### ¿Que es inyección de dependencias?

Basicamente es quitarle la responsabilidad a nuestra clase principal de crear instancias a sus dependencias. En pocas palabras, en vez de que nuestra clase haga una instancia internamente a otra clase que depende, mejor le pasamos la clase que va a implementar como un parámetro adicional, de esta manera evitamos la dependencia directa. ¿Se entendió?, bueno ... espero que con el ejemplo se entienda mejor.

### Nuestro ejemplo

En nuestro ejemplo vamos a suministrar a un soldado que arma debería usar en el combate. Para ello, hemos creado las siguientes clases:

#### Las armas convertidas a Clase

public class Revolver

{

public string Disparar()

{

return "Pum Pum ..";

}

}

public class Rifle

{

public string Disparar()

{

return "Pum pum pum pum pum ..";

}

}

public class Escopeta

{

public string Disparar()

{

return "pum PUMMM !! ..";

}

}

Como se darán cuenta, las armas implementan el método**Disparar()**, la cual el disparo varía dependiendo del tipo de arma que se elija.

#### Nuestra clase Soldado

Esta clase implementa los métodos necesarios para realizar los disparos para cada arma.

public class Soldado

{

public string DispararRevolver()

{

return new Revolver().Disparar();

}

public string DispararRifle()

{

return new Rifle().Disparar();

}

public string DispararEscopeta()

{

return new Escopeta().Disparar();

}

}

* **¿Cual es el problema?,**que la dependencia la tiene que implementar el mismo Soldado.
* **¿Y eso en que nos afecta?**
  + Vamos a tener que modificar nuestra clase Soldado en un futuro si queremos agregar más armas, ahora esta simple porque solo hay 3 armas y cada una tiene un método. ¿Pero si fuerán 200 armas y cada una tiene 100 métodos?.
  + Realizar una prueba unitaria sería bastante tedioso, hay que probar método por método.
  + La clase a futuro va a ser un dolor de cabeza, teniendo un código que no permita su mantenimiento/mejora a futuro.
  + Al momento de instanciar la clase, vamos a tener que **llamar a c/u de los métodos**para manipular el arma seleccionada. Eso nos daría un serio problema, porque tendríamos que **modificar el código constantemente** si es que quisieramos cambiar de arma o hacer un uso **excesivo y vulgar de los IF o SWITCH para saber con que arma se debe trabajar**.

### ¿Cual es la solución?

Implementar una **interface** yque el **constructor de la clase Soldado la reciba** como parámetro.

#### Paso #1

Crear la interface y hacer que las armas implementen dicha interface.

public interface IArma {

string Disparar();

}

public class Revolver : IArma

{

public string Disparar()

{

return "Pum Pum ..";

}

}

public class Rifle : IArma

{

public string Disparar()

{

return "Pum pum pum pum pum ..";

}

}

public class Escopeta : IArma

{

public string Disparar()

{

return "pum PUMMM !! ..";

}

}

#### Paso #2

Modificar nuestra clase Soldado para que reciba dicha interface.

public class Soldado

{

protected IArma arma;

public Soldado(IArma \_arma)

{

this.arma = \_arma;

}

public string Disparar()

{

return this.arma.Disparar();

}

}

Si se dan cuenta ahora, solo tenemos un **método "Disparar"**. Nuestra clase Soldado sabe que debe disparar un Arma, es nuestro constructor el encargado de setear que arma debe usar.

Veamos como se instancia nuestra clase Soldado usando como arma principal el Revolver:

var soldado = new Soldado(new Revolver());

soldado.Disparar();

Si queremos usar la escopeta:

var soldado = new Soldado(new Escopeta());

soldado.Disparar();

¿Y el rifle?

var soldado = new Soldado(new Rifle());

soldado.Disparar();

¿Se dieron cuenta lo lindo que ha quedado nuestro código?. Al final lo que hemos hecho es eliminar la dependenia de una clase dentro de otra clase.

**NOTA**: Me imagino que si has comprendido el ejemplo, tu mente ahora procesa varios casos para la cual usaría la **inyección de dependencia.**Sino fuera así, deja un comentario y te puedo orientar.

