|  |  |
| --- | --- |
|  | LAB | Gang of Four Design Patterns |
|  | WORKSHOP | Design Patterns |
|  | Crystal Tenn  crtenn@microsoft.com |

# SETUP LAB ENVRIOMENT

## Prerequisites

To perform the tasks in this lab you need following:

* Visual Studio 2015 or 2017, any version

## Objectives

In this lab you will complete following tasks:

* Work through samples of code and read through why you would use each design pattern.
* Each pattern has a definition and a frequency of use rated from 1-5. 5 is highly likely to be utilized. 1 is unlikely to be utilized. Each section of design patterns (creational, structural, behavioral) will have a generic definition, reiterate the list of designs in them, and an easy way to remember which designs are within each set.
* For any design patterns 1/5 to 3/5 frequency of use, I will have you step through the code but there will be no need for you to add any code as these are not as high priority.
* For some patterns used more commonly, you will add portions of the code. Pay close attention to the highly used design patterns!!! They will be marked as 4/5 or 5/5 for frequency of use.
* At the end you will have a working solution with 23 projects with one working sample per design pattern.

### Estimated Completion Time: 60 minutes.

## Reference

**Creational Patterns (5)**

* 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 (7)**

* 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
* Façade: 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 (11)**

* 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

## Task: Creational Patterns

Creational Patterns deal with **controlling the creation objects**. These design patterns all relate to rules to do with creation of instances(s) of classes.

Try to remember this phrase: **“**Both types of **Factory** (Abstract Factory and Factory) patterns help you **Build** (Builder) a **Single** (Singleton) **Prototype.”**

* 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

**Abstract Factory:** “The super factory that creates other factories.”

Frequency of Use: 5/5 High

Definition: The abstract factory pattern provides a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes. In normal usage, the client software creates a concrete implementation of the abstract factory and then uses the generic interface of the factory to create the concrete objects that are part of the theme. The client doesn't know (or care) which concrete objects it gets from each of these internal factories since it uses only the generic interfaces of their products. This pattern separates the details of implementation of a set of objects from their general usage and relies on object composition, as object creation is implemented in methods exposed in the factory interface.

1. Open the CoreDesignPatterns solution file in Visual Studio.
2. Open the CreationalPatterns folder. Go to the AbstractFactory project.
3. Look at the Carnivore and Herbivore classes.
4. Look at each of the animal files and notice which animals are Carnivores and Herbivores (they will inherit from those classes)
5. Create a file called ContinentFactory with the following code:

namespace AbstractFactory

{

/// <summary>

/// The 'AbstractFactory' abstract class

/// </summary>

public abstract class ContinentFactory

{

public abstract Herbivore CreateHerbivore();

public abstract Carnivore CreateCarnivore();

}

}

1. Create a file called AmericaFactory with the following code:

namespace AbstractFactory

{

/// <summary>

/// The 'ConcreteFactory2' class

/// </summary>

public class AmericaFactory : ContinentFactory

{

public override Herbivore CreateHerbivore()

{

return new Bison();

}

public override Carnivore CreateCarnivore()

{

return new Wolf();

}

}

}

1. Create a file called AfricaFactory with the following code:

namespace AbstractFactory

{

/// <summary>

/// The 'ConcreteFactory1' class

/// </summary>

public class AfricaFactory : ContinentFactory

{

public override Herbivore CreateHerbivore()

{

return new Wildebeest();

}

public override Carnivore CreateCarnivore()

{

return new Lion();

}

}

}

1. Create a file called AnimalWorld with the following code:

namespace AbstractFactory

{

/// <summary>

/// The 'Client' class

/// </summary>

public class AnimalWorld

{

private Herbivore \_herbivore;

private Carnivore \_carnivore;

// Constructor

public AnimalWorld(ContinentFactory factory)

{

\_carnivore = factory.CreateCarnivore();

\_herbivore = factory.CreateHerbivore();

}

public void RunFoodChain()

{

\_carnivore.Eat(\_herbivore);

}

}

}

1. Add the following to Program.cs

using System;

namespace AbstractFactory

{

class Program

{

/// <summary>

/// Entry point into console application.

