C# Design Patterns Demystified: Upgrade your skill to senior development

**What are design patterns and what problems solve?**

Design patterns are reusable and generalizable solutions to common problems that software developers encounter when designing and building software applications.

They are established best practices that have evolved over time and have been identified as effective solutions to recurring design and architectural challenges in software development.

Design patterns provide a common vocabulary and framework for developers to communicate and share their knowledge about how to solve specific problems in a structured and maintainable way, addressing a variety of issues and problems in software development, including:

**Code Reusability:** Design patterns promote the reuse of proven, well-tested design and coding techniques, this can lead to more efficient development processes and less duplication of effort.

**Scalability:** They help in designing systems that can easily adapt and scale to changing requirements, whether it's adding new features or handling increased user loads.

**Maintainability:** Design patterns encourage clean and modular code, making it easier to maintain and extend software over time.

This can reduce the cost of maintenance and decrease the likelihood of introducing bugs.

**Flexibility:** Design patterns allow for flexibility in system design; developers can modify or extend existing patterns to fit specific project requirements without completely redesigning the entire system.

**Communication:** Design patterns provide a common language for developers to discuss and document solutions to common problems; this improves collaboration and knowledge sharing among team members.

**Performance:** While not all design patterns directly address performance concerns, some can help optimize code for better runtime efficiency.

**Separation of Concerns:** Many design patterns promote the separation of different concerns within an application, such as separating user interface logic from business logic, this separation improves code organization and maintainability.

**Testability:** Patterns that encourage loose coupling and modularity often make it easier to write unit tests and perform testing in isolation.

There are numerous design patterns, which are typically categorized into three main groups:

**Creational Patterns:** These patterns focus on object creation mechanisms, trying to create objects in a manner suitable to the situation.

Examples include the Singleton, Factory Method, and Abstract Factory patterns.

**Structural Patterns:** Structural patterns deal with object composition, helping to form relationships between objects to create larger, more complex structures.

Examples include the Adapter, Decorator, and Composite patterns.

**Behavioral Patterns:** Behavioral patterns are concerned with how objects interact and communicate with one another.

They help define the responsibilities of objects and how they collaborate.

Examples include the Observer, Strategy, and Command patterns.

By applying these design patterns appropriately, developers can make their software more maintainable, scalable, and adaptable while solving common design challenges effectively.

However, it's essential to use design patterns judiciously and not overcomplicate a solution by applying patterns unnecessarily.

The choice of a design pattern should always align with the specific needs and constraints of the project.

**SOLID principles.**

The SOLID principles were formulated in the early 2000s and deal with defining a certain style in software development.

If the programmer adopts certain practices, they will be able to write code that is clearer, more maintainable, modular, and testable.

These principles were formulated to address architectural issues in the development of larger projects that need to be maintained over time.

SOLID is an acronym and defines 5 important principles:

(S) Single Responsibility Principle (SRP)

(O) Open-Closed Principle (OCP)

(L) Liskov Substitution Principle (LSP)

(I) Interface Segregation Principle (ISP)

(D) Dependency Inversion Principle (DIP)

**Single responsability principle (SRP)**

This principle states that each object should have a single responsibility, dealing with a specific, singular task.

For example, if we have a bulletin board class, it should be responsible for post management, including creation, modification, and deletion, but not, for instance, for saving those posts.

The responsibility of the bulletin board class is therefore to handle posts, while the saving task would be delegated to another class.

This separation is important because, if we were to change the way posts are saved, such as switching from a database to a text file in the file system or the cloud, we would be forced to modify the bulletin board class, this would go against another principle we'll discuss, the Open-Closed Principle.

The goal is to write code that minimizes impact: if we need to modify the logic of post writing, we'll only need to modify the class responsible for that, such as, for example, a Feed class, which would be responsible solely for post management.

public class Feed

{

private readonly List<string> Posts = new List<string>();

private static int Count = 0;

public void AddPost(string Text)

{

Posts.Add($"{++Count}: {Text}");

}

public void RemovePost(int index)

{

Posts.RemoveAt(index);

}

public override string ToString()

{

StringBuilder sb = new StringBuilder();

Posts.ForEach(x => sb.Append(x));

return sb.ToString();

}

}

We might be tempted to add a "save" method to the class to save the posts, but this would violate the Single Responsibility Principle, therefore, we delegate the sole responsibility of saving the posts to a new manager:

public class SavingManager

{

public void Save(Feed feed, string fileName, bool overwrite = false)

{

if (overwrite || !File.Exists(fileName))

{

File.WriteAllText(fileName, feed.ToString());

}

}

}

So we can use the "feed" object and later save it using the following code.

It becomes evident that the creation and deletion of feeds, as well as their saving, are different activities delegated to different objects:

Feed feed = new Feed();

feed.AddPost($"This is the first post {Environment.NewLine}");

feed.AddPost($"This is the second post {Environment.NewLine}");

SavingManager savingManager = new SavingManager();

savingManager.Save(feed, @"C:\feed.txt");

**Open close principle (OCP).**

This principle dictates that well-structured code should always allow for the extension of its functionality while simultaneously preventing the modification of existing classes.

This is done so that the addition of new features happens through new modules, rather than by modifying existing ones, which would have significant impacts on the already written code.

Let's take an example: suppose we need to implement different types of invoices, each of which gives rise to a specific discount.

At first glance, we could write the following code, using enumeration to distinguish between the types and using an "if" statement to apply a specific discount based on the type:

public class Invoice

{

public double GetInvoiceDiscount(double amount, InvoiceType invoiceType)

{

double finalAmount = 0;

if (invoiceType == InvoiceType.FinalInvoice)

{

finalAmount = amount - 100;

}

else if (invoiceType == InvoiceType.ProposedInvoice)

{

finalAmount = amount - 50;

}

return finalAmount;

}

}

public enum InvoiceType

{

FinalInvoice,

ProposedInvoice

};

However, this code has a problem: it violates the Open-Closed Principle, because if in the future we need to add a new type of invoice, we will have to modify this class by adding a new "if" statement.