**Dependency property**: A property that is backed by a **[DependencyProperty](http://msdn.microsoft.com/en-us/library/system.windows.dependencyproperty.aspx)**.

The purpose of dependency properties is to provide a way to compute the value of a property **based** on the **value of other inputs**.

These other inputs might include system properties such as themes and user preference, just-in-time property determination mechanisms such as data binding and animations/storyboards, multiple-use templates such as resources and styles, or values known through parent-child relationships with other elements in the element tree.

KeyWords

* C#
* C++
* Java
* SQL

Interviews Questions:

1. Experience and Importance of the support, testing and develop procedures.
2. How to solve the problems of replication between two Databases.
   * Pooling or Observer
3. What is a BTree.

. Arbol balanceado:  a **B-tree** is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time. The **B-tree** is a generalization of a binary search tree in that a node can have more than two children.

Uses: Index in data bases.

1. Stateless and statefull protocol, what type is HTTP?
   1. Stateless: Sin estado. Asincrono. HTTP.
   2. Statefull: Con estado. Syncrono. FTP.
2. What is AJAX and uses?
   1. **JavaScript Asíncrono y XML (AJAX)** no es una tecnología por sí misma, es un término que describe un nuevo modo de utilizar conjuntamente varias tecnologías existentes. Esto incluye: [HTML](https://developer.mozilla.org/es/docs/Web/Guide/es/HTML) o [XHTML](https://developer.mozilla.org/es/docs/Web/Guide/es/XHTML), [CSS](https://developer.mozilla.org/es/docs/Web/Guide/es/CSS), [JavaScript](https://developer.mozilla.org/es/docs/Web/Guide/es/JavaScript), [DOM](https://developer.mozilla.org/es/docs/Web/Guide/es/DOM), [XML](https://developer.mozilla.org/es/docs/Web/Guide/es/XML), [XSLT](https://developer.mozilla.org/es/docs/Web/Guide/es/XSLT), y el objeto [XMLHttpRequest](https://developer.mozilla.org/es/docs/Web/Guide/es/XMLHttpRequest). Cuando estas tecnologías se combinan en un modelo AJAX, es posible lograr aplicaciones web capaces de actualizarse continuamente sin tener que volver a cargar la página completa. Esto crea aplicaciones más rápidas y con mejor respuesta a las acciones del usuario.
3. Desgin Pattern: Singleton, Observe.
   1. Singleton:

A class of which only a single instance can exist and it’s globaly accessible. Methods: static Singleton getInstance(), the constructor is private.

* 1. Observe:

A way of notifying change to a number of classes. Communicate the changes in one class to the reset connected. 2 Interfaces: ISubject and IObserver. Methods: ISubject.AttachObserber(), ISubject.DettachObserber() and ISubject.SendNotification().

* 1. Factory- Abstract Factory:

The abstract class or interface constructor must be protected. Used in the List of the User Interface to group the same family. Abstract-Factory = Factory of factories (grouping different families). getType().

1. Create a small numerical function.

### [Senior Software Engineer at Amazon was asked...](https://www.glassdoor.ca/Interview/Amazon-Senior-Software-Engineer-Interview-Questions-EI_IE6036.0,6_KO7,31.htm)

18 Oct, 2011

|  |
| --- |
| Rand(7) from Rand(5)  [7 Answers](https://www.glassdoor.ca/Interview/Rand-7-from-Rand-5-QTN_200619.htm)  The simple solution is to implement RandBin() using Rand5(). int RandBin() { }  Sorry this UI posts without warning int RandBin() { int rand5Res = Rand5(); return rand5Res &lt; 2 ? 0 : rand5Res &lt; 4 ? 1 : RandBin(); } And then use RandBin() to implement Rand7; int Rand7() { int rand = RandBin() &lt;&lt; 2 | RandBin() &lt;&lt; 1 | RandBin(); return rand &lt; 7 ? rand : Rand7(); }  int rand7() { while(1) { int n = ((rand5()%2)\*4 + (rand5()%2)\*2 + (rand5()%2)\*1); if(n == 0) continue; return n; } } The rand5()%2 will generate 0 and 1 with equal probability and we need 3 bits since we are going from 000 upto 111. So we call this function thrice for each bit position. |

C++

int rand7(){

int x=8;

while(x>7)

x=rand5()+5\*rand5()-5;

return x;}

Desgin Pattern:

# Design Patterns

In software engineering, a **design pattern** is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

### Uses of Design Patterns

Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns.