/// </summary>

public static void Main()

{

// Create and run the African animal world

ContinentFactory africa = new AfricaFactory();

AnimalWorld world = new AnimalWorld(africa);

world.RunFoodChain();

// Create and run the American animal world

ContinentFactory america = new AmericaFactory();

world = new AnimalWorld(america);

world.RunFoodChain();

// Wait for user input

Console.ReadKey();

}

}

}

1. Add a breakpoint to Program.cs on line 14. Step through the code and understand how the America and Africa factory are implementing the ContinentFactory. And, understand how each country’s factory are helping to build the AnimalWorld.

**Builder**:

Frequency of Use: 2/5 Low

Definition: The builder pattern uses another object, a builder, that receives each initialization parameter step by step and then returns the resulting constructed object at once. The builder pattern has another benefit: It can be used for objects that contain flat data (HTML code, SQL query, X.509 certificate…), that is to say, data that can't be easily edited step by step and hence must be edited at once.

1. Open the CreationalPatterns folder. Go to the Builder project.
2. View the Vehicle class.
3. View the Motorcycle Builder, ScooterBuilder, and CarBuilder classes.
4. View the Shop class.
5. Go into Program.cs and put a breakpoint on line 29 and see how a Scooter is built in the Shop. Do the same for Car and Motorcycle on line 33 and 37 as you continue to step through the code.

**Factory Method:**

Frequency of Use: 5/5 High

Definition: Deals with the problem of creating objects without having to specify the exact class of the object that will be created. This is done by creating objects by calling a factory method—either specified in an interface and implemented by child classes, or implemented in a base class and optionally overridden by derived classes—rather than by calling a constructor.

1. Open the CreationalPatterns folder. Go to the FactoryClient and FactoryMethod projects.
2. Please view the files in the Factory Method project in the Concrete and Contracts file directories.
3. In the empty Factory Directory, find the empty class called StrawberryFactory.cs and add the following code:

using System;

using FactoryMethod.Concrete;

using FactoryMethod.Contracts;

namespace FactoryMethod.Factory

{

public class StrawberryFactory

{

//Factory method responsbile for returning instantiated object at runtime

//Based upon month parameter, returns strawberry sourcer to client via an interface

public static IDistributor SourceOrder(int month)

{

// Ensure that month parameter is valid

if (month <= 0 || month > 12)

throw new ArgumentOutOfRangeException("month", "Argument must be greater than 0 and no larger than 12");

if (month > 3 && month < 8)

// Configure contrete class and return interface reference

return new California {Location = "California",

Carriers = Carriers.JBHunt,

GrossPricePerPeck = 1.27m,

FdaInspectionRequired = false,};

if (month > 10 || month < 2)

// Configure contrete class and return interface reference

return new Mexico {Location = "Mexico",

Carriers = Carriers.RoadwayExpress,

GrossPricePerPeck = .11m,

FdaInspectionRequired = true,};

if (month > 8 && month < 10)

// Configure contrete class and return interface reference

return new Washington {Location = "Washington",

Carriers = Carriers.UPS,

GrossPricePerPeck = .87m,

FdaInspectionRequired = false,};

return new NullDistrbutor {Location = "NullDistrbutor",

Carriers = Carriers.None,

GrossPricePerPeck = 0m,

FdaInspectionRequired = false,};

}

}

}

1. In the FactoryClient project’s GroceryStoreClient.cs file, add the following code:

using System;

using FactoryMethod.Contracts;

using FactoryMethod.Factory;

namespace FactoryMethod

{

internal class GroceryStoreClient

{

private static void Main()

{

var program = new GroceryStoreClient();

program.OrderStrawberries();

