.NET Framework 4.6 & C# 6.0

Lesson 13

Design Principles and Patterns

Concept of Design Pattern

Design Pattern is a solution to a problem in a context.

Pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution."

Design Patterns are "reusable solutions to recurring problems that we encounter during software development."

What is a Design Pattern?
"A pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem in such a way that you can use this solution a million times over, without ever using it the same way twice."
Patterns can be applied to many areas of human endeavor, including software development.

Why Design Patterns?

Rationale behind using Design Patterns

- Patterns enable programmers to "…recognize a problem and immediately determine the solution without having to stop and analyze the problem first."
- Provide reusable solutions
- Enhance productivity

Why Design Patterns?
Designing object-oriented code is hard, and designing reusable object-oriented software is even harder.
Patterns enable programmers to "…recognize a problem and immediately determine the solution without having to stop and analyze the problem first."
Well structured object-oriented systems have recurring patterns of classes and objects.
The patterns provide a framework for communicating complexities of OO design at a high level of abstraction. Bottom line is productivity.
Experienced designers reuse solutions that have worked in the past.
Design Patterns enable large-scale reuse of software architecture and also help document systems.
Patterns explicitly capture expert knowledge and design tradeoffs and make it more widely available.
Patterns help improve a developer communication as they form a common vocabulary

History of Design Patterns

Chronological Order of Events

- 1979 – Christopher Alexander pens The Timeless Way of Building.
  - Building Towns for Dummies
  - It had nothing to do with software
- 1987 – Cunningham and Beck used Alexander's ideas to develop a small pattern language for smalltalk.
- 1990 – The Gang Of Four (Gamma, Helm, Johnson, and Vlissides) begin work on compiling a catalog of design patterns.
- 1994 – The GOF publish the first book on Design Patterns.

Note:
The history of GOF Design Patterns is listed on the slide.

Design Patterns are NOT

- Data structures that can be included in classes and reused as is (i.e. linked lists, hash tables, etc.)
- Complex domain-specific design solutions for the entire application

Instead, they are:

- Proven solutions to a general design problem in a particular context which can be customized to suit our needs

Features of JSP:
Do not ever think that the class structure given by Design Patterns can be used as is unlike Data structures (namely, link lists, hash tables, etc).
They do not provide a domain specific design solution for an entire application.
Instead they provide a proven solution to a general design problem in a particular context (not domain) which can be customized to suit our needs.
In short, from Design Patterns we do not get “code reuse” rather we get “experience re-use”.

Classification of Design Patterns

Broad level Categories of Design Patterns

Design Patterns can be broadly classified as:

- Fundamental patterns
- Creational patterns
- Structural patterns
- Behavioral Patterns

Classification of GOF Design Patterns:

The Design Patterns can be broadly classified as :

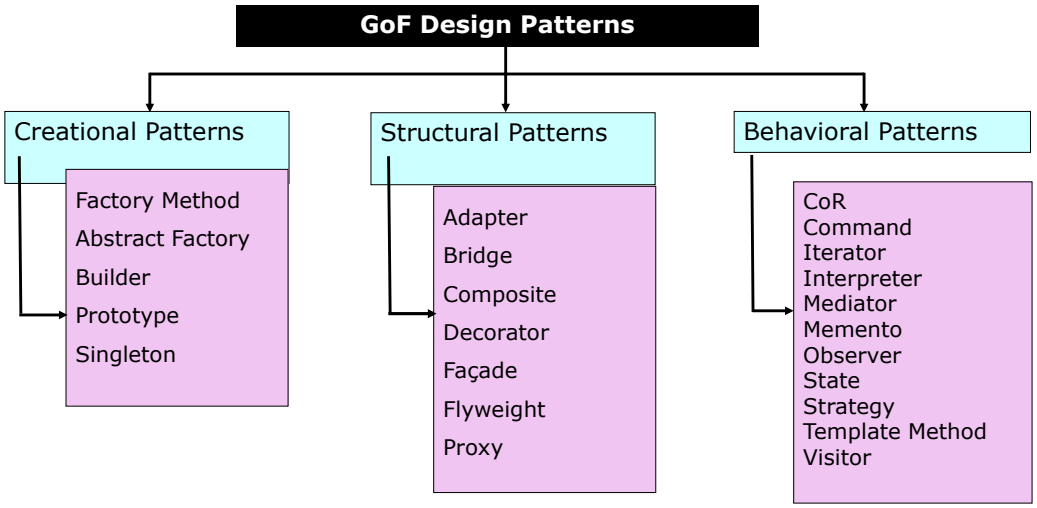
Fundamental Patterns: They are the building blocks for the other three categories of Design Patterns.

