the interface of a class into an interface the client expects. It allows incompatible types to work together by wrapping an existing class with a new interface that supports the interface required by the client.

  The adapter pattern is used when an existing component may offer functionality that needs to be used, but its interface is not compatible with the design of the system currently being developed.

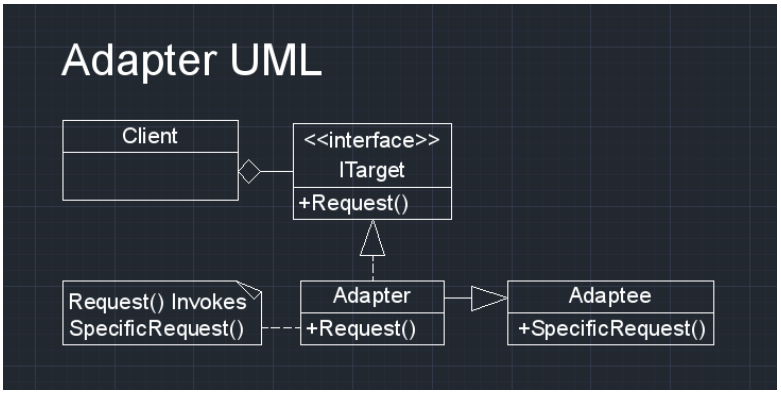
  The adapter pattern can help when reusing existing old components and designing new components for an application. There are always issues and things that are not quite right between the old and the new components.

Use:

- To use an existing class that its interface does not match your needs.
- To create a reusable class to make it compatible.
- To adapt the interface of a parent class.

Participants:

- Target
- Client
- Adaptee
- Adapter



- **Bridge**

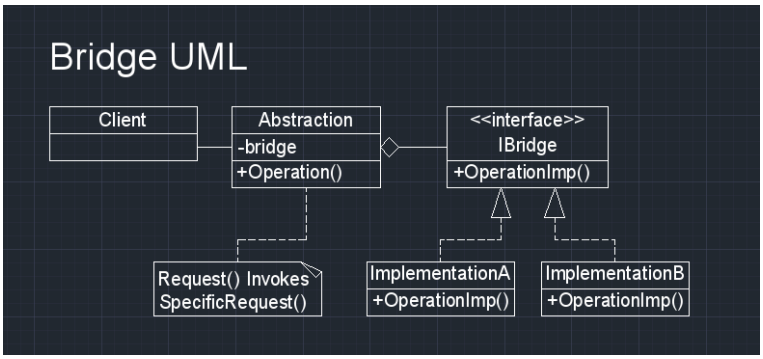
The bridge pattern is used to separate an abstraction from its implementation, providing a way that the implementation details can vary without modifying the abstraction.

Use:

- To avoid permanent binding between an abstraction and its implementation.
- The abstraction and its implementation should be extensible by sub classing.
- Changing the implementation should have no impact on its clients.
- To share an implementation among multiple objects.

Participants:

- Abstraction
- Refined Abstraction
- Implementer
- Concrete Implementer



- **Composite**

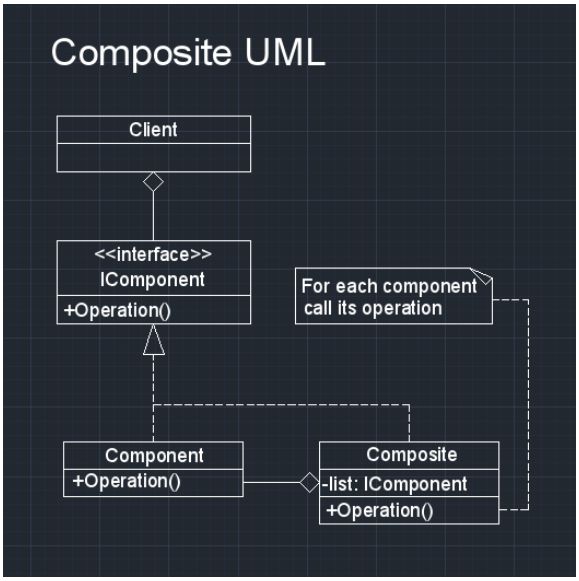
  The composite pattern is used to compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

  Use:

  - To represent part-whole hierarchies of objects.
  - To treat all objects in the structure uniformly.