}

private void OrderStrawberries()

{

//Holds reference to strawberry sourcer

IDistributor sourcer;

//Dynamically assign my favorite strawberry month

sourcer = StrawberryFactory.SourceOrder(5);

Console.WriteLine("I love strawberries in {0} in which we get strawberries from {1} @ $ {2} per peck",

"May",

sourcer.Location,

sourcer.GrossPricePerPeck);

//Dynamically assign my favorite strawberry month

sourcer = StrawberryFactory.SourceOrder(12);

Console.WriteLine("I hate strawberries in {0} in which we get strawberries from {1} @ $ {2} per peck",

"December",

sourcer.Location,

sourcer.GrossPricePerPeck);

Console.ReadLine();

}

}

}

1. Drop a breakpoint on line 21 and run this project. Understand how the SourceOrder method is returning the type of strawberry based on the month that is passed in.

**Prototype**: The prototype pattern is a creational design pattern in software development. It is used when the type of objects to create is determined by a prototypical instance, which is cloned to produce new objects. This pattern is used to:

* avoid subclasses of an object creator in the client application, like the factory method pattern does.
* avoid the inherent cost of creating a new object in the standard way (e.g., using the 'new' keyword) when it is prohibitively expensive for a given application.

Frequency of Use: 3/5 Medium

Definition:

1. Go to the project called Prototype.
2. View the ColorPrototype class. See how colors can be saved in this ColorPrototye colors Dictionary and retrieved by key.
3. View the Color and ColorManager classes.
4. Drop a breakpoint on line 19 and step through the code. Understand how we are naming colors by their standard RGB color codes. Then see how the clones can get the selected colors by the name we input.

**Singleton**: Restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system. The concept is sometimes generalized to systems that operate more efficiently when only one object exists, or that restrict the instantiation to a certain number of objects. The term comes from the mathematical concept of a singleton.

Frequency of Use: 4/5 Medium High

Definition:

1. Go to the project called Singleton.
2. Create a LoggingService.cs file (Read through each line and the comments and understand why every piece is there):

using System;

namespace Singleton

{

// Marked Sealed as to prevent any deriving, which could add instances.

public sealed class LoggingService : IDisposable

{

// Implements Double-Check Locking Pattern ==> https://msdn.microsoft.com/en-us/library/ff650316.aspx

private static readonly object \_lockObject = new object();

// Holds reference of instance of itself.

// Make volatile to ensure that assignment to variable completes before

// variable can be accessed

private static volatile LoggingService \_instance;

// Might create loging service here, but is is thread safe?

//private static LoggingService \_instance = new LoggingService();

private readonly string \_returnValue;

// Private constructor is dead-giveaway of LoggingService object - prevents

// other classes from creating instances of it.

// Note here how object implements lazy-instantiation -- does not

// instantiate object until absolutely needed by application.

private LoggingService()

{

\_returnValue = "Successfully Logged";

}

// Here is how we instantiate LoggingService: Publically-scoped static property

// that returns reference of class.

// Property determine whether object has been instantiated.

public static LoggingService Instance

{

get

{

if (\_instance == null)

{

// CLR cannot ensure thread safety

// Double-check locking approach solves thread concurrency issues

// Add 'lock' statement to ensure only thread can enter critical section of code

// 'lock' will block until it is released

lock (\_lockObject)

// Implement "double-check" lock to prevent multiple threads from

// creating separate instances of singleton at same time

if (\_instance == null)

//Finally, instantiate singleton class

\_instance = new LoggingService();

}

return \_instance;

}

}

public void Dispose()

{

// Release objects

\_instance = null;

}

public string LogMessage(string message)

{

// Implement Logging Code.