Often, people only understand how to apply certain software design techniques to certain problems. These techniques are difficult to apply to a broader range of problems. Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem.

In addition, patterns allow developers to communicate using well-known, well understood names for software interactions. Common design patterns can be improved over time, making them more robust than ad-hoc designs.

### [Creational design patterns](https://sourcemaking.com/design_patterns/creational_patterns)

These design patterns are all about class instantiation. This pattern can be further divided into class-creation patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.

[](https://sourcemaking.com/design_patterns/abstract_factory)

* [**Abstract Factory**](https://sourcemaking.com/design_patterns/abstract_factory)  
  Creates an instance of several families of classes
* [**Builder**](https://sourcemaking.com/design_patterns/builder)  
  Separates object construction from its representation
* [**Factory Method**](https://sourcemaking.com/design_patterns/factory_method)  
  Creates an instance of several derived classes
* [**Object Pool**](https://sourcemaking.com/design_patterns/object_pool)  
  Avoid expensive acquisition and release of resources by recycling objects that are no longer in use
* [**Prototype**](https://sourcemaking.com/design_patterns/prototype)  
  A fully initialized instance to be copied or cloned
* [**Singleton**](https://sourcemaking.com/design_patterns/singleton)  
  A class of which only a single instance can exist

### [Structural design patterns](https://sourcemaking.com/design_patterns/structural_patterns)

These design patterns are all about Class and Object composition. Structural class-creation patterns use inheritance to compose interfaces. Structural object-patterns define ways to compose objects to obtain new functionality.

[](https://sourcemaking.com/design_patterns/decorator)

* [**Adapter**](https://sourcemaking.com/design_patterns/adapter)  
  Match interfaces of different classes
* [**Bridge**](https://sourcemaking.com/design_patterns/bridge)  
  Separates an object’s interface from its implementation
* [**Composite**](https://sourcemaking.com/design_patterns/composite)  
  A tree structure of simple and composite objects
* [**Decorator**](https://sourcemaking.com/design_patterns/decorator)  
  Add responsibilities to objects dynamically
* [**Facade**](https://sourcemaking.com/design_patterns/facade)  
  A single class that represents an entire subsystem
* [**Flyweight**](https://sourcemaking.com/design_patterns/flyweight)  
  A fine-grained instance used for efficient sharing
* **[](https://sourcemaking.com/design_patterns/proxy)**

[**Private Class Data**](https://sourcemaking.com/design_patterns/private_class_data)  
Restricts accessor/mutator access

* [**Proxy**](https://sourcemaking.com/design_patterns/proxy)  
  An object representing another object

### [Behavioral design patterns](https://sourcemaking.com/design_patterns/behavioral_patterns)

These design patterns are all about Class's objects communication. Behavioral patterns are those patterns that are most specifically concerned with communication between objects.

[](https://sourcemaking.com/design_patterns/interpreter)

* [**Chain of responsibility**](https://sourcemaking.com/design_patterns/chain_of_responsibility)  
  A way of passing a request between a chain of objects
* [**Command**](https://sourcemaking.com/design_patterns/command)  
  Encapsulate a command request as an object
* [**Interpreter**](https://sourcemaking.com/design_patterns/interpreter)  
  A way to include language elements in a program
* [**Iterator**](https://sourcemaking.com/design_patterns/iterator)  
  Sequentially access the elements of a collection
* [**Mediator**](https://sourcemaking.com/design_patterns/mediator)  
  Defines simplified communication between classes
* [**Memento**](https://sourcemaking.com/design_patterns/memento)  
  Capture and restore an object's internal state
* [**Null Object**](https://sourcemaking.com/design_patterns/null_object)  
  Designed to act as a default value of an object
* [**Observer**](https://sourcemaking.com/design_patterns/observer)  
  A way of notifying change to a number of classes
* **[](https://sourcemaking.com/design_patterns/state)**

[**State**](https://sourcemaking.com/design_patterns/state)  
Alter an object's behavior when its state changes

* [**Strategy**](https://sourcemaking.com/design_patterns/strategy)  
  Encapsulates an algorithm inside a class
* [**Template method**](https://sourcemaking.com/design_patterns/template_method)  
  Defer the exact steps of an algorithm to a subclass
* [**Visitor**](https://sourcemaking.com/design_patterns/visitor)  
  Defines a new operation to a class without change

### Criticism

The concept of design patterns has been criticized by some in the field of computer science.