The solution is to create a base class that implements default behavior that can be overridden in the classes that inherit from it.

public class Invoice

{

public virtual double GetInvoiceDiscount(double amount)

{

return amount - 10;

}

}

Then, we proceed to have our concrete classes inherit from the interface, implementing the specific discount logic based on the type of invoice.

We will create a concrete class for each invoice type:

public class FinalInvoice : Invoice

{

public override double GetInvoiceDiscount(double amount)

{

return base.GetInvoiceDiscount(amount) - 50;

}

}

public class ProposedInvoice : Invoice

{

public override double GetInvoiceDiscount(double amount)

{

return base.GetInvoiceDiscount(amount) - 30;

}

}

This way, if we want to add a new invoice type, we will create a new concrete class, just like we did with FinalInvoice or ProposedInvoice.

In other words, we will simply add a new module without modifying an existing one.

The code that now adheres to the Open-Closed Principle can be used as follows:

Invoice finalInvoice = new FinalInvoice();

Invoice proposedInvoice = new ProposedInvoice();

double finalInvoiceAmount = finalInvoice.GetInvoiceDiscount(20000);

double proposedInvoiceAmount = proposedInvoice.GetInvoiceDiscount(20000);

**Liskov sobstitution principle**

This principle dictates that derived classes should be able to extend their base classes without modifying their behavior, meaning that derived classes should be substitutable for their base types.

Let's look at an example of a violation of this principle to better understand the concept.

Suppose you have a base class Animal:

public class Animal

{

public virtual void MakeSound()

{

Console.WriteLine("An animal makes a sound.");

}

}

Now, you create two derived classes, Dog and Cat, which inherit from the Animal class:

public class Dog : Animal

{

public override void MakeSound()

{

Console.WriteLine("A dog barks.");

}

}

public class Cat : Animal

{

public override void MakeSound()

{

Console.WriteLine("A cat meows.");

}

}

Suppose we introduce a new derived class called Fish that inherits from the Animal class:

public class Fish : Animal

{

// This class does not override the MakeSound method becouse Fish doesn't make sound

}

In this case, SilentAnimal is derived from Animal but does not override the MakeSound() method, which means it inherits the default behavior from the base class.

Now, let's see how this violates the Liskov Substitution Principle:

Animal myPet3 = new Fish();

myPet3.MakeSound(); // Output: An animal makes a sound.

In this example, we have created an instance of SilentAnimal and assigned it to a variable of type Animal.

When we call MakeSound() on myPet3, we get "An animal makes a sound," which is the default behavior inherited from the Animal base class, but this is wrong because the derived class contradicts the behavior of the base class

To implement a Fish class that adheres to the Liskov Substitution Principle you can override the MakeSound() method to provide a silent behavior, which is typically represented by an empty or minimal sound.

Here's how you can implement a Fish class:

public class Fish : Animal

{

public override void MakeSound()

{

// This class does not override the MakeSound method becouse Fish doesn't make sound

Console.WriteLine("A fish doesn't make sound");

}

}

**Interface segregation principle (ISP)**

The Interface Segregation Principle states that no client should be forced to implement large interfaces with numerous methods; a component should be scalable and, as such, should implement only the methods it needs by inheriting simple interfaces.

Let's take an example: suppose we need to implement a multifunction printer that can print, scan, and fax documents, if we build it this way, it consumes an interface with many heterogeneous methods:

class MultifunctionalPrinter : IMachine

{

public void fax(Document document)

{

}

public void Print(Document document)

{

}

public void Scan(Document document)

{

}

}

This is the interface implemented by the class above:

public interface IMachine

{

void Print(Document document);

void fax(Document document);

void Scan(Document document);

}

It's evident that the interface above is not generic and does not abstract, as if you wanted to implement a simple printer that only prints or a simple scanner, etc., and if we make our classes implement the same interface, we would be forced to implement unnecessary methods declared in the interface.

The solution is to break down the interface with many methods into multiple simple interfaces.

Afterward, the component can inherit from more than one interface (a class supports the ability to inherit from multiple interfaces in .NET, which is not possible when it tries to inherit from multiple classes).

public interface IPrinter

{

void Print(Document document);

}

public interface IScanner

{

void Scan(Document document);

}

public interface IFax

{

void Fax(Document document);

}

So, our simple printer will implement only the "print" method using the IPrint interface:

class Printer : IPrinter

{

public void Print(Document document)

{

}

}

While the multifunction printer can inherit from multiple interfaces.

class MultifunctionalPrinter : IPrinter, IScanner, IFax

{

public void Fax(Document document)

{

}

public void Scan(Document document)

{

}

public void Print(Document document)

{

}

}

**Dependency inversion principle (DIP)**

The Dependency Inversion Principle states that a high-level module should not depend on a low-level module, but both should depend on an abstraction.

An abstraction should not depend on details, and details should not depend on an abstraction.

In other words, the interaction between classes should occur through abstract classes or interfaces.

Let's see an example.

Suppose we want to implement a class that allows us to log errors:

class ExceptionLogger

{

private ILogger logger;

public ExceptionLogger(ILogger logger)

{

this.logger = logger;

}

public void LogException(Exception ex)

{

logger.LogMessage(ex.Message);

}

}

As we can see, the class is associated with an interface of type ILogger and not directly with a Logger object.

This association is referred to as loose coupling, as opposed to tight coupling where objects are strongly dependent on each other.

The ILogger interface is defined as follows:

public interface ILogger

{

void LogMessage(string message);

}

Weak coupling allows us to delegate the concrete implementation of how to log the message to a third class, so that the logic of the third class is completely independent of the described implementation context.