// Most likely a factory pattern that

// would return the current logging framework.

return \_returnValue;

}

}

}

1. Go to the SingletonLauncher class, uncomment the code there.

using System;

namespace Singleton

{

internal class SingletonLauncher

{

private static void Main(string[] args)

{

var user1 = LoggingService.Instance;

Console.WriteLine(user1.LogMessage("First Message to Log"));

var user2 = LoggingService.Instance;

Console.WriteLine(user2.LogMessage("Second Message to Log"));

var user3 = LoggingService.Instance;

Console.WriteLine(user3.LogMessage("Third Message to Log"));

Console.ReadLine();

}

}

}

1. Add a breakpoint to line 9 of the SingletonLauncher.cs file. Step through the code and understand how the Singleton is created.

## Task: Structural Patterns

Structural design patterns are concerned with how classes and objects can be composed, to form larger structures. They are design patterns that ease the design by identifying a simple way to realize relationships between entities.

Try to remember: A **Bridge** is a **Structure** that is **Composed** (Composite) of many small parts, and acts as a **Proxy** between two pieces of land. You can **Adapt** the Bridge to make it fancy by **Decorating** (Decorator) it, though it will only be a **Facade**. You can **Fly** (Flyweight) over a **Bridge**.

* 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
* Façade: A single class that represents an entire subsystem
* Flyweight: A fine-grained instance used for efficient sharing
* Proxy: An object representing another object

**The Adapter Pattern**

Frequency of Use: 4/5 Medium High

Definition: (also known as Wrapper, an alternative naming shared with the Decorator pattern) that allows the interface of an existing class to be used as another interface. It is often used to make existing classes work with others without modifying their source code. An example is an adapter that converts the interface of a Document Object Model of an XML document into a tree structure that can be displayed.

For Example: A card reader acts as an adapter between memory card and a laptop. You can plugin the memory card into card reader and card reader into the laptop so that memory card can be read via laptop.

1. Go to the Adapter pattern project under the StructuralPatterns folder.
2. View the code in the ChemicalDatabank.cs and LegacyCompound.cs files.
3. Add a new class CompoundAdapter.cs where you put the following code:

using System;

namespace Adapter

{

/// <summary>

/// The Adapter class

/// Adapts the Adaptee, ChemicalDataBank, to Target interface, the

/// legacy Compound class.

/// </summary>

public class CompoundAdapter : LegacyCompound

{

// CompoundAdapter uses composition to communicate with Legacy interface.

private ChemicalDatabank \_bank;

// Constructor

public CompoundAdapter(string name) : base(name)

{

}

// Override and adapt Display to extract specific information

// from the legacy ChemicalDatabank.

public override void Display()

{

// The Adaptee

\_bank = new ChemicalDatabank();

\_criticalBoilingPoint = \_bank.GetCriticalPoint(\_chemical, "B");

\_meltingPoint = \_bank.GetCriticalPoint(\_chemical, "M");

\_molecularWeight = \_bank.GetMolecularWeight(\_chemical);

\_molecularFormula = \_bank.GetMolecularStructure(\_chemical);

// Call base display

base.Display();

Console.WriteLine(" Formula: {0}", \_molecularFormula);

Console.WriteLine(" Weight : {0}", \_molecularWeight);

Console.WriteLine(" Melting Point: {0}", \_meltingPoint);

Console.WriteLine(" Critical Boiling Point: {0}", \_criticalBoilingPoint);

}

}

}

1. In the Program.cs file, add the following code:

using System;

namespace Adapter

{

public class Program

{

private static void Main()

{

// Note how we upcast each reference to Compound, the Target class.

// Client interacts directly with base class via polymorphism.

// Get data for chemical compound from legacy source

LegacyCompound sulfur = new LegacyCompound("Sulfur");

sulfur.Display();

// Get data for chemical compound from databank via Adapter

LegacyCompound water = new CompoundAdapter("Water");

water.Display();

LegacyCompound benzene = new CompoundAdapter("Benzene");

benzene.Display();

LegacyCompound ethanol = new CompoundAdapter("Ethanol");

ethanol.Display();

Console.ReadKey();

}

}

}

1. Add a breakpoint in your Program.cs file on line 13 and step through all the code. Understand how we adapt the LegacyCompound class to the CompoundAdapter class and get information from the chemical database bank class.