#### Targets the wrong problem

The need for patterns results from using computer languages or techniques with insufficient abstraction ability. Under ideal factoring, a concept should not be copied, but merely referenced. But if something is referenced instead of copied, then there is no "pattern" to label and catalog. Paul Graham writes in the essay [**Revenge of the Nerds**](http://www.paulgraham.com/icad.html).

Peter Norvig provides a similar argument. He demonstrates that 16 out of the 23 patterns in the Design Patterns book (which is primarily focused on C++) are simplified or eliminated (via direct language support) in Lisp or Dylan.

#### Lacks formal foundations

The study of design patterns has been excessively ad hoc, and some have argued that the concept sorely needs to be put on a more formal footing. AtOOPSLA 1999, the Gang of Four were (with their full cooperation) subjected to a show trial, in which they were "charged" with numerous crimes against computer science. They were "convicted" by ⅔ of the "jurors" who attended the trial.

#### Leads to inefficient solutions

The idea of a design pattern is an attempt to standardize what are already accepted best practices. In principle this might appear to be beneficial, but in practice it often results in the unnecessary duplication of code. It is almost always a more efficient solution to use a well-factored implementation rather than a "just barely good enough" design pattern.

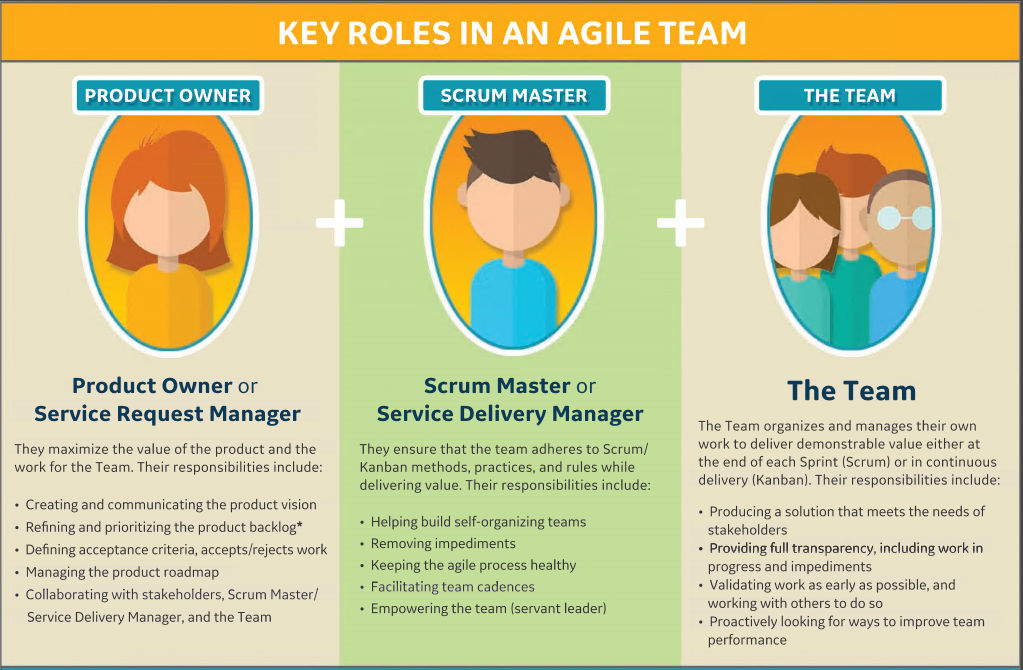
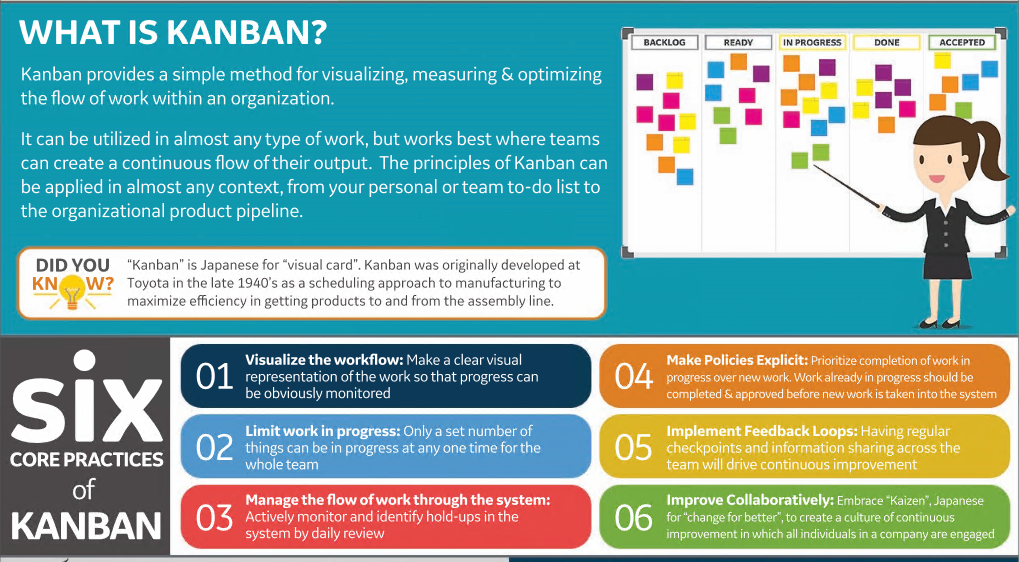
#### Does not differ significantly from other abstractions