For example, we could decide to save messages to a database rather than a file like this:

class DbLogger : ILogger

{

public void LogMessage(string message)

{

//here saving logic on DB

}

}

class FileLogger : ILogger

{

public void LogMessage(string message)

{

//here saving logic on file

}

}

The client that instantiates the ExceptionLogger object to save the exception can decide whether to save errors to a database or a file like this:

ExceptionLogger el = new ExceptionLogger(new DbLogger());

//or

//ExceptionLogger el = new ExceptionLogger(new FileLogger());

try

{

//here some code...

}

catch (Exception ex)

{

el.LogException(ex);

}

Notice how we pass a concrete object, DbLogger, that implements the Ilogger interface to the ExceptionLogger constructor.

Therefore, the LogException method will use the logic defined in the DbLogger class to save the message to the database.

If we had passed an instance of FileLogger to our constructor, the message would be stored in a file.

This change has minimal impact on the overall logic, which remains almost identical in both cases.

If we needed to add a third error storage method, we would simply add a new concrete class that implements Ilogger, following the Open-Closed Principle.

In other words, we have created code that is open to future changes without impacting existing code.

**How design patterns follow SOLID principles?**

Design patterns and SOLID principles are complementary concepts in software engineering. SOLID is an acronym that represents a set of five principles aimed at promoting clean, maintainable, and flexible software design.

These principles guide developers in creating software that is easier to understand, extend, and maintain.

Design patterns, on the other hand, provide proven solutions to common design challenges.

Let's see how design patterns align with the SOLID principles:

**Single Responsibility Principle (SRP):**

This principle states that a class should have only one reason to change.

Design patterns often help in achieving SRP by promoting separation of concerns.

For example, the Observer pattern separates the subject (which changes) from the observers (which react to changes), thereby adhering to SRP.

**Open/Closed Principle (OCP):**

The OCP encourages software entities to be open for extension but closed for modification.

Many design patterns, such as the Strategy and Decorator patterns, follow this principle.

They allow you to add new behavior or features to a system without changing existing code.

**Liskov Substitution Principle (LSP):**

This principle states that objects of a derived class should be substitutable for objects of the base class without affecting the correctness of the program.

Some design patterns, like the Factory Method and Abstract Factory patterns, support the LSP by providing a way to create objects without specifying their concrete classes, thus ensuring that derived classes can be used interchangeably with their base class.

**Interface Segregation Principle (ISP):**

ISP suggests that clients should not be forced to depend on interfaces they don't use.

Design patterns like the Adapter and Bridge patterns can help adhere to ISP by allowing clients to interact with specific interfaces that are tailored to their needs rather than having a monolithic interface with unnecessary methods.

**Dependency Inversion Principle (DIP):**

DIP advocates that high-level modules should not depend on low-level modules but instead both should depend on abstractions.

Design patterns like the Dependency Injection (DI) pattern and the Observer pattern with the use of interfaces help achieve this principle by promoting the use of abstractions (interfaces or abstract classes) to decouple high-level and low-level modules.

In summary, design patterns can be seen as practical implementations of SOLID principles.

By applying design patterns in your software design, you are more likely to adhere to these principles, resulting in code that is modular, maintainable, and easier to extend without causing widespread changes.

Design patterns help to guide you in making design decisions that align with SOLID principles and promote good software design practices.

However, it's essential to understand both the principles and the patterns thoroughly and apply them judiciously to meet the specific needs of your project.

**Creational patterns.**

Creational design patterns are a category of design patterns that deal with object creation mechanisms, providing various ways to create objects while abstracting the instantiation process.

These patterns help manage object creation by controlling which classes to instantiate, how to instantiate them, and where to instantiate them.

Creational patterns aim to make a system more flexible, extensible, and independent of the specific classes it uses for object creation.

**Singleton Pattern**

The Singleton pattern is a creational design pattern that restricts the instantiation of a class to only one instance and provides a global point of access to that instance.

It is used to ensure that there is only one instance of a particular class in an application, and that instance can be easily accessed from anywhere in the codebase.

This is one example:

public sealed class Singleton

{

private static Singleton instance;

private static readonly object lockObject = new object();

private Singleton()

{

// Private constructor to prevent the creation of external instances.

}

public static Singleton Instance

{

get

{

if (instance == null)

{

lock (lockObject)

{

if (instance == null)

{

instance = new Singleton();

}

}

}

return instance;

}

}

// Other methods and properties of the Singleton class can be added here.

}

In this example:

The Singleton class has a private constructor, which means that instances of this class cannot be created directly from the outside.

The static variable instance keeps track of the single instance of the Singleton class. It is initialized as null initially.

The static property Instance is the access point to get the Singleton instance. If instance is null, a new Singleton object is created within a synchronized block to avoid concurrency issues.

The synchronized block ensures that only one thread at a time can create the Singleton instance and that the instance is created only when necessary.

Now you can use the Singleton in your code like this:

Singleton singleton1 = Singleton.Instance;

Singleton singleton2 = Singleton.Instance;

// The two variables singleton1 and singleton2 share the same Singleton instance.

Console.WriteLine(singleton1 == singleton2); // Should be "True".

When you call Singleton.Instance, it will always return the same shared instance of the Singleton class, regardless of how many times it is called.

This is useful for maintaining shared state or ensuring that there is only one instance of a class that manages, for example, database connections or application settings.

Here is a concrete example of using the Singleton pattern in a C# application to create a database connection manager:

public class DatabaseConnection

{

private static DatabaseConnection instance;

private string connectionString;

// Private constructor to avoid creating external instances.

private DatabaseConnection()

{

// Initialize the database connection string.

connectionString = “Server=example.com;Database=mydb;User=myuser;Password=mypassword;";

}

// Properties to access the Singleton instance.

public static DatabaseConnection Instance

{

get

{

if (instance == null)

{

instance = new DatabaseConnection();

}

return instance;

}

}

// Method for opening a database connection.

public void OpenConnection()

{

Console.WriteLine("Open database connection: " + connectionString);

// Here you can implement the code to open the database connection.