**Bridge Pattern**

Frequency of Use: 3/5 Medium

Definition: Meant to "decouple an abstraction from its implementation so that the two can vary independently". The bridge uses encapsulation, aggregation, and can use inheritance to separate responsibilities into different classes.

1. Go to the Bridge pattern project.
2. View the code in the CustomersBase and DataObject files first.
3. View the code in the Customers file.
4. View the code in the CustomersData file.
5. Go to the Program.cs file. Add a breakpoint on line 21. Step through the code. Understand how the CustomersBase is the Abstraction class here and how it works with the other files.

**Composite Pattern**

Frequency of Use: 4/5 Medium High

Definition: The composite pattern describes a group of objects that is treated the same way as a single instance of the same type of object. The intent of a composite is to "compose" objects into tree structures to represent part-whole hierarchies. Implementing the composite pattern lets clients treat individual objects and compositions uniformly.

1. Go to the Composite pattern project.
2. View the code in the Component class.
3. View the code in the Composite and Leaf class.
4. Go to the Program.cs class and add the following code:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Composite

{

class Program

{

/// <summary>

/// Entry point into console application.

/// </summary>

static void Main()

{

// Create a tree structure

Composite root = new Composite("root");

root.Add(new Leaf("Leaf A"));

root.Add(new Leaf("Leaf B"));

Composite comp = new Composite("Composite X");

comp.Add(new Leaf("Leaf XA"));

comp.Add(new Leaf("Leaf XB"));

root.Add(comp);

root.Add(new Leaf("Leaf C"));

// Add and remove a leaf

Leaf leaf = new Leaf("Leaf D");

root.Add(leaf);

root.Remove(leaf);

// Recursively display tree

root.Display(1);

// Wait for user

Console.ReadKey();

}

}

}

1. Put a break point on line 17. Step through the code and understand how the tree is built.

**Decorator Pattern**

Frequency of Use: 3/5 Medium

Definition: (also known as Wrapper, an alternative naming shared with the Adapter pattern) is a design pattern that allows behavior to be added to an individual object, either statically or dynamically, without affecting the behavior of other objects from the same class. The decorator pattern is often useful for adhering to the Single Responsibility Principle, as it allows functionality to be divided between classes with unique areas of concern.

1. Go to the Decorator pattern project under the StructuralPatterns folder.
2. View the code on the Car class.
3. View the CarOptionDecorator class.
4. View the OptionDecorator classes.
5. Put a breakpoint in Program.cs on line 18 and step through the code. Understand how the additional Decorator classes are able to be added on and “upgrades” are added to the car without altering the Car class.

**Facade Pattern**

Frequency of Use: 5/5 High

Definition: A facade is an object that provides a simplified interface to a larger body of code, such as a class library. A facade can

* make a software library easier to use, understand, and test, since the facade has convenient methods for common tasks,
* make the library more readable, for the same reason,
* reduce dependencies of outside code on the inner workings of a library, since most code uses the facade, thus allowing more flexibility in developing the system,
* wrap a poorly designed collection of APIs with a single well-designed API.

1. Go to the Facade pattern project under the StructuralPatterns folder.
2. View the code files in the Controllers folder.
3. View the code in the JoggingFacade and Program file:
4. Add a breakpoint to Line 14 of the Program.cs file. Compare the differences in the code paths on the JogWithoutUsingFacade vs. the JogWithFacade path.

**Flyweight Pattern**

Frequency of Use: 1/5 Very Low

Definition: A flyweight is an object that minimizes memory usage by sharing as much data as possible with other similar objects; it is a way to use objects in large numbers when a simple repeated representation would use an unacceptable amount of memory. Often some parts of the object state can be shared, and it is common practice to hold them in external data structures and pass them to the objects temporarily when they are used.