  Participants:

  - Component
  - Leaf
  - Composite
  - Client



- **Decorator**

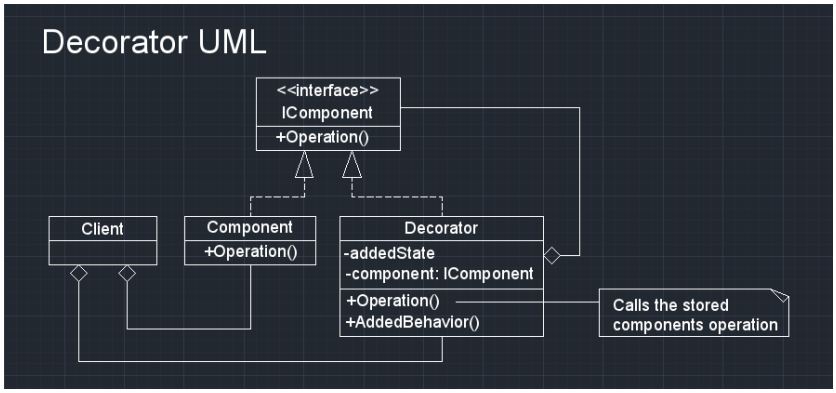
  The decorator pattern is used to attach additional responsibilities to an object dynamically or alter the functionality of objects at runtime by wrapping them in an object of a decorator class. This provides a flexible alternative to using sub-classing or inheritance to modify behavior.

  Use:

  - To add responsibilities to objects dynamically and transparent.
  - Ability to remove added responsibilities.
  - When extending by sub classing is not possible or impractical.

  Participants:

  - Component
  - Concrete component
  - Decorator
  - Concrete decorator



- **Facade**

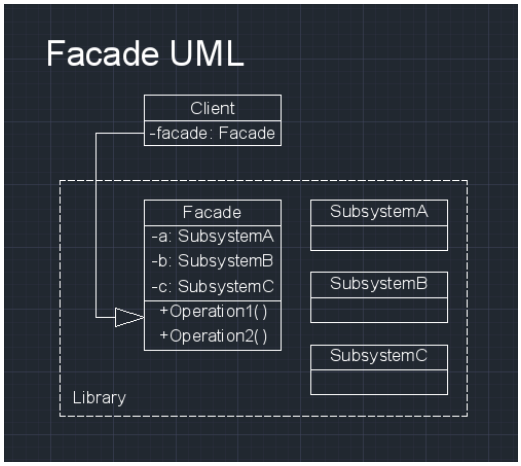
  The facade pattern is used to define a unified interface to a set of interfaces in a subsystem. Facade wraps a complicated subsystem with a simpler interface that makes the subsystem easier to use.

  Use:

  - To create a complex system easier to use and understand.
  - To make a system library more readable
  - To add layers to a sub system.

  Participants:

  - Façade
  - Sub System classes.



- **Flyweight**

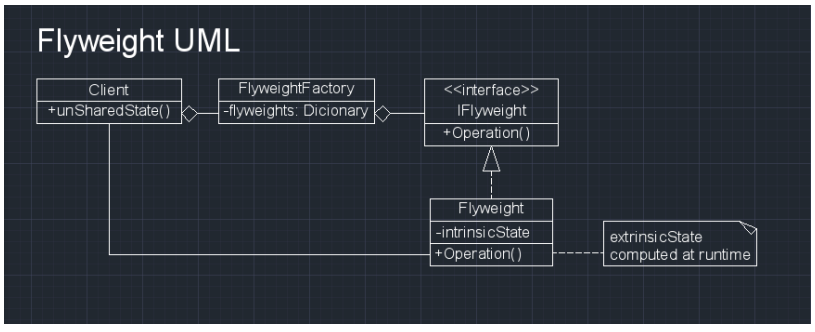
  The flyweight pattern is used for sharing to support large numbers of fine-grained objects efficiently.

  Use:

  - For applications that have many objects.
  - Minimize memory usage by sharing data with objects.
  - Groups of objects can be defined with shared objects
  - Application is not dependent on object identity.

  Participants:

  - Flyweight
  - Concrete Flyweight
  - Unshared Concrete Flyweight
  - Flyweight Factory
  - Client



- **Proxy**

  The proxy pattern is used to provide a surrogate or placeholder object to control access to it. The proxy pattern provides an extra level of indirection to support distributed, controlled, or intelligent access.