}

// Method for closing the database connection.

public void CloseConnection()

{

Console.WriteLine("Database connection closed: " + connectionString);

// Here you can implement the code to close the database connection.

}

}

you can use the DatabaseConnection in your code like this:

// Get the singleton instance of the database connection manager.

DatabaseConnection dbConnection = DatabaseConnection.Instance;

// Open and close the database connection.

dbConnection.OpenConnection();

dbConnection.CloseConnection();

// Verify that the singleton instances are the same.

DatabaseConnection dbConnection2 = DatabaseConnection.Instance;

Console.WriteLine(dbConnection == dbConnection2); // Should print "True".

In this example, DatabaseConnection is a Singleton class that manages the database connection.

The Singleton instance is accessible through the static property Instance.

You can open and close the database connection using the OpenConnection and CloseConnection methods. Being a Singleton class, the operations of opening and closing the connection are performed on the same instance throughout the application, ensuring that multiple connections to the database are not opened.

Here is another concrete example of Singleton represented by the general setting options for managing an application:

public class AppSettings

{

private Dictionary<string, string> settings;

// Instantiate the Singleton when the class is loaded.

private static readonly AppSettings instance = new AppSettings();

// Private constructor to avoid creating external instances.

private AppSettings()

{

// Initialize settings with default values.

settings = new Dictionary<string, string>

{

{ "BackgroundColor", "White" },

{ "FontColor", "Black" },

{ "FontSize", "12pt" }

};

}

// Property for accessing the Singleton instance.

public static AppSettings Instance

{

get { return instance; }

}

// Method to get the value of a specific setting.

public string GetSetting(string key)

{

if (settings.ContainsKey(key))

{

return settings[key];

}

else

{

return null;// Return null if the key does not exist.

}

}

// Method for setting the value of a specific setting.

public void SetSetting(string key, string value)

{

if (settings.ContainsKey(key))

{

settings[key] = value;

}

else

{

settings.Add(key, value);

}

}

}

It's interesting to note that the appsetting instance is obtained when the instance is loaded, which has a drawback - the instance will occupy memory for the entire lifetime of the application, and there's no way to finalize it.

You can use Appsettings in your code like this:

// Get the Singleton instance of the application settings.

AppSettings appSettings = AppSettings.Instance;

// Read and change settings.

Console.WriteLine("Background Color: " + appSettings.GetSetting("BackgroundColor"));

Console.WriteLine("Font Color: " + appSettings.GetSetting("FontColor"));

// Changing the setting.

appSettings.SetSetting("BackgroundColor", "Blue");

Console.WriteLine("Updated Background Color: " + appSettings.GetSetting("BackgroundColor"));

// Check that changes are reflected everywhere.

AppSettings appSettings2 = AppSettings.Instance;

Console.WriteLine("Background Color in Another Instance: " + appSettings2.GetSetting("BackgroundColor"));

// The Singleton instance is the same for appSettings and appSettings2.

Console.WriteLine(appSettings == appSettings2); // Should print "True".

In this example, AppSettings is a Singleton class that manages application settings.

There is only one instance of AppSettings, accessible through the static property Instance.

You can read and modify settings through the GetSetting and SetSetting methods.

Changes made to settings through one instance are reflected in all other instances since they share the same Singleton instance.

**Factory method.**

The Factory Method Pattern provides a method for creating objects.

Here's how the Factory Method Pattern works:

public class Point

{

public double x { get; set; }

public double y { get; set; }

protected Point(double x, double y)

{

this.x = x;

this.y = y;

}

public static Point NewCartesianPoint(double x, double y)

{

return new Point(x, y);

}

public static Point NewPolarPoint(double rho, double theta)

{

return new Point(rho \* Math.Cos(theta), rho \* Math.Sin(theta));

}

}

Let's suppose we have an object representing a point; a point can be expressed using either Cartesian or polar coordinates, both of which are represented by a data type double.

We cannot have an overload of constructors because the signature for both would be identical.

The solution could involve creating the object not through a constructor (the constructor is declared as protected to prevent object instantiation) but through two factory methods (NewCartesianPoint, NewPolarPoint). This methods are static, for this reason is possible to use them without to instantiate the class point, these methods return the instantiated object.

Here is the client side code:

// Client code using Factory Method Pattern.

CreationalPatterns.FactoryMethod.Point cartesianPoint = Point.NewCartesianPoint(3, 5);

CreationalPatterns.FactoryMethod.Point polarPoint = Point.NewPolarPoint(3, 5);

**Prototype**

In general, the goal is to avoid building any physical or computer object from scratch every time, as it would be a significant waste of time.

Let's think about the industry; nothing is completely reinvented.

Instead, a common template is often used as a starting point.

It could be the chassis of a car, for instance.

Once the state of the art is achieved with this common template, different models of cars are designed based on it.

The prototype pattern is precisely applied to avoid redefining an object from scratch.

It starts with an object that has certain characteristics, and this object is copied (cloned) and then customized by modifying some of its features.

Every time, you don't need to redefine all the properties of the object, as this would be a significant waste of time, lead to duplicated and verbose code, and introduce a greater potential for errors in object construction.

In summary, there are several reasons why the prototype pattern should be used, and the more objects we have in our application with a common template, the more it makes sense to apply this pattern.

Prototype pattern is often used to efficiently create new objects when the cost of creating them is more expensive or complex.

Here's an explanation of the Prototype pattern's key concepts and how it works:

Prototype: The prototype is an existing object that serves as a blueprint for creating new objects.

It contains the initial state and structure of the object to be cloned.

This can be an instance of a class, a data structure, or any other object.

Clone: The process of creating a new object by duplicating the prototype is called cloning.

The new object is created with the same properties and state as the prototype but is a separate instance.

Cloning can be shallow or deep, depending on whether it copies only the top-level structure or includes nested objects within the prototype.

Client: The client code is responsible for requesting the creation of new objects based on the prototype.