Some authors allege that design patterns don't differ significantly from other forms of abstraction, and that the use of new terminology (borrowed from the architecture community) to describe existing phenomena in the field of programming is unnecessary. The Model-View-Controller paradigm is touted as an example of a "pattern" which predates the concept of "design patterns" by several years. It is further argued by some that the primary contribution of the Design Patterns community (and the Gang of Four book) was the use of Alexander's pattern language as a form of documentation; a practice which is often ignored in the literature.

<https://sourcemaking.com/design_patterns>

# Agile

# 

# SOLID

## **Revising SOLID principles**

S stands for SRP (Single responsibility principle):- A class should take care of only one responsibility.

O stands for OCP (Open closed principle):- Extension should be preferred over modification.

L stands for LSP (Liskov substitution principle):- A parent class object should be able to refer child objects seamlessly during runtime polymorphism.

I stands for ISP (Interface segregation principle):- Client should not be forced to use a interface if it does not need it.

D stands for DIP (Dependency inversion principle) :- High level modules should not depend on low level modules but should depend on abstraction.

<https://www.codeproject.com/Articles/703634/SOLID-architecture-principles-using-simple-Csharp>

### Modelo vista controlador (MVC)

Modelo Vista Controlador (MVC) es un estilo de arquitectura de software que separa los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos.

Se trata de un modelo muy maduro y que ha demostrado su validez a lo largo de los años en todo tipo de aplicaciones, y sobre multitud de lenguajes y plataformas de desarrollo.

* El **Modelo** que contiene una representación de los datos que maneja el sistema, su lógica de negocio, y sus mecanismos de persistencia.
* La **Vista**, o interfaz de usuario, que compone la información que se envía al cliente y los mecanismos interacción con éste.
* El **Controlador**, que actúa como intermediario entre el Modelo y la Vista, gestionando el flujo de información entre ellos y las transformaciones para adaptar los datos a las necesidades de cada uno.

#### **El modelo es el responsable de:**

* Acceder a la capa de almacenamiento de datos. Lo ideal es que el modelo sea independiente del sistema de almacenamiento.
* Define las reglas de negocio (la funcionalidad del sistema). Un ejemplo de regla puede ser: "Si la mercancía pedida no está en el almacén, consultar el tiempo de entrega estándar del proveedor".
* Lleva un registro de las vistas y controladores del sistema.
* Si estamos ante un modelo activo, notificará a las vistas los cambios que en los datos pueda producir un agente externo (por ejemplo, un fichero por lotes  que actualiza los datos, un temporizador que desencadena una inserción, etc.).