Creational Patterns: They deal with creation, initializing, and configuring classes and objects.

Structural Patterns: They facilitate decoupling interface and implementation of classes and objects.

Behavioral Patterns: They take care of dynamic interactions among societies of classes and objects. They also give guidelines on how to distribute responsibilities amongst the classes.

Further Classification of Design Patterns



Note:
There are 23 GOF Design Patterns.
They have been classified as shown on the slide. Each of the Design Patterns will be explained in detailed in the subsequent lessons.

Categories of Design Patterns

Scope/Purpose	Creational	Structural	Behavioral
Class	Factory Method	Adapter	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Categories of Design Patterns:
The Design Patterns are categorized as Class based or Object-based.
Class based Design Patterns uses “inheritance” as the basic principle whereas the object based patterns use “composition”.
One of the design principles says, “Favor Composition over Inheritance”.
As seen from the slide, “composition” is being favored by Design Patterns as well as most of the Design Patterns are “Object-based”.

Advantages of Design Patterns

- Design patterns allows a designer to be more productive and the resulting design to be more flexible and reusable.
- Design patterns make communication between designers more efficient

Software professionals can immediately picture the high-level design in their heads when they refer the name of the pattern used to solve a particular issue when discussing system design.

Drawbacks of Design Patterns

Listed below are some of the drawbacks of design patterns:

- Patterns do not allow direct code reuse.
- Patterns are deceptively simple.
- Design might result into Pattern overload.
- Patterns are validated by experience and discussion rather than by automated testing.

Note:
Besides the drawbacks mentioned in the slide, Integrating patterns into a software development process is a human-intensive task.

What is a Fundamental Design Pattern?

Fundamental Patterns are fundamental in the sense that they are widely used by other patterns or are frequently used in a large number of programs.

Note:
The fundamental design patterns are the building blocks of the other design patterns.

Different Fundamental Patterns

Fundamental patterns are further classified as:

- Delegation Pattern
- Interface Pattern
- Abstract Superclass
- Interface and abstract class
- Immutable Pattern
- Marker Interface

Fundamental Design Patterns:
Delegation Pattern: It is a way of extending and reusing a class using composition rather than inheritance.
Interface Pattern: This pattern facilitates the design principle – “Program to Interface rather than Implementation”.
Abstract Superclass: It ensures consistent behavior of conceptually related classes by giving them a common abstract superclass.
Interface and abstract class: It is a combination of Interface and Abstract superclass Patterns.
Immutable Pattern: This pattern prevents the object from changing its state information after it is constructed thereby eliminating the issues related to concurrent access of the object.
Marker Interface: It is used to mark an object to be part of a particular group thereby allowing other objects to treat them in a uniform way.

Usage of Interface Pattern

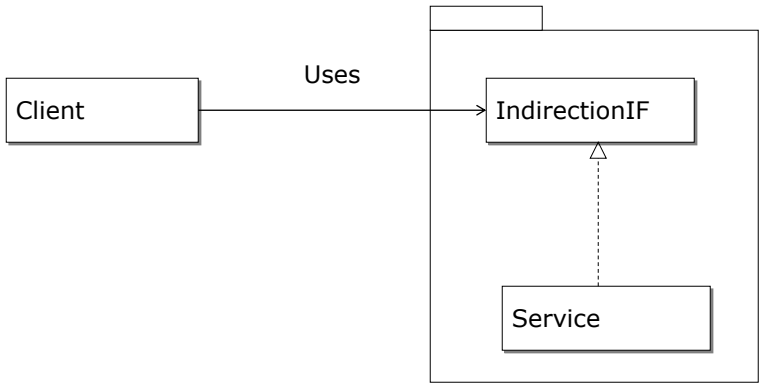
Interfaces are "more abstract" than classes since they do not say anything at all about representation or code. All they do is describe public operations.

You can keep a class that uses data and services provided by instances of other classes independent of those classes by having it access those instances through an interface.

Interface Pattern:
A common problem is that you want an object to be able to use another object that provides a service without having to assume the class of the other object. The usual solution to the problem is to have the object assume that the object it uses implements a particular interface, rather than it being an instance of any particular class.
An interface encapsulates a coherent set of services and attributes, without explicitly binding this functionality to that of any particular object or code.
A class which uses other classes such as data and services should be built in such a way that the class should be independent of the kind of data and service objects it is going to deal with.

When to use:
An object relies on another object for data or services. If the object must assume that the other object upon which it relies belongs to a particular class, then the reusability of the object's class would be compromised.
You want to vary the kind of object used by other objects for a particular purpose without making the object dependent on any class other than its own.
Requirement is for generic functionality.