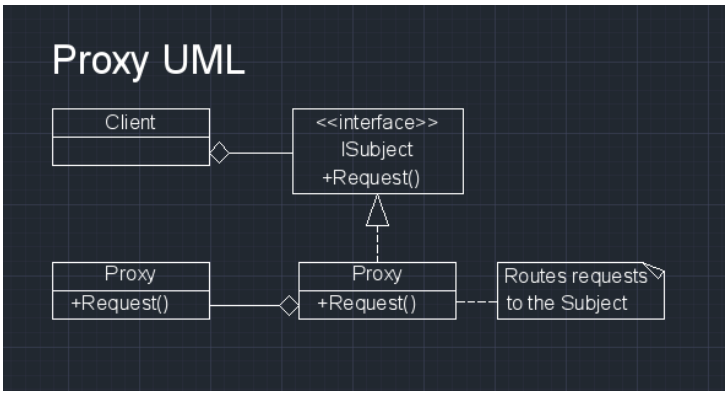
  The proxy design pattern provides a way to create a wrapper class to provide a simple interface to existing objects. The proxy wrapper class can be used for adding additional functionality to an existing object without changing the code of the existing object.

  Use:

  - Add control of security to an existing object.
  - Making complex objects easier to access.
  - To provide an interface for a remote resource.
  - To create extensive memory objects on demand.
  - Add thread-safety to an existing class.

  Participants:

  - Proxy
  - Subject
  - Real Subject



Behavioral Patterns

Provide communication between classes and objects.

Behavioral object patterns use composition instead of inheritance and behavioral class patterns use inheritance between classes, examples of behavioral class patterns are the “Template Method” and “Interpreter”.

- **Chain of Responsibility**

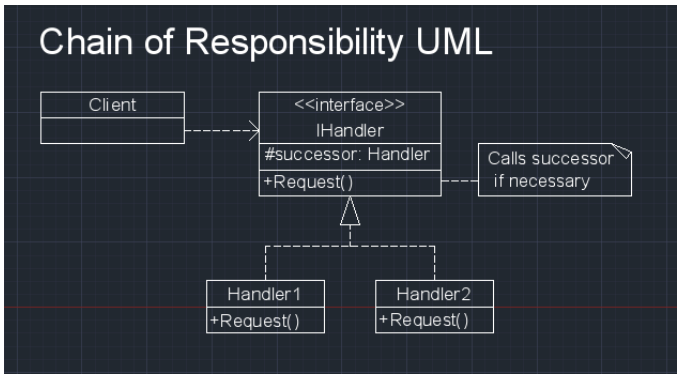
  The chain of responsibility pattern is used to avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

  Use:

  - When more than one object can handle a request.
  - To issue a request to an object without specifying the receiver.
  - Objects to receive a request are specified dynamically.
  - When loose coupling of objects is desired.

  Participants:

  - Handler
  - Concrete Handler
  - Client



- **Command**

  The command pattern is used encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations. The command may then be executed immediately or held for later use.

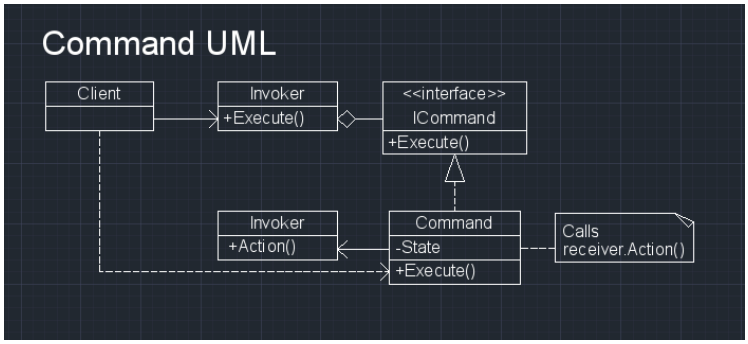
  A command method can also be thought of as a Transaction or Action pattern.