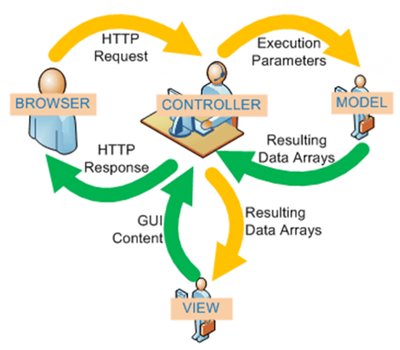
#### **El controlador es responsable de:**

* Recibe los eventos de entrada (un clic, un cambio en un campo de texto, etc.).
* Contiene reglas de gestión de eventos, del tipo "SI Evento Z, entonces Acción W". Estas acciones pueden suponer peticiones al modelo o a las vistas. Una de estas peticiones a las vistas puede ser una llamada al método "Actualizar()". Una petición al modelo puede ser "Obtener\_tiempo\_de\_entrega ( nueva\_orden\_de\_venta )".

#### **Las vistas son responsables de:**

* Recibir datos del modelo y los muestra al usuario.
* Tienen un registro de su controlador asociado (normalmente porque además lo instancia).
* Pueden dar el servicio de "Actualización()", para que sea invocado por el controlador o por el modelo (cuando es un modelo activo que informa de los cambios en los datos producidos por otros agentes).

El flujo que sigue el control generalmente es el siguiente:



1. El usuario interactúa con la interfaz de usuario de alguna forma (por ejemplo, el usuario pulsa un botón, enlace, etc.)
2. El controlador recibe (por parte de los objetos de la interfaz-vista) la notificación de la acción solicitada por el usuario. El controlador gestiona el evento que llega, frecuentemente a través de un gestor de eventos (handler) o callback.
3. El controlador accede al modelo, actualizándolo, posiblemente modificándolo de forma adecuada a la acción solicitada por el usuario (por ejemplo, el controlador actualiza el carro de la compra del usuario). Los controladores complejos están a menudo estructurados usando un patrón de comando que encapsula las acciones y simplifica su extensión.
4. El controlador delega a los objetos de la vista la tarea de desplegar la interfaz de usuario. La vista obtiene sus datos del modelo para generar la interfaz apropiada para el usuario donde se refleja los cambios en el modelo (por ejemplo, produce un listado del contenido del carro de la compra). El modelo no debe tener conocimiento directo sobre la vista. Sin embargo, se podría utilizar el patrón Observador para proveer cierta indirección entre el modelo y la vista, permitiendo al modelo notificar a los interesados de cualquier cambio. Un objeto vista puede registrarse con el modelo y esperar a los cambios, pero aun así el modelo en sí mismo sigue sin saber nada de la vista. El controlador no pasa objetos de dominio (el modelo) a la vista aunque puede dar la orden a la vista para que se actualice. Nota: En algunas implementaciones la vista no tiene acceso directo al modelo, dejando que el controlador envíe los datos del modelo a la vista.
5. La interfaz de usuario espera nuevas interacciones del usuario, comenzando el ciclo nuevamente.

### Part 1 – C# Design Pattern Interview Questions (Basic)

This first part covers the basic C# Design Pattern Interview Questions and Answers

#### Q1. What is Pattern Designs and explain its significance?

**Answer:**  
Design Patterns are efficient solutions to the General Software Design problems. The common problems are resolved by applying these Design Patterns which are the best practices to obtain efficient solutions for the complex known and common problems in the design challenges of the software design and development processes. These design solutions for the common problems were obtained after performing several trial and error methods by many developers for a long time in many designs to obtain these effective solutions to the commonly known problems. There are different types of Design patterns. They are Behavioural Patterns, Creational Patterns, and Structural Patterns.

#### Q2. What are the different uses of Design Patterns?

**Answer:**  
The below are the different uses of the Design Patterns –Speeds up the development process

1. Ensures effective and efficient software designs
2. More reusability for the common problems across the design model
3. Enables code readability
4. Good documentation and easier to maintain for other developers
5. Robust applications can be developed
6. Ensures customer retention for delivering efficient and robust solutions

Let us move to the next C# Design Pattern Interview Questions.