Structure of Interface Pattern



Structure of Interface Pattern:
Following are the roles played by these classes and interfaces:
Client: The Client class uses classes that implement the IndirectionIF interface.

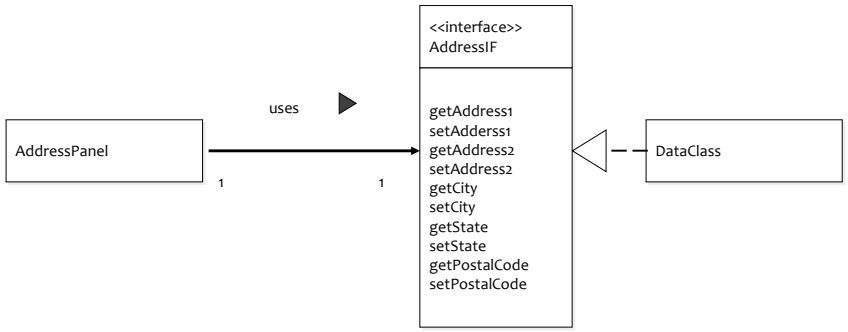
IndirectionIF: The IndirectionIF interface provides indirection that keeps the Client class independent of the class that is playing the Service role. Interfaces in this role are generally public.

Service: Classes in this role provide a service to classes in the client role. Classes in this role are ideally private to their package. Making Service classes private forces classes outside of their package to go through the interface. However, it is common to have implementations of an interface that are in different packages.

Example

Street Address is a very common entity that is required for any application that needs to maintain address of employee/customer/vendor.

To have uniform structure for this address throughout the application, the Interface Pattern can be used as shown below:



Interface Pattern:
To avoid classes having to depend on other classes because of a uses/usedby relationship, make the usage indirect through an interface.
For example, suppose you are writing an application to purchase goods for a business. Your program will need to be informed of such entities as vendors, freight companies, receiving locations, and billing locations. One thing these have in common is that they all have a street address. These street addresses will be displayed in different parts of the user interface. You will want to have a class for displaying and editing street addresses so that you can reuse it wherever there is an address in the user interface. We will call that class AddressPanel.
You want AddressPanel objects to be able to get and set address information in a separate data object. This raises the question of what instances of the AddressPanel class can assume about the class of the data objects that they will use. Clearly, you will use different classes to represent vendors, freight companies, and the like. If you program in a language that supports multiple inheritance, like C++, you can arrange for the data objects that instances of AddressPanel use to inherit from an address class in addition to the other classes they inherit from. If you program in a language like Java that uses a single inheritance object model, then you must explore other solutions.
You can solve the problem by creating an address interface. Instances of the AddressPanel class would then simply require data objects that implement the address interface. They would then be able to call the accessor methods of that object to get and set its address information. Using the indirection that the interface provides, instances of the AddressPanel class are able to call the methods of the data object without having to be aware of what class it belongs to. Here is a class diagram showing these relationships.
Here in this example we have seen that the AddressPanel class is independent or does not have to specifically assume the data object it is going to deal with. In turn, it will work upon the data object with the help of the AddressIF interface which is actually implemented by the data classes.

Usage of Abstract Superclass

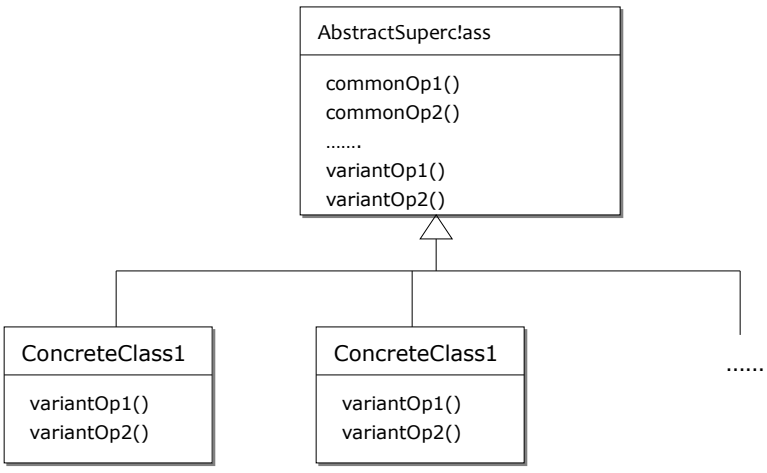
Abstract superclass ensures consistent behavior of conceptually related classes by giving them a common abstract superclass.

Forces:

- You want to ensure that logic common to the related classes is implemented consistently for each class.
- You want to avoid the runtime and maintenance overhead of redundant code.
- You want to make it easy to write related classes.