  Use:

  - Parameterize objects by the command to be performed.
  - Object Oriented replacement for callbacks.
  - Execute requests at different times or setup queues.
  - Allows creating an undo

Participants:

- Command
- Concrete Command
- Client
- Invoker



- **Interpreter**

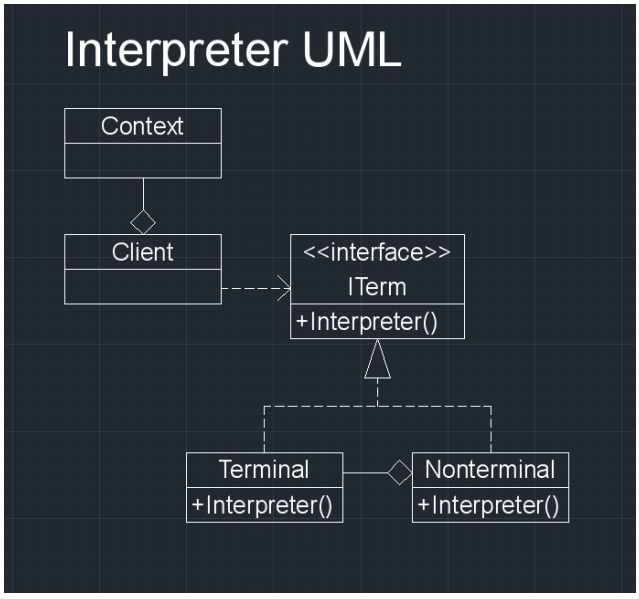
  The interpreter pattern is used to define a representation for a language or notations grammar with an interpreter that uses the representation for instructions to interpret sentences in the language or notation.

  Use:

  - To interpret grammar, best used when the grammar is simple.
  - Best when efficiency is not a critical concern
  - Used with specified query languages like SQL

  Participants:

  - Abstract Expression
  - Terminal Expression
  - Nonterminal Expression
  - Context
  - Client



- **Iterator**

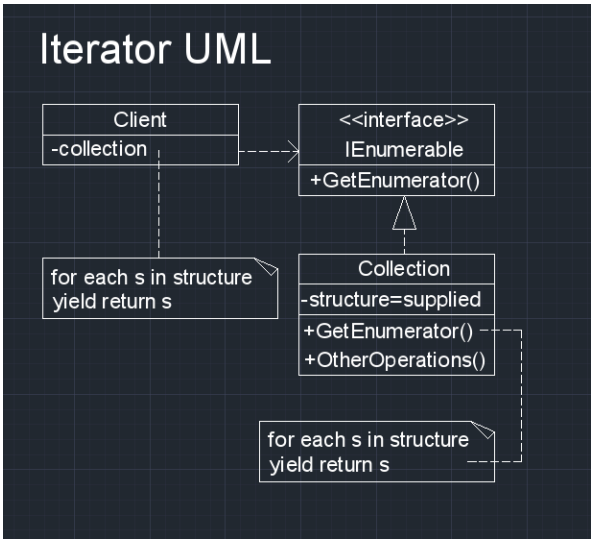
  The iterator pattern is used to provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

  Use:

  - Access an objects contents without exposing its underlying representation.
  - Provide an interface to traverse the objects structure.

  Participants:

  - Iterator
  - Concrete Iterator
  - Aggregate
  - Concrete Aggregate



- **Mediator**

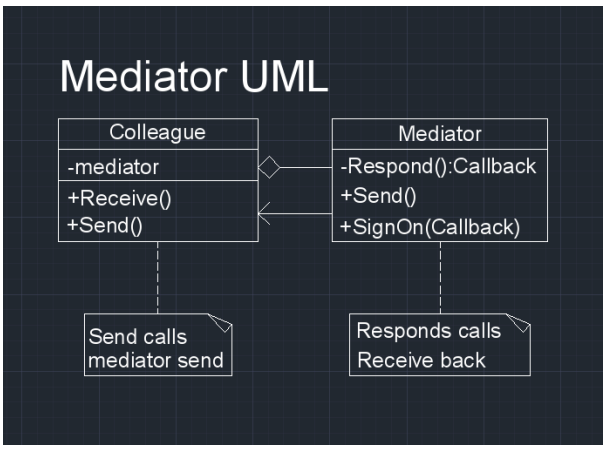
  The mediator pattern is used to define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

  Use:

  - When objects communicate to simplify the understanding of the dependencies.
  - To reuse an object that communicates with other objects.
  - To distribute a behavior between classes without sub classing.