Instead of creating objects using traditional constructors or factory methods, the client simply asks the prototype to create a new object by cloning itself.

Benefits of the Prototype pattern:

Efficient object creation: The Prototype pattern can be more efficient than creating objects from scratch, especially when object creation is resource-intensive or complex.

Reduces subclassing: It allows for the creation of new objects by cloning existing ones, reducing the need to create subclasses of objects with different configurations.

Dynamic object creation: The Prototype pattern allows for the creation of new object variations at runtime, as the client can change the prototype object or configure it as needed.

Maintainable code: It helps keep the client code more maintainable by centralizing the object creation logic in the prototype, making it easier to manage object variations and updates.

However, there are some considerations when using the Prototype pattern:

Deep copying: If the prototype contains nested objects, it's important to implement deep copying to ensure that the cloned object is a true copy and not just a reference to the original nested objects.

Identifying and managing prototypes: Proper management and identification of prototypes are important to ensure that the client can easily access and clone the appropriate objects.

The Prototype pattern can be implemented by providing a clone method or by using language-specific mechanisms for object copying.

It's a valuable pattern when you need to create objects with similar initial state or when you want to avoid the overhead of creating complex objects from scratch.

In itself, it is not a difficult pattern to apply, but we need to be careful about how the object is copied from the prototype object.

In fact, as long as the copy is made on value types, there is no issue.

However, when we clone reference types, we need to ensure that a deep copy is performed and not a shallow one.

Let's provide an example to illustrate how easy it is to improperly clone a reference type without applying a deep copy: Let's define a Person class and an Address class:

There is a protected method implemented by the .NET Framework in the Object class, which is present in all objects, allowing you to perform a shallow copy of the object.

By using this method in the implementation of our clone method after having Person inherit the ICloneable interface, we obtain the following code:

public class Person : ICloneable

{

public string Name;

public Address Address;

public Person(string name, Address address)

{

Name = name;

Address = address;

}

public object Clone()

{

return (Person)this.MemberwiseClone();

}

}

public class Address

{

public readonly string StreetName;

public int HouseNumber;

public Address(string streetName, int houseNumber)

{

StreetName = streetName;

HouseNumber = houseNumber;

}

}

As mentioned, however, this code has two defects.

The first is that it still provides a shallow copy of the object and not a deep copy, and the second is that the ICloneable interface does not provide explicit information to the programmer about the type of copy (deep or shallow).

Now, if we define a person as Rossi Marco who lives at a certain address, Via Mazzini 18, and then clone Marco to create an object, Persona Carla, inheriting all of Marco's characteristics, and subsequently customize the Persona object by changing its name and address, we will notice that the name and the address has been cloned correctly, the address has not.

In fact, by changing Carla's house number, we also change Marco's house number!.

Person MarcoRossi =

new Person("Rossi Marco",

new Address("via mazzini", 18));

var CarlaVerdi = MarcoRossi.Clone();

CarlaVerdi.Name = "Carla Verdi";

CarlaVerdi.Address.HouseNumber = 23; //house number of marco has changed!

This happens because a shallow copy has occurred, meaning only the Person object has been copied, while the Address object contained within it has not been copied and remains shared between both Carla and Marco objects.

In other words, the pointer to the Address object remains the same, so modifying an attribute of the Address object reflects the change in both Marco and Carla objects.

To appropriately clone the object, a deep copy needs to be applied. A deep copy recursively copies not only the main object but also all the objects contained within it.

Let's see how to address these issues.

To make the type of copy explicit and avoid ambiguity, we will create an interface named IDeepCopyable, which we will strongly type using generics (making it more efficient than the built-in Entity Framework interface that returns an Object and requires casting operations).

Let's create the interface:

public interface IDeepCopyable<T>

{

T DeepCopy();

}

So, we make interface implemented by both the internal Address object and the Person object:

public class Address : IDeepCopyable<Address>

{

public readonly string StreetName;

public int HouseNumber;

public Address(string streetName, int houseNumber)

{

StreetName = streetName;

HouseNumber = houseNumber;

}

public Address DeepCopy()

{

return new Address(StreetName, HouseNumber);

}

}

public class Person : IDeepCopyable<Person>

{

public string Name;

public Address Address;

public Person(string name, Address address)

{

Name = name;

Address = address;

}

public Person DeepCopy()

{

var copy = new Person("Mario Rossi",

Address.DeepCopy());

return copy;

}

}

Now, if we try to make a copy, we will notice that a deep copy is performed:

Person person1 =

new("Mario Rossi",

new Address("Via Mazzini", 15));

Person person2 = person1.DeepCopy();

person2.Address.HouseNumber = 45;

The civic number of person1 has not changed! It is still 15, while that of person2 is 45.

This proves that the assignment of civic numbers occurred on two different Address objects.

Each object type should be copied using a specific technique, excluding value types (numbers, dates, structures, etc.), and strings (which are immutable types and behave like value types) should simply be copied by assignment.

Arrays, for example, are copied using their own copy method:

Array.Copy()

To copy a dictionary, an efficient and concise method is through LINQ:

var d = new Dictionary<string, PrototypeDeepCopy.Address>();

d.Add("IndirizzoMario", new PrototypeDeepCopy.Address("Via Mazzini", 18));

var d2 = d.ToDictionary(x => x.Key, x => x.Value.DeepCopy());

When we think about complex object graphs, meaning objects composed of many objects, which in turn are composed of other objects, performing a deep copy, as discussed earlier, can become a cumbersome task.

Fortunately, there are techniques that allow us to obtain a deep copy of an object with just a few steps.

The first strategy for achieving a deep copy in a few steps is binary object serialization.

For example, let's create a generic extension method that applies binary serialization to a generic object T:

public static class CopierManagerExtension

{

public static T DeepCopy<T>(this T self)

{

using (var stream = new MemoryStream())

{

BinaryFormatter formatter = new BinaryFormatter();

formatter.Serialize(stream, self);

stream.Seek(0, SeekOrigin.Begin);

object copy = formatter.Deserialize(stream);

return (T)copy;

}

}

}

As we can see, the generic object T is first serialized and then deserialized.