Abstract Superclass:
The Abstract superclass helps in maintaining consistency in behavior across the sub classes.
Whatever is common is separated out in an abstract class and all the related classes are made to inherit from this abstract class so that they inherit this common behavior.

Structure of Abstract Superclass



Abstract Superclass:
Organize the common behavior of related classes into an abstract superclass.
To the extent possible, organize variant behavior into methods with common signatures. Declare the abstract superclass to have abstract methods with these common signatures. The above figure shows this organization.
Following are the roles that classes play in the Abstract Superc!ass pattern:

AbstractSuperclass:
A class in this role is an abstract superclass that encapsulates the common logic for related classes. The related classes extend this class so they can inherit methods from it. Methods whose signature and logic are common to the related classes are put into the superclass so their logic can be inherited by the related classes that extend the superclass. Methods with different logic but the same signature are declared in the abstract class as abstract methods, ensuring that each concrete subclass has a method with those signatures.
ConcreteClassl, ConcreteClass2, and so on:
A class in this role is a concrete class whose logic and purpose is related to other concrete classes. Methods common to these related classes are refactored into the abstract superclass. Common logic that is not encapsulated in common methods is refactored into common methods.

Usage of Interface and Abstract Class

You need to keep client classes independent of classes that implement a behavior and ensure consistency of behavior between the behaviorimplementing classes.

Interface and Abstract Class:

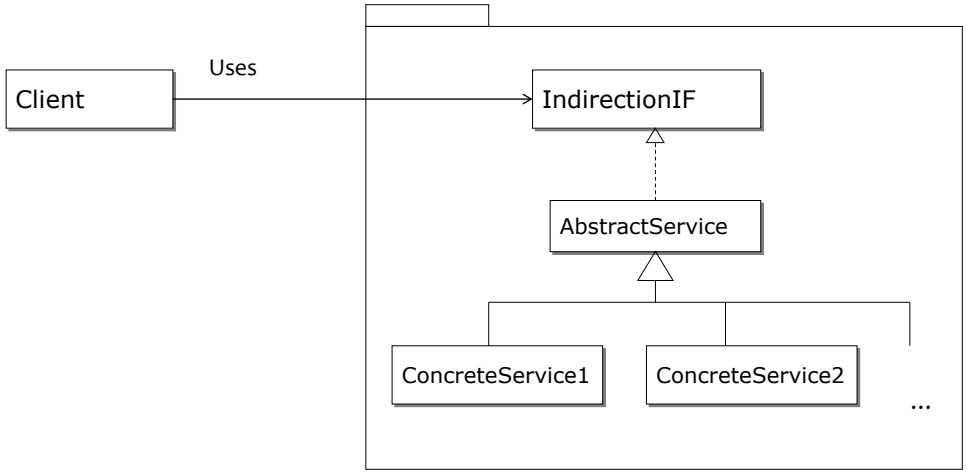
You need to keep client classes independent of classes that implement a behavior and ensure consistency of behavior between the behaviorimplementing classes. Do not choose between using an interface and an abstract_ class. Have the classes implement an interface and extend an abstract class.

Forces:

Using the Interface pattern, Java interfaces can be used to hide the specific class that implements a behavior from the clients of the class.

Organizing classes that provide related behaviors with a common superclass helps to ensure consistency of implementation. Through reuse, it may also reduce the effort required for implementation.

Structure of Interface and Abstract Class



Interface and Abstract Class:

The main intention here is to hide the classes that implement some behavior making the classes private to the package by having them implement a public interface.

- By isolating client from service through Interface
- By isolating Abstraction from implementation through Abstract class

What is a Creational Design Pattern?

- Design patterns that deal with object creation mechanisms.
- Help to create objects in a manner suitable to the situation.
- Provide guidance on how to create objects when their creation requires making decisions.

Introduction to Creational Patterns:
Creational Patterns provide guidelines on creation, configuration, and initialization for objects.
“Decisions typically involve dynamically deciding which class to instantiate or which objects an object will delegate responsibility to.”

Different Creational Design Patterns

Creational Design Patterns are further classified as:

- Factory method
- Singleton

Different Creational Patterns:

Creational Design Patterns are further classified as:

Factory Method: It creates objects without specifying the exact object to create.

Abstract Factory: It groups object factories that have a common theme.

Prototype: It creates objects by cloning an existing object.

Builder: It constructs complex objects by separating construction and representation.