  Participants:

  - Mediator
  - Concrete Mediator
  - Colleague classes



- **Memento**

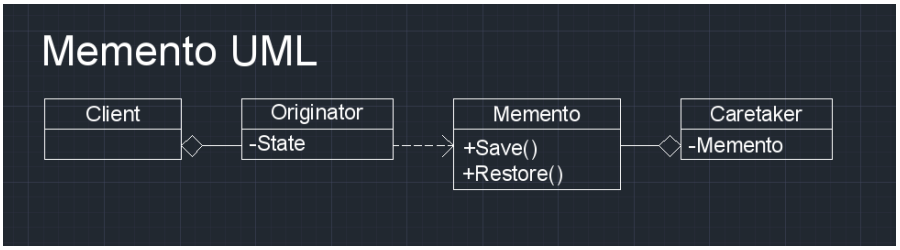
  The memento pattern is used to capture and externalize the current state of an object's internal state and store it in such a manner that it can be restored at a later time without breaking the rules of encapsulation.

  Use:

  - The state of an object needs to be saved so that it can be restored later.
  - When obtaining the state would expose the implementation.

  Participants:

  - Memento
  - Originator
  - Caretaker



- **Observer**

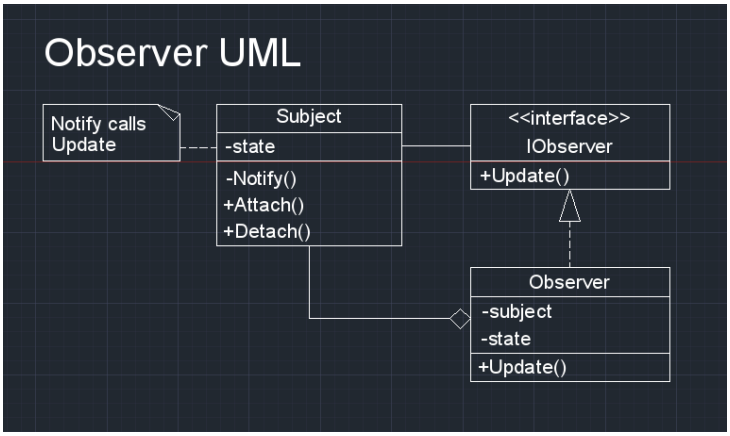
  The observer pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

  Use:

  - When an abstraction has two aspects and they are dependent on each other.
  - When an object changes and multiple objects need to be notified or changed.
  - When objects need to notify other objects.

  Participants:

  - Subject
  - Observer
  - Concrete Subject
  - Concrete Observer



- **State**

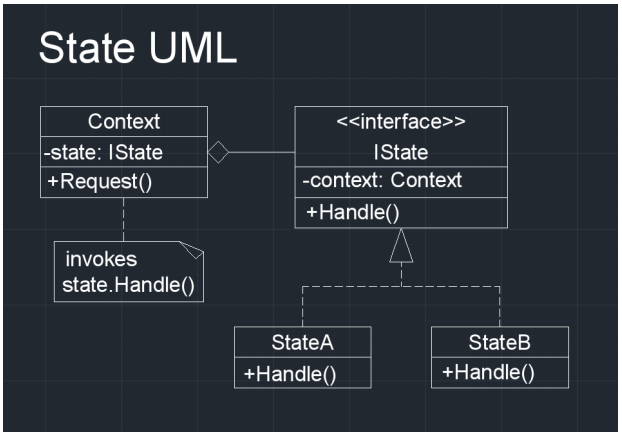
  The state pattern is used to alter the behavior of an object when its internal state changes. The object will appear to change its class. The pattern allows the class for an object to apparently change at run-time.

  Use:

  - When an object depends on its state and the state changes at run-time.
  - To treat the object's state as an independent object.

  Participants:

  - Context
  - State
  - Concrete State Sub Classes



- **Strategy**

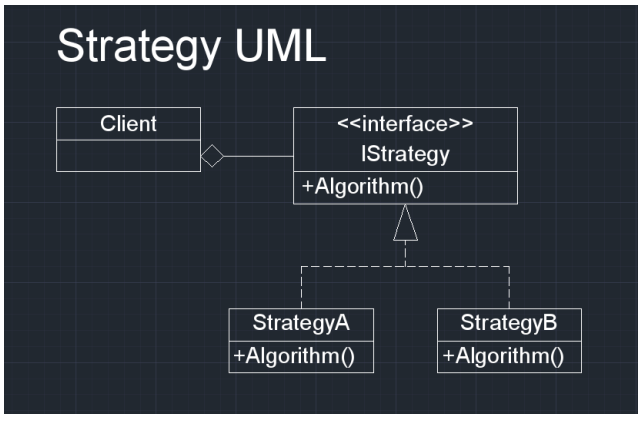
  The strategy pattern defines a family of algorithms, encapsulate each one, and make them interchangeable. The strategy pattern lets the algorithm vary independently from the clients that use it. It captures the abstraction in an interface and hides the implementation details in derived classes.