This approach allows for a fast deep copy, but we should remember to mark all the classes we want to clone in this way with the [Serializable] attribute.

Alternatively, if we want to avoid marking all classes as Serializable, we can opt for XML serialization.

Let's add another extension method for our generic object:

public static T DeepCopyXml<T>(this T self)

{

using (var ms = new MemoryStream())

{

XmlSerializer xmlSerializer = new XmlSerializer(typeof(T));

xmlSerializer.Serialize(ms, self);

ms.Position = 0;

return (T)xmlSerializer.Deserialize(ms);

}

}

**Builder**

The Builder Pattern is used in software development to construct complex objects step by step.

It separates the construction of a complex object from its representation, allowing you to create different variations of the object using the same construction process.

This pattern is particularly useful when you have an object with many optional properties or configurations.

The Builder Pattern typically involves the following components:

Director: This is responsible for orchestrating the construction process. It works with the builder to build the complex object.

Builder: The abstract interface or base class for concrete builders. It defines methods for constructing parts of the complex object.

Concrete Builder: Implementations of the builder interface. Each concrete builder is responsible for creating a specific variant of the complex object.

Product: The final complex object that is constructed by the builder. It represents the result of the construction process.

Here's an example of how to implement the Builder Pattern in C#:

// Product

class Computer

{

public string CPU { get; set; }

public string RAM { get; set; }

public string Storage { get; set; }

public string GraphicsCard { get; set; }

public void ShowDetails()

{

Console.WriteLine($"CPU: {CPU}");

Console.WriteLine($"RAM: {RAM}");

Console.WriteLine($"Storage: {Storage}");

Console.WriteLine($"Graphics Card: {GraphicsCard}");

}

}

// Builder interface

interface IComputerBuilder

{

void SetCPU();

void SetRAM();

void SetStorage();

void SetGraphicsCard();

Computer Build();

}

// Concrete Builder

class GamingComputerBuilder : IComputerBuilder

{

private Computer computer = new Computer();

public void SetCPU()

{

computer.CPU = "Intel Core i9";

}

public void SetRAM()

{

computer.RAM = "32GB DDR4";

}

public void SetStorage()

{

computer.Storage = "1TB SSD";

}

public void SetGraphicsCard()

{

computer.GraphicsCard = "NVIDIA GeForce RTX 3080";

}

public Computer Build()

{

return computer;

}

}

// Director

class ComputerDirector

{

private IComputerBuilder builder;

public ComputerDirector(IComputerBuilder builder)

{

this.builder = builder;

}

public Computer ConstructComputer()

{

builder.SetCPU();

builder.SetRAM();

builder.SetStorage();

builder.SetGraphicsCard();

return builder.Build();

}

}

Here is how to use the code above:

// Create a director and a concrete builder

var director = new ComputerDirector(new GamingComputerBuilder());

// Construct a gaming computer

Computer gamingComputer = director.ConstructComputer();

// Display computer details

Console.WriteLine("Gaming Computer Details:");

gamingComputer.ShowDetails();

In this example, we have a Computer class representing the complex object, a ComputerDirector that orchestrates the construction process, a IComputerBuilder interface, and a GamingComputerBuilder as a concrete builder for creating a gaming computer.

The director instructs the builder on how to construct the computer, and you can create different types of computers by implementing different builders.

**Abstract factory**

An abstract factory is a design pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes.

The abstract factory pattern is used when you need to ensure that a group of related objects, but not necessarily a group, are created together and are compatible with each other.

It allows you to create instances of classes that implement a common interface or belong to a common family of classes, without specifying the exact class names; instead, you work with abstract interfaces and factories to create objects.

The abstract factory interface that declares factory methods for objects gives the name to the pattern (Abstract factory), this factory needs a concrete implementation!.

Here is an example:

// Abstract Product: Button

interface IButton

{

void Render();

}

// Concrete Products for Windows OS

class WindowsButton : IButton

{

public void Render()

{

Console.WriteLine("Render a Windows button");

}

}

// Concrete Products for macOS

class MacOSButton : IButton

{

public void Render()

{

Console.WriteLine("Render a macOS button");

}

}

// Abstract Product: Checkbox

interface ICheckbox

{

void Render();

}

// Concrete Products for Windows OS

class WindowsCheckbox : ICheckbox

{

public void Render()

{

Console.WriteLine("Render a Windows checkbox");

}

}

// Concrete Products for macOS

class MacOSCheckbox : ICheckbox

{

public void Render()

{

Console.WriteLine("Render a macOS checkbox");

}

}

// Abstract Factory

interface IUIFactory

{

IButton CreateButton();

ICheckbox CreateCheckbox();

}

// Concrete Factory for Windows OS

class WindowsUIFactory : IUIFactory

{

public IButton CreateButton()

{

return new WindowsButton();

}

public ICheckbox CreateCheckbox()

{

return new WindowsCheckbox();

}

}

// Concrete Factory for macOS

class MacOSUIFactory : IUIFactory

{

public IButton CreateButton()

{

return new MacOSButton();

}

public ICheckbox CreateCheckbox()

{

return new MacOSCheckbox();

}

}

Here is how you can use the pattern:

// Create a Windows UI Factory

IUIFactory windowsFactory = new WindowsUIFactory();

// Create Windows UI components

IButton windowsButton = windowsFactory.CreateButton();

ICheckbox windowsCheckbox = windowsFactory.CreateCheckbox();

windowsButton.Render();

windowsCheckbox.Render();

// Create a macOS UI Factory

IUIFactory macFactory = new MacOSUIFactory();

// Create macOS UI components

IButton macButton = macFactory.CreateButton();

ICheckbox macCheckbox = macFactory.CreateCheckbox();

macButton.Render();

macCheckbox.Render();

In this example:

We have abstract product interfaces IButton and ICheckbox, which define the methods for rendering these UI components.