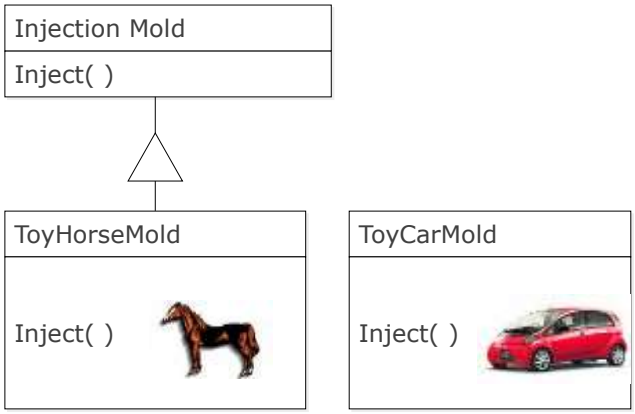
Singleton: It restricts object creation for a class to only one instance.

Usage of Factory Method Pattern

- Helps to model an interface for creating an object
- Allows subclasses to decide which class to instantiate
- Helps to instantiate the appropriate subclass by creating the right object from a group of related classes
- Promotes loose coupling.

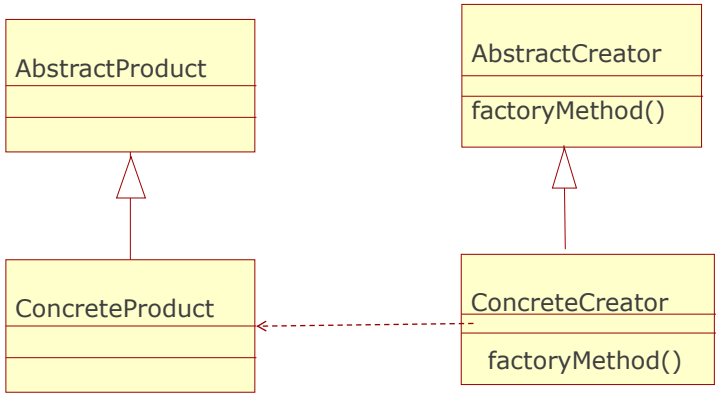
Factory Method:
You can define an interface for creating an object, however, let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
It is also called as a Factory Pattern since it is responsible for “manufacturing” an Object.
The Factory Method pattern is an object-oriented design pattern. Like other creational patterns, it deals with the problem of creating objects (products) without specifying the exact class of object that will be created. Factory Method handles this problem by defining a separate method for creating the objects, which can then be overridden by subclasses to specify the derived type of the product that will be created.
They promote loose coupling by eliminating the need to bind application specific classes into the code.

Example of Factory Method Pattern



Example of Factory Method Pattern:
The Factory Method defines an interface for creating objects, but lets subclasses decide which classes to instantiate.
Injection molding presses demonstrate this pattern. Manufacturers of plastic toys process plastic molding powder, and inject the plastic into molds of the desired shapes. The class of toy (car, action figure, etc.) is determined by the mold.

Structure of Factory Method Pattern



Structure of Factory Method Pattern:
Structure Elements:
AbstractProduct: It defines the interface of objects that the factory method creates.
ConcreteProduct: It implements/extends the AbstractProduct interface/class.
AbstractCreator:

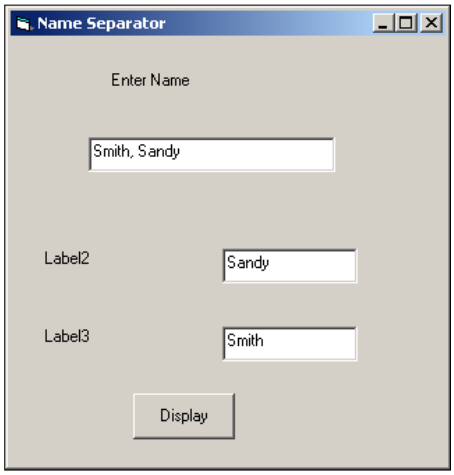
- It declares the factory method, which returns an object of type AbstractProduct.
- It may also define a default implementation of the factory method that returns a default ConcreteProduct object.
- It may call the factory method to create a Product object.

ConcreteCreator: It overrides the factory method to return an instance of a ConcreteProduct.