  Use:

  - When a class needs to be configured for multiple behaviors.
  - When different variations of an algorithm are required.
  - To avoid exposing complex structures.
  - When many conditionals are required they can be moved into a strategy class.

  Participants:

  - Strategy
  - Concrete Strategy
  - Context



- **Template Method**

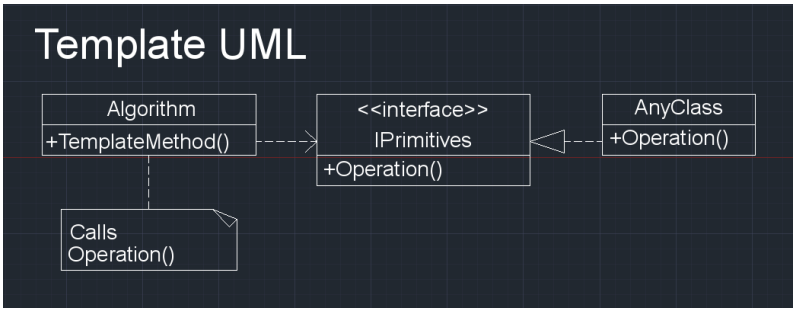
  The template method pattern is used to define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. The template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

  Use:

  - To implement an algorithm and let the sub classes implement the behavior.
  - To avoid code duplication in sub classes.
  - To control sub class extensions.

  Participants:

  - Abstract Class
  - Concrete Class



- **Visitor**

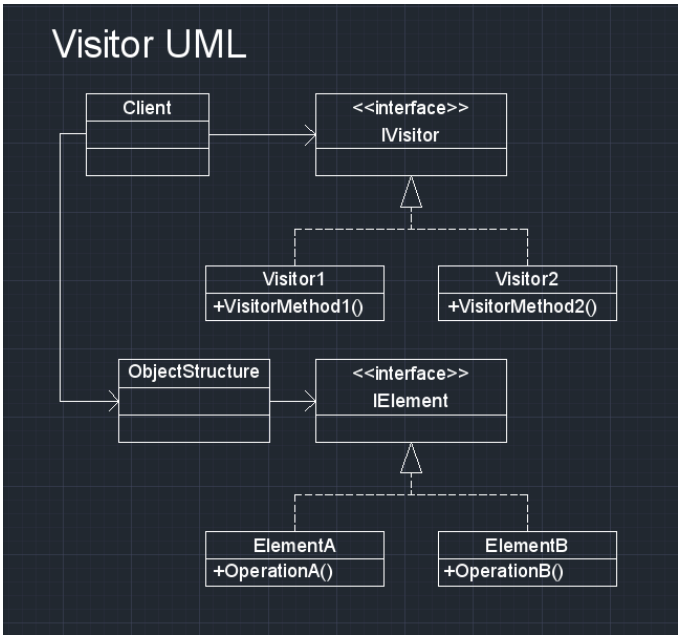
  The visitor pattern is used to represent an operation to be performed on the elements of an object structure. The visitor pattern allows defining a new operation without modifying the operating classes.

  Use:

  - To operate on objects of an object structure that depend on other classes.
  - To define related operations in a single class to keep them together.
  - When classes defining objects rarely change.

  Participants:

  - Visitor
  - Concrete Visitor
  - Element
  - Concrete Element
  - Object Structure



Applying design pattern code and refactoring existing code for design patterns

When starting a new project it would be good practice to setup and design using design patterns, but often projects evolve from some test code or existing code that often does not use design patterns. Also, it is seldom beneficial to start a project with the attitude that it will be structured with a parrticular set of design patterns. This approach will often be unsucessful due to too much effort required to make a desired pattern fit in unknown code. It is often better to design the code in the same way that you have always worked and then refactor. As design patterns are used more often in the code, they become easier to implement.