Concrete product classes (WindowsButton, WindowsCheckbox, MacOSButton, MacOSCheckbox) implement these interfaces to provide platform-specific implementations.

The abstract factory interface IUIFactory declares factory methods for creating IButton and ICheckbox objects.

Concrete factory classes (WindowsUIFactory and MacOSUIFactory) implement the IUIFactory interface to create platform-specific UI components.

In the Main method, we create instances of the concrete factories (windowsFactory and macFactory) and use them to create UI components.

This allows us to create Windows or macOS UI components without knowing the concrete classes used.

When you run the program, it will create and render UI components based on the factory chosen (Windows or macOS), demonstrating how the Abstract Factory pattern decouples the client code from the specific UI implementations.

**Structural patterns.**

Structural patterns are used to optimize the structure of one's application so that it is more easily maintainable and the code is usable in the simplest way possible.

**Adapter**

The adapter pattern is used whenever there is a need to provide a compatible interface to adapt a specific object for use in a particular context. Suppose we want to create a library of filters to modify images. It is necessary to build a viewer for previewing the filters. This viewer uses an apply method that applies a specific filter to the image.

class ImageView

{

private Image image;

public ImageView(Image image)

{

this.image = image;

}

public void Apply(IFilter filter)

{

filter.Apply(image);

}

}

The filter will be used by the viewer by applying the Dependency Inversion Principle, decoupling the viewer class from the concrete implementation of the filter (SoftFilter) through the abstraction entity (IFilter).

interface IFilter

{

void Apply(Image img);

}

class SoftFilter : IFilter

{

public void Apply(Image img)

{

Console.WriteLine("apply soft filter");

}

}

Now, however, let's suppose we want to use a third-party filter (Mosaic) whose interface is not compatible. If we try to force the built-in interface to be adopted by our code, we will encounter an error:

ImageView imageView = new ImageView(new Adapter.Image());

imageView.Apply(new SoftFilter());

imageView.Apply(new ThirdParts.Mosaic()); // error!

The solution involves generating an adapter that adapts the interface used by our application to apply filters with the built-in interface exposed by our third-party component.

class MosaicAdapter : IFilter

{

private Mosaic mosaic;

public MosaicAdapter(Mosaic mosaic)

{

this.mosaic = mosaic;

}

public void Apply(Image img)

{

mosaic.init();

mosaic.Render(img);

}

}

We have encapsulated in the Apply method of our interface the behavior required by the third-party filter to apply the effect to the image. At this point, the filter can be applied in the same way as SoftFilter was applied:

ImageView imageView = new StructuralPatterns.Adapter.ImageView(new Image());

imageView.Apply(new StructuralPatterns.Adapter.SoftFilter());

imageView.Apply(new StructuralPatterns.Adapter.MosaicAdapter(new StructuralPatterns.Adapter.ThirdParts.Mosaic()));//apply adapter to adapt a third parts component call

**Command**

The Command Pattern aims to encapsulate a request (an action or operation) as an object, allowing you to parameterize clients with different requests, queue or log requests, and support undoable operations.

Here's a simple and clear explanation:

**Encapsulation**: It wraps a request (like turning on a light or saving a file) into an object called a "command." This object contains all the information needed to perform the request.

**Parameterization**: Clients (the code that initiates the request) can be given different command objects to perform various tasks. Think of it like giving a remote control multiple buttons, each corresponding to a different command.

**Queuing and Logging**: You can store and manage these command objects, allowing you to create queues of commands or even log them for future use.

**Undo/Redo**: The Command Pattern often supports undoing and redoing actions by storing enough information in the command objects to reverse the action when needed.

In summary, the Command Pattern helps you make your code more flexible and extensible by turning requests into objects. It's like a remote control for your code, where you can push different buttons (commands) to make things happen, and you can even rewind (undo) and fast forward (redo) if needed.

In simple terms, it decouples the sender (the one who initiates the action) from the receiver (the one who performs the action).

It does this by encapsulating a request as an object, which allows you to parameterize clients with different requests, queue or log requests, and even undo/redo operations.

Here is a simple example:

// Command interface

public interface ICommand

{

void Execute();

void Undo();

}

// Receiver

public class Light

{

public void TurnOn()

{

Console.WriteLine("Light is on");

}

public void TurnOff()

{

Console.WriteLine("Light is off");

}

}

// ConcreteCommand

public class TurnOnLightCommand : ICommand

{

private readonly Light \_light;

public TurnOnLightCommand(Light light)

{

\_light = light;

}

public void Execute()

{

\_light.TurnOn();

}

public void Undo()

{

\_light.TurnOff();

}

}

// ConcreteCommand

public class TurnOffLightCommand : ICommand

{

private readonly Light \_light;

public TurnOffLightCommand(Light light)

{

\_light = light;

}

public void Execute()

{

\_light.TurnOff();

}

public void Undo()

{

\_light.TurnOn();

}

}

// Invoker

public class RemoteControl

{

private readonly List<ICommand> \_commands = new List<ICommand>();

public void AddCommand(ICommand command)

{

\_commands.Add(command);

}

public void PressButton(int index)

{

if (index >= 0 && index < \_commands.Count)

{

\_commands[index].Execute();

}

}

public void UndoButton(int index)

{

if (index >= 0 && index < \_commands.Count)

{

\_commands[index].Undo();

}

}

}

Here is how you can use command pattern from client:

Light light = new Light();

ICommand turnOnCommand = new TurnOnLightCommand(light);

ICommand turnOffCommand = new TurnOffLightCommand(light);

RemoteControl remote = new RemoteControl();

remote.AddCommand(turnOnCommand);

remote.AddCommand(turnOffCommand);

remote.PressButton(0); // Turn on the light

remote.PressButton(1); // Turn off the light

remote.UndoButton(1); // Undo turning off (turn on again)

In this example:

The ICommand interface defines the Execute and Undo methods that all concrete commands must implement.