#### Q3. What are the different types of Design Patterns and explain?

**Answer:**  
The different types of Design patterns are Behavioural Patterns, Creational Patterns and Structural Patterns. They are explained as below.  
**Behavioural Pattern:**The Behavioural Pattern is all about the communication between the several objects of the classes. It mainly concentrates on the communication and invocation between the objects.  
**Creational Pattern:**The Creational Pattern is about the instantiation of the class and also further extends to the object creation. This category can be further divided into Class creational patterns and object creational patterns. The main theme used in this pattern is a delegation to get the work done. It mainly depends on the delegation model.  
**Structural Pattern:**The Structural Pattern is about the composition of the classes and objects while interacting with several other classes. This ensures the application is more loosely coupled than getting complex.

#### Q4. What is Singleton Pattern and how can it be implemented in C#?

**Answer:**  
This is the basic C# Design Pattern Interview Question asked in an interview. A Singleton Pattern is a design pattern that comes under the Creational design pattern which is used to instantiate only a single object from that class. Finally, that class will be confined only to a single object in its entire life cycle. The Singleton[design pattern](https://www.educba.com/what-is-design-pattern/) is very important in hiding the class instantiation functionality to be hidden from the other classes in order to maintain the object instantiation limitations for the purpose of application or design requirements to be fulfilled. Always only a single instance can be created from Singleton Design pattern. In C# this can be implemented using the single constructor which is given with access keyword private and without any parameters i.e., default no-args constructor. This method does not provide any thread safety. For any thread safety, pre-checks like null checks can be evaluated before proceeding with the next flow of execution in order to prevent the instantiation of other objects once a single object is created out of that class.

#### Q5. Which Design Pattern is used to implement any complex method or object?

**Answer:**  
There are some cases where there will need to implement complex methods or objects such as where a method needs to be implemented with more than 6 or 7 arguments. In that case, the method is going to be complex and this results in poor quality of code. To avoid this problem, Builder Pattern can be used to implement an efficient way of handling and operating complex methods or objects. Builder Pattern will have a chain of methods and a build() method in order to be executed at the end of calling all the methods. This will construct a complex object easily by invoking in a chain method.

### Part 2 – C# Design Pattern Interview Questions (Advanced)

Let us now have a look at the advanced C# Design Pattern Interview Questions.

#### Q6. What are the advantages of using Design Pattern with any Object Oriented Programming languages like C#?

**Answer:**  
The advantages of using Design Patterns especially with Object Oriented Programming Languages like [C# or Java](https://www.educba.com/c-sharp-vs-javascript/) etc. are as below:

1. It gives a proven solution to a problem
2. Ensures loosely coupled application development
3. Enables instant high-level overview after l looking at the code immediately
4. Easier to understand
5. Easier to maintain and develop new features.

#### Q7. What is prototype Design Pattern?

**Answer:**  
The Prototype Design Pattern comes under Creational Design Pattern which is used to clone the objects in the form of the prototypal instance. This can be implemented in C# programming language by declaring a base abstract class by using the clone() method.

Let us move to the next C# Design Pattern Interview Question.

#### Q8. What is a Factory Design Pattern?

**Answer:**  
A factory Design Pattern is a Creational Design Pattern which is used to create instances of many derived classes. The name itself says that a Factory Design Pattern is used to instantiate many objects as required.

#### Q9. What is a Façade Design Pattern?

**Answer:**  
This is the most asked C# Design Pattern Interview Questions in an interview. A Façade Design Pattern is a Structural Design Pattern which is implemented to represent a complete subsystem as a single class. This is efficient in case there is a huge number of classes in different areas.

#### Q10. What are the SOLID design principles?

**Answer:**  
The SOLID Design Principles are the best-known principles in the area of Object-Oriented Software Design and Development. The five SOLID design principles are listed below:

1. Single Responsibility Principle (SRP)
2. Open/Closed Principle (OCP)
3. Liskov Substitution Principle (LSP)
4. Interface Segregation Principle (ISP)
5. Dependency Inversion Principle (DIP)