Design Principles and Patterns

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."

Broad level Categories of Design Patterns

Design Patterns can be broadly classified as:

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

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.

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.

Fundamental Design Pattern

Fundamental Patterns are fundamental in the sense that they are widely used by other patterns or are frequently used in many programs.

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

Summary of Design Principles & Patterns

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.

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.

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 behavior implementing classes. Do not choose between using an interface and an abstract class. Have the classes implement an interface and extend an abstract class.

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.

- Factory method
- Singleton
- Abstract Factory
- Prototype
- Builder

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.

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.

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

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.

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.

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.

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.

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 functionalities 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.