Example of Factory Method Pattern

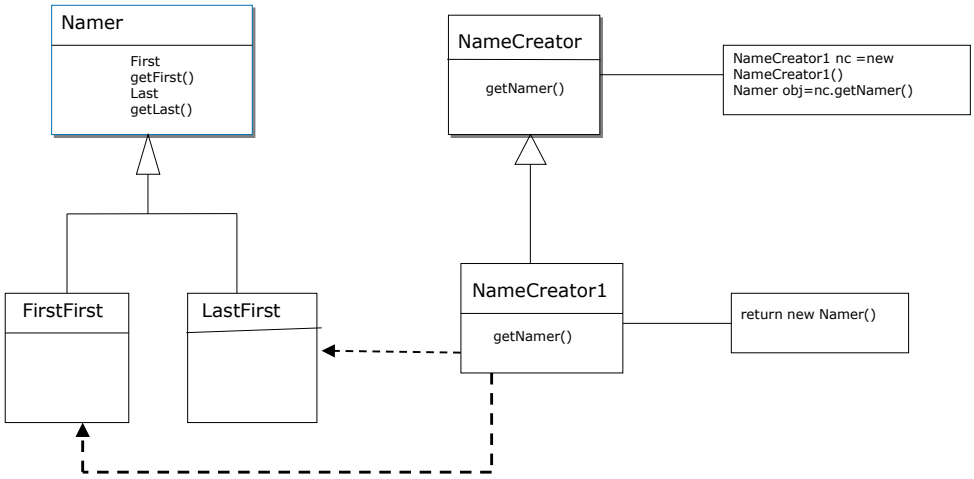
Suppose we have an entry form and we want to allow the user to enter his/her name either as "first name last name" or as "last name, first name".

Name order is always decided by whether comma is in between the last and first name.



What will be your approach?

Factory Method Pattern Solution



Factory Method Pattern Solution:
Product: Namer class
Concrete Product: FirstFirst and LastFirst classes
Creator: NameCreator class
Concrete Creator: NameCreator1 class

Pros and Cons of Factory Method

Pros:

- Loose Coupling
  - Factory Method eliminates the need to bind application classes into client code.
- Object Extension
  - It enables classes to provide an extended version of an object.
- Appropriate Instantiation
  - Right object is created from a set of related classes.

Pros and Cons of Factory Method:
Pros:
Loose Coupling: Factory Method eliminates the needs to bind application classes into client code. The code deals with any classes that implement a certain interface. Application becomes more flexible and reusable.
Object Extension: It enables the subclasses to provide an extended version of an object.
Appropriate Instantiation: It helps to instantiate the appropriate subclass by creating the right object from a group of related classes.

Pros and Cons of Factory Method

Cons:

- Two major varieties
  - The creator superclass may or may not implement the factory method
  - In any case, the superclass needs to be subclassed to create a concrete product
- Parameterized factory methods
  - Multiple kinds of products can be created by passing parameters to factory methods
  - There is no standardization on this aspect

Pros and Cons of Factory Method:
Cons:
Two major varieties: The two main variations of the Factory Method pattern are: (1) the case when the Creator class is an abstract class and does not provide an implementation for the factory method it declares, (2) the case when the Creator is a concrete class and provides a default implementation for the factory method. A potential disadvantage of factory methods is that clients might have to subclass the Creator class just to create a particular ConcreteProduct object.
Parameterized factory methods: Another variation on the pattern lets the factory method create multiple kinds of products. The factory method takes a parameter that identifies the kind of object that has to be created. All objects that the factory method creates share the Product interface. In the Namer example, Namer might support different kinds of naming mechanisms. You pass getNamer an extra parameter to specify the kind of Naming mechanism.

Usage of Singleton Pattern

- The Singleton pattern ensures that only one instance of a class is created
- All objects that use an instance of that class use the same instance

Singleton Pattern:

There are cases in programming where you need to ensure that there can be one and only one instance of a class which is used by the application.

The Singleton pattern ensures that a class has only one instance, and provides a global point of access to it.

It is important for some classes to have exactly one instance.

Although there can be many printers in a system, there should be only one printer spooler.

There should be only one File system and one Window manager.

A digital filter will have one A/D converter.

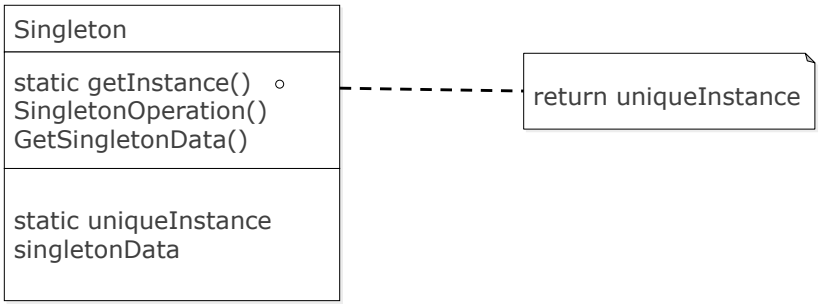
An accounting system will be dedicated to serving one company.

How do we ensure that a class has only one instance and that the instance is easily accessible?

A global variable makes an object accessible. However, it does not keep you from instantiating multiple objects.