1. Go to the Adapter pattern project under the StructuralPatterns folder.
2. View the code in the Flyweight file.
3. View the code in the ConcreteFlyweight, UnsharedConcreteFlyweight, and then the Flyweight Factory files.
4. Go to Program.cs and add a breakpoint on line 21 and step through the program.

**Proxy Pattern**

Frequency of Use: 4/5 Medium High

Definition: A proxy, in its most general form, is a class functioning as an interface to something else. The proxy could interface to anything: a network connection, a large object in memory, a file, or some other resource that is expensive or impossible to duplicate. In short, a proxy is a wrapper or agent object that is being called by the client to access the real serving object behind the scenes. Use of the proxy can simply be forwarding to the real object or can provide additional logic. In the proxy, extra functionality can be provided, for example caching when operations on the real object are resource intensive, or checking preconditions before operations on the real object are invoked. For the client, usage of a proxy object is like using the real object, because both implement the same interface.

1. Go to the Proxy project under the StructuralPatterns folder.
2. View the code in IMath.
3. View the code in Math.cs.
4. Add this code to the MathProxy:

namespace Proxy

{

/// <summary>

/// The 'Proxy Object' class

/// </summary>

public class MathProxy : IMath

{

private Math \_math = new Math();

public double Add(double x, double y)

{

return \_math.Add(x, y);

}

public double Sub(double x, double y)

{

return \_math.Sub(x, y);

}

public double Mul(double x, double y)

{

return \_math.Mul(x, y);

}

public double Div(double x, double y)

{

return \_math.Div(x, y);

}

}

}

1. Add this code to the Program.cs file:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Proxy

{

class Program

{

/// <summary>

/// Entry point into console application.

/// </summary>

static void Main()

{

// Create math proxy

MathProxy proxy = new MathProxy();

// Do the math

Console.WriteLine("4 + 2 = " + proxy.Add(4, 2));

Console.WriteLine("4 - 2 = " + proxy.Sub(4, 2));

Console.WriteLine("4 \* 2 = " + proxy.Mul(4, 2));

Console.WriteLine("4 / 2 = " + proxy.Div(4, 2));

// Wait for user

Console.ReadKey();

}

}

}

1. Drop a breakpoint in Program.cs on line 21 and step through the code to understand the math logic.

## Task: Behavioral Patterns

Behavioral patterns are concerned with the assignment of responsibilities between objects, or, encapsulating behavior in an object and delegating requests to it. These patterns increase flexibility in carrying out this communication.

Try to remember: People are **Behavioral**. People are **Responsible** (Chain of Resp), **Commanders** (Command), **Interpreters**, **Iterators**, **Mediators**, **Observers**, **Strategists** (Strategy), and **Visitors**. People tend to like keeping **Mementos**, live in **States**, and create **Templates**.

* 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

**Chain of Responsibility Pattern**

Frequency of Use: 2/5 Low

Definition: In object-oriented design, the chain-of-responsibility pattern is a design pattern consisting of a source of command objects and a series of processing objects. Each processing object contains logic that defines the types of command objects that it can handle; the rest are passed to the next processing object in the chain. A mechanism also exists for adding new processing objects to the end of this chain.

1. Go to the ChainResp pattern project under the BehavioralPatterns folder.
2. View the code in the Approver and Purchase classes.
3. View the code in the Director, President, and Vice President classes.
4. Go to Program.cs. Add a breakpoint to line 21. Understand how approvers and successors are working in this app.

**Command Pattern**

Frequency of Use: 4/5 Medium High

Definition: An object is used to encapsulate all information needed to perform an action or trigger an event later. This information includes the method name, the object that owns the method and values for the method parameters.