To refactor code to use a design pattern is no different than refactoring other existing code that needs cleanup or added functionality. The primary difference is in building the structure to fit within the desired pattern. To do this requires a good understanding of object oriented programming and of the patterns that are being used.

After spending some time learning different design patterns, the approach to refactoring existing code should be the same as with any refactoring is to break the code into simpler classes and methods, then determine what design pattern best suits for a solution. Then build the generic framework of the selected design pattern and move the existing code into the new structure.

When to use design patterns?

This is a difficult question to answer, since different patterns have different uses it would require understanding the different patterns and the existing code. Typically when something is being done requiring a lot of conditions there are several design patterns that can be used. Also, if the project is not reusing code or typical code is being replicated there are several patterns that could simplify reuse of code.

How to learn the different design patterns?

To learn different design patterns requires a lot of time, there are several websites with code samples and definitions. The above descriptions of the patterns were derived from several websites and the Design patterns book mentioned above. Would suggest taking a few patterns that fit the type of code that you typically do, learn them and use them wherever they apply, then keep expanding learning new patterns.

Why would I change existing code to use design patterns?

The main reason to refactor existing code is for readability and reuse. As the code is refactored to take advantage of design patterns it will be easier to maintain and add additional functionality. Keep in mind one of the main aspects of design patterns which is : “Design patterns are ways to write well defined solutions to problems that someone else has solved”.

How design patterns can help you build better applications

Design patterns provide defined structures that assist in solving problems. When working on a solution for a problem, there are many variations that need to be considered for the solution to problems to determine a design pattern that fits. The problem will often need to be broken into smaller parts to find a solution using a design pattern. Learning design patterns is an important step to building better applications due to the time tested solutions that design patterns provide.

Design patterns provide a common vocabulary and improve code readability for developers that are familiar with the patterns making maintenance and adding functionality easier and faster. In the development process, if a developer wants to use a particular design pattern for a problem there is a common point of reference to discuss if it will solve the problem without implementing the solution first. Programmers and developers encounter problems and have used design pattern 'solutions'.

Design Patterns the good:

- **Readability:** Using design patterns creates code that is more understandable to all developers who have taken the time to learn design patterns. The code is easier to understand due to design patterns being defined with standard object-oriented programming techniques instead of each developer doing it in his own way that often does not use well defined techniques.
- **Maintainability:** Using design patterns makes the code easier to maintain because it is more understandable and again will use standard object-oriented programming techniques. This makes changing and adding additional functionality much easier.
- **Communication:** Design patterns provide a common vocabulary and assist in communicating design goals amongst different programmers and developers.
- **Intention:** The code is easier to understand when another programmer or developer is learning the code.
- **Re-use:** Using common solutions to common problems can assist in making the code easier to use for other solutions. Design patterns can help prevent small problems that can become major problems.
- **Less code:** Code can be derived for common functionality from common base classes.
- **Time Tested:** They have been used and tested to provide proven and sound solutions.

Design patterns the bad:

- **Levels of indirection:** Many design patterns provide extra levels of indirection making the code more complex and result in the code not being more readable and maintainable.
- **Learning:** As with other programming techniques, learning when and how to use design patterns can take some effort. When learning design patterns, they are often abused and used where they do not fit. Sometimes simple tasks do not require the extra work of being solved by using a design pattern.
- **Interpretations:** In the learning process and from misunderstanding the usage of design patterns can be interpreted in different ways and can result in a usage that is not a good fit.
- **More Code:** Some design patterns require multiple layers of sub classing and can sometimes result in extra code being required to satisfy the requirements of using the design pattern.
- **Wrong Use:** A good example is that the singleton pattern is often used when a static class or variable would be acceptable. As with other techniques in an effort to use a newly learned technique, some programmers and developers will use design patterns where they may not fit.

Have sample code using design patterns that can be immediately used in your applications.

Please download the applications that have been uploaded to support this class.

They are C# applications to illustrate using different design patterns for AutoCAD plugins.

Design pattern links used for reference to create this document.

The following links were used in creating this document and provide good reference for learning design patterns.

Samples and descriptions:

https://en.wikipedia.org/wiki/Design_Patterns

https://sourcemaking.com/design_patterns

http://www.dofactory.com/net/design-patterns

http://www.blackwasp.co.uk/GofPatterns.aspx

http://code.tutsplus.com/articles/a-beginners-guide-to-design-patterns--net-12752