A better solution is to make the class itself responsible for keeping track of its sole instance. The class can ensure that no other instance can be created (by intercepting requests to create new objects), and it can provide a way to access the instance. This is the Singleton pattern.

Structure of Singleton Pattern



Structure of Singleton Pattern:
Structure Elements:
Singleton:

It defines a getInstance operation that lets clients access its unique instance.
getInstance is a class operation (i.e. static method)
It may be responsible for creating its own unique instance.
It should have a private constructor so that the client cannot creates its instance using “new” keyword.

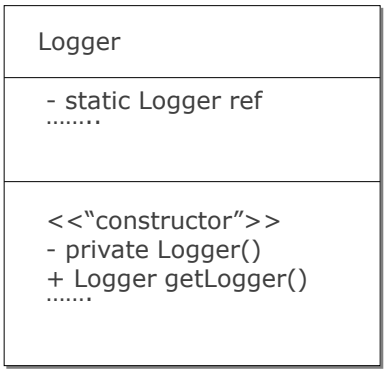
In case of multiple class loaders, the Singleton Implementation will be difficult as each class loader will have one instance of the Singleton class which is as good as having multiple instances of the same class.

Example of Singleton Pattern

We need to implement the logger and log it to some file according to date time. In this case, we cannot have more than one instance of Logger in the application, otherwise the file in which we need to log will be created with every instance.

How can you implement this logger?

Singleton Pattern Solution



Solution:

Singleton: Logger

Logger class needs to be a singleton. This is because application uses the single Logger to log the information. This can be achieved by restricting an application from creating multiple instances, by providing private constructor. To ensure that the instance is created the first time and later the same instance is used by the application, getLogger() method is defined. You also need to provide the static instance variable of Logger class to make it global.

Pros and Cons of Singleton

Pros

- It provides controlled access to unique instance
- It provides reduced namespace

Cons:

- If the Singleton class has subclasses then ensuring unique instance is a challenge
- Ensuring a unique instance at all times is a challenge
- It poses difficulty in unit testing as the Singletons introduce global state into the application

Pros and Cons of Singleton:
Pros:
Controlled access to sole instance: Since the Singleton class encapsulates its sole instance, it can have strict control over how and when clients access it.

Reduced name space: The Singleton pattern is an improvement over global variables. It avoids polluting the name space with global variables that store sole instances.

Cons:
Subclassing the Singleton class: The main issue is not in defining the subclass but ensuring its unique instance so that clients will be able to use it. In other words, the variable that refers to the singleton instance must get initialized with an instance of the subclass.
Ensuring a unique instance: In case of multiple class loaders, it is difficult to ensure single instance.
Hard to test: Singleton introduces global state into an application and hence it is difficult to unit test it. Also the hidden dependency of users on a singleton makes testing very difficult as there is no way to mock out or inject a test instance of the singleton. Also the state of the singleton affects the execution of a suite of tests such

that they are not properly isolated from each other.

Case Study on Singleton

We need to design a workflow application for the supply chain division of a manufacturing company by using SQL Server as backend.

The workflow configuration is maintained in XML file stored on file system of server. All the business logic classes will access this configuration, so a single point of access is required.

Issue: Parsing the workflow configuration file every time that it is needed is a costly affair.

What is the solution?

Solution:
The configuration file can be loaded only once when the application is deployed. This can be implemented by using the Singleton Pattern. You can create a class that will load the XML file and make use of the configuration settings in the XML file. You will have to ensure that only one instance of this class (workflow configuration class) is created using Single ton Pattern. All the business logic classes will then need to make use of this single instance.

Adapter & Factory Method Pattern

Adapter

The *adapter* pattern, a common mechanism of bridging systems and platforms, is implemented in a variety of ways in the .NET framework. One of the most prevalent examples of this in .NET are Runtime Callable Wrappers, or RCW's. RCW's, generated with the tlbimp.exe program, provide adapters that let .NET managed code easily call into legacy COM code via a .NET API.

Factory Method

The *factory method* pattern is probably one of the most well-known patterns. It is implemented quite commonly throughout the .NET framework, particularly on primitive times, but also on many others. An excellent example of this pattern in the framework is the Convert class, which provides a host of methods to create common primitives from other common primitives.

Additionally, another pervasive form of this pattern are the .Parse() and .TryParse() methods found on many primitive and basic types.