1. Go to the Command pattern project.
2. View the code in the Reciever, Command, and Invoker classes.
3. View code in the Concrete command class.
4. Add the following to Program.cs in the Main method:

// Create receiver, command, and invoker

Receiver receiver = new Receiver();

Command command = new ConcreteCommand(receiver);

Invoker invoker = new Invoker();

// Set and execute command

invoker.SetCommand(command);

invoker.ExecuteCommand();

Console.ReadKey();

1. Step through the code and put a breakpoint on line 17. Understand how the command is setup and invoked later.

**Interpreter Pattern**

Frequency of Use: 1/5 Very Low

Definition: A design pattern that specifies how to evaluate sentences in a language. The basic idea is to have a class for each symbol (terminal or nonterminal) in a specialized computer language. The syntax tree of a sentence in the language is an instance of the composite pattern and is used to evaluate (interpret) the sentence for a client.

1. Go to the Interpreter pattern project.
2. View the code in all files.
3. Add a breakpoint to line 17 in Program.cs and step through the code.
4. Change the Roman Numerals around in line 16 to see it converted to numbers.

**Iterator Pattern**

Frequency of Use: 5/5 High

Definition: An iterator is used to traverse a container and access the container's elements. The iterator pattern decouples algorithms from containers; in some cases, algorithms are necessarily container-specific and thus cannot be decoupled.

1. Go to the Iterator pattern project.
2. View the code in Container.cs and read through the comments to understand how we explicitly implement iterators.
3. Put a breakpoint on line 8 in Program.cs and step through the program and read the comments on the Program.cs file.

**Mediator Pattern**

Frequency of Use: 2/5 Low

Definition: With the mediator pattern, communication between objects is encapsulated within a mediator object. Objects no longer communicate directly with each other, but instead communicate through the mediator. This reduces the dependencies between communicating objects, thereby reducing coupling.

1. Go to the Mediator pattern project
2. View the code in the AbstractChatroom and Participant files first.
3. Look at the Beatle, Chatroom, and NonBeatle files.
4. Go to Program.cs and add a breakpoint on line 23 and step through the project.

**Memento Pattern**

Frequency of Use: 1/5 Very Low

Definition: Provides the ability to restore an object to its previous state (undo via rollback). The memento pattern is implemented with three objects: the originator, a caretaker and a memento. The originator is some object that has an internal state. The caretaker is going to do something to the originator, but wants to be able to undo the change. The caretaker first asks the originator for a memento object. Then it does whatever operation (or sequence of operations) it was going to do. To roll back to the state before the operations, it returns the memento object to the originator. The memento object itself is an opaque object (one which the caretaker cannot, or should not, change). When using this pattern, care should be taken if the originator may change other objects or resources - the memento pattern operates on a single object.

1. Go to the Memento pattern project
2. View the code in all the files.
3. Go to Program.cs and drop a breakpoint on line 17 and see the state get restored.

**Observer Pattern**

Frequency of Use: 5/5 High

Definition: An object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.

* It is mainly used to implement distributed event handling systems, in "event driven" software. Most modern languages such as Java and C# have built in "event" constructs which implement the observer pattern components, for easy programming and short code.
* The observer pattern is also a key part in the familiar model–view–controller (MVC) architectural pattern. The observer pattern is implemented in numerous programming libraries and systems, including almost all GUI toolkits.

1. Go to the Observer pattern project
2. View the code in the Contracts and Providers folders.
3. Add this code to the MessageManager:

using System;

using Observer.Contracts;

namespace Observer

{

public class MessageManager : IMessageManager

{

public MessageManager() {}

//Declare event

//Use built-in EventHandler delegate

public event EventHandler<NewMesageEventArgs> SendNewMessageEvent;

public string Sender { get; set; }

public string Recipient { get; set; }

public string Subject { get; set; }

public string Body { get; set; }

public void SendMessage(string sender, string recepient, string subject, string message)

{

Sender = sender;

Recipient = recepient;

Subject = subject;