Iterator Pattern

Iterator

- The *Iterator* pattern is implemented via a couple interfaces and some language constructs, like foreach and the yeild keyword in C#.
- The Ienumerable interface and its generic counterpart are implemented by dozens of collections in the .NET framework, allowing easy, dynamic iteration of a very wide variety of data

```
IEnumerable<T>
IEnumerator<T>

foreach(var thing in someEnumerable)
{
  //
}
```

Yield

The yeild keyword in C# allows the true form of an *iterator* to be realized, only incurring the cost of processing an iteration through a loop when that iteration is demanded:

```
IEnumerable<string> TokenizeMe(string complexString)
{
  string[] tokens = complexString.Split(' ');
  foreach (string token in toekens)
  {
    yield return token;
  }
}
```

Builder Pattern

- The *Builder* pattern is implemented a few times in the .NET framework.
- A couple of note are the connection string builders.
- Connection strings can be a picky thing, and constructing them dynamically at runtime can sometimes be a pain. Connection String Builder classes demonstrate the builder pattern ideally:

```
string connectionString = new SqlConnectionStringBuilder
{
   DataSource = "localhost",
   InitialCatalog = "MyDatabase",
   IntegratedSecurity = true,
   Pooling = false
}.ConnectionString;
```


Other classes throughout the .NET framework, such as UriBuilder, also implement the builder pattern.

Observer Pattern

- The *observer* pattern is a common pattern that allows one class to watch events of another.
- As of .NET 4, this pattern is supported in two ways: via
  - language-integrated events (tightly coupled observers)
  - IObservable/IObserver interfaces (loosely coupled events)
- Classic language events make use of *delegates*, or strongly-typed function pointers, to track event callbacks in *event* properties.
- An event, when triggered, will execute each of the tracked callbacks in sequence. Events like this are used pervasively throughout the .NET framework.

```
public class EventProvider
{
  public event EventHandler SomeEvent;

  protected virtual void OnSomeEvent(EventArgs args)
  {
    if (SomeEvent != null)
    {
      SomeEvent(this, args); // Trigger event
    }
  }
}

public class EventConsumer
{
  public EventConsumer(EventProvider provider)
  {
    provider.SomeEvent += someEventHandler; // Register as observer of event
  }

  private void someEventHandler(EventArgs args)
  {
    // handle event
  }
}
```


New with the .NET 4 framework are loosely coupled events. These are accomplished by implementing the IObservable<out T> and IObserver<in T> interfaces, which more directly support the original Observer design pattern. While not directly implemented by any .NET framework types that I am aware of, the core infrastructure for the pattern is an integral part of .NET 4.

Decorator Pattern

- The *decorator* pattern is a way of providing alternative representations, or forms, of behavior through a single base type.
- Quite often, a common set of functionality is required, but the actual implementation of that functionality needs to change.
- An excellent example of this in the .NET framework is the Stream class and its derivatives.
- All streams in .NET provide the same basic functionality, however each stream functions differently.

Stream

- MemoryStream
- BufferedStream
- FileStream
  - IsolatedStorageFileStream
- PipeStream
  - AnonymousPipeClientStream
  - AnonymousPipeServerStream
  - NamedPipeClientStream
  - NamedPipeServerStream
- CryptoStream
- GZipStream

Many, many other design patterns are used within the .NET framework. Almost every aspect of .NET, from language to framework to fundamental runtime concepts, are based on common design patterns. Significant portions of the .NET framework, such as ASP.NET, are in and of themselves patterns. Take, for example, the ASP.NET MVC framework, which is an implementation of the web variant of MVC, or *Model-View-Controller*. The WPF and Silverlight UI frameworks directly support a pattern called MVVM, or *Model-View-ViewModel*. The ASP.NET pipeline itself is a collection of patterns, including *intercepting filter*, *page controller*, *router*, etc. Finally, one of the most commonly used patterns, *composition*, is used so extensively in the .NET framework that it is probably one of the most fundamental patterns of the entire framework.

Summary

In this lesson, you have learnt:

- Concept of Design Pattern
- Rationale behind using Design Patterns
- Classification of Design Patterns
- Categories of Design Patterns
- Drawbacks of Design Patterns

Review – Questions

1. Design pattern is a solution to ___ within a particular ___.
2. Name different types of GOF Design Patterns.
3. The Design Patterns are categorized with ___ and ___ scope.

People matter, results count.

About Capgemini

With more than 190,000 people, Capgemini is present in over 40 countries and celebrates its 50th Anniversary year in 2017. A global leader in consulting, technology and outsourcing services, the Group reported 2016 global revenues of EUR 12.5 billion. Together with its clients, Capgemini creates and delivers business, technology and digital solutions that fit their needs, enabling them to achieve innovation and competitiveness. A deeply multicultural organization, Capgemini has developed its own way of working, the Collaborative Business Experience™, and draws on Rightshore®, its worldwide delivery model.

Learn more about us at

www.capgemini.com