Body = message;

//Construct object that contains email information that we pass to subscribers

var e = new NewMesageEventArgs(sender, recepient, subject, message);

//Raise the event to notify subscribers

OnSendNewMessage(e);

}

protected virtual void OnSendNewMessage(NewMesageEventArgs e)

{

// Always ensure that someone is listening before raising event

if (SendNewMessageEvent != null)

SendNewMessageEvent(this, e);

}

}

}

1. Add this code to the NewMessageEventArgs:

using System;

namespace Observer

{

public sealed class NewMesageEventArgs : EventArgs

{

public NewMesageEventArgs(string sender, string recepient, string subject, string message)

{

Sender = sender;

Recipient = recepient;

Subject = subject;

Message = message;

}

public string Sender { get; set; }

public string Recipient { get; set; }

public string Subject { get; set; }

public string Message { get; set; }

}

}

1. Add this code to the Program.cs Main method:

var \_observerClient = new ObserverClient();

\_observerClient.RunClient();

Console.WriteLine("Press any key to continue");

Console.ReadLine();

1. Add a breakpoint to the Program.cs on line 14 and step through the code and understand the event handling.

**State Pattern**

Frequency of Use: 3/5 Medium

Definition: Implements a state machine in an object-oriented way. With the state pattern, a state machine is implemented by implementing each individual state as a derived class of the state pattern interface, and implementing state transitions by invoking methods defined by the pattern's superclass.

The State pattern allows an object to change its behavior when its internal state changes. This pattern can be observed in a vending machine. Vending machines have states based on the inventory, amount of currency deposited, the ability to make change, the item selected, etc. When currency is deposited, and a selection is made, a vending machine will either deliver a product and no change, deliver a product and change, deliver no product due to insufficient currency on deposit, or deliver no product due to inventory depletion.

1. Go to the State pattern project
2. View the code in State, then Account.
3. View the code in Gold, Red and Silver State.
4. Add a breakpoint to Program.cs on line 20. This real-world code demonstrates the State pattern which allows an Account to behave differently depending on its balance. The difference in behavior is delegated to State objects called RedState, SilverState and GoldState. These states represent overdrawn accounts, starter accounts, and accounts in good standing.

**Strategy Pattern**

Frequency of Use: 4/5 Medium High

Definition: Strategy pattern (also known as the policy pattern) is a behavioral software design pattern that enables selecting an algorithm at runtime. The strategy pattern:

* defines a family of algorithms,
* encapsulates each algorithm, and
* makes the algorithms interchangeable within that family.

Strategy lets the algorithm vary independently from clients that use it.

1. Go to the Strategy pattern project
2. View the code in Contracts, Selector, and Strategies.
3. Go to the StrategyClient Project now, which references the Strategy project.
4. Go to Program.cs and put a breakpoint on line 17, and step through the code.

**Template Method Pattern**

Frequency of Use: 3/5 Medium

Definition: Defines the program skeleton of an algorithm in an operation, deferring some steps to subclasses. It lets one redefine certain steps of an algorithm without changing the algorithm's structure.

1. Go to the Template pattern project
2. View the code in all the files.
3. Go to TemplateMethodClient and add a breakpoint to line 9.
4. Step through the code.

**Visitor Pattern**

Frequency of Use: 1/5 Very Low

Definition: The visitor design pattern is a way of separating an algorithm from an object structure on which it operates. A practical result of this separation is the ability to add new operations to existent object structures without modifying the structures. It is one way to follow the open/closed principle.

The visitor allows adding new virtual functions to a family of classes, without modifying the classes. Instead, a visitor class is created that implements all of the appropriate specializations of the virtual function. The visitor takes the instance reference as input, and implements the goal through double dispatch.

1. Go to the Visitor pattern project
2. View the code.
3. Add a breakpoint on line 17 of Program.cs

**GoF Code references:**

<http://www.dofactory.com>

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