The Light class is the receiver of the commands. It has methods to perform the actions we want to encapsulate.

TurnOnLightCommand and TurnOffLightCommand are concrete command classes that encapsulate the actions to turn the light on and off.

RemoteControl acts as the invoker, which stores and executes the commands.

We create instances of the commands, add them to the remote control, and then use the remote control to execute and undo the commands.

This example demonstrates how the Command Pattern separates the sender (remote control) from the receiver (light) and allows for flexible and extensible command handling.

**Bridge**

The bridge pattern allows separating the interface of a class from its implementation.

Through the bridge pattern, we achieve a code structure that is easier to manage.

If an abstraction with various implementations usually, using inheritance inevitably leads to merging the abstraction with the implementation itself.

This means two things: the proliferation of classes and duplication of code.

The bridge pattern solves these problems by aiming for linear growth of classes and code.

Let's take an example.

Suppose we want to develop the logic for the operation of remote controls.

We need to implement two types of remote controls, a basic remote control and an advanced one.

If we use inheritance (the most common and natural choice), we could proceed as follows:

Define an abstract class RemoteControl that outlines the functionality of the basic remote control without implementing it:

public abstract class RemoteControl

{

public abstract void TurnOn();

public abstract void TurnOff();

}

So, define another abstract class that inherits from the previous one and outlines (without implementing them) additional advanced functionalities:

public abstract class AdvanceRemoteControl : RemoteControl

{

public abstract void SetChannel(int Number);

}

Now, if we go on to implement the specific functionalities for the Sony remote control, we will need to create two concrete classes.

These classes will inherit from RemoteControl and AdvanceRemoteControl respectively.

Through the override keyword, we redefine and implement the method signatures defined in the inherited abstract classes:

class SonyRemoteControl : RemoteControl

{

public override void TurnOff()

{

Console.WriteLine("Sony: TurnOff");

}

public override void TurnOn()

{

Console.WriteLine("Sony: TurnOn");

}

}

class SonyAdvancedRemoteControl : AdvanceRemoteControl

{

public override void SetChannel(int Number)

{

Console.WriteLine("Sony: TurnOn setchannel" + Number);

}

public override void TurnOff()

{

Console.WriteLine("Sony: TurnOff");

}

public override void TurnOn()

{

Console.WriteLine("Sony: TurnOn");

}

}

By doing this, two things become evident.

First, the redundancy of code (e.g., turnOff is implemented twice, both in RemoteControl and AdvanceRemoteControl).

Second, the proliferation of concrete classes. In fact, if we had to implement the Philips remote control, we would need to create two new classes with the resulting duplicated code.

The bridge pattern represents a better solution that can be more easily maintained.

It involves decoupling the concrete implementation of functionalities from the classes that define their abstract interface.

This is achieved through a bridge (hence the name of the pattern).

Let's now rewrite the code for implementing the remote controls using the bridge pattern.

The bridge consists of an interface that defines the functionalities of the remote control:

public interface Device

{

void TurnOn();

void TurnOff();

void SetChannel(int number);

}

The bridge is injected, through composition, into the abstract interface:

public class RemoteControl

{

protected Device device;

public RemoteControl(Device device)

{

this.device = device;

}

public void TurnOn()

{

device.TurnOn();

}

public void TurnOff()

{

device.TurnOff();

}

}

public class AdvanceRemoteControl : RemoteControl

{

public AdvanceRemoteControl(Device device)

: base(device)

{

this.device = device;

}

public void SetChannel(int number)

{

device.SetChannel(number);

}

}

The same bridge is inherited by the concrete class, which develops the functionalities of our remote control:

class SonyTv : Device

{

public void SetChannel(int number)

{

Console.WriteLine("Sony: set channel $number" + number);

}

public void TurnOff()

{

Console.WriteLine("Sony: TurnOff");

}

public void TurnOn()

{

Console.WriteLine("Sony: TurnOn");

}

}

Here, Device, so to speak, has one foot in the "abstract interface" and one in the concrete class, configuring itself as a bridge.

Notice two things: the absence of redundant code and the fact that, in case we need to implement a new concrete command such as Philips, we won't have to add as many classes as there are abstractions as before. Instead, we just need to create a new class, let's say PhilipsTv, which inherits from Device and implements its specific functionalities.

At this point, if we want to use our Sony RemoteControl, we would write this code:

RemoteControl RemoteControl = new(new SonyTv());

RemoteControl.TurnOn();

While to use the advanced interface of the AdvanceRemoteControl remote control:

AdvanceRemoteControl advanceRemoteControl = new AdvanceRemoteControl(new SonyTv());

advanceRemoteControl.SetChannel(5);

**Composite**

The primary goal of the Composite pattern is to treat both individual objects and compositions of objects uniformly.

In other words, it lets you compose objects into tree structures to represent part-whole hierarchies.

Here's a breakdown of the key components in the Composite pattern:

Component:

This is the abstract class or interface that declares the common interface for all concrete classes (both leaf and composite).

It defines the operations that can be performed on both individual objects and compositions.

Leaf:

This is the concrete class that implements the Component interface.

It represents the individual objects in the composition, the "leaf" nodes of the tree structure.

Composite:

This is also a concrete class that implements the Component interface.

It represents the compositions or containers of objects. Composites can contain both individual objects (leaves) and other composites.

The Composite pattern allows clients to treat individual objects and compositions of objects uniformly.

This simplifies the client code, as it doesn't need to distinguish between different types of objects in the composite structure.

Whether it's a leaf or a composite, the client interacts with them through the same interface defined by the Component.

A classic example of where the Composite pattern is often applied is in the construction of graphical user interfaces (GUIs) where a user interface element (like a window or a button) can contain other UI elements, forming a hierarchical structure.

The Composite pattern promotes a consistent and uniform treatment of objects, making it easier to add new types of components to the system without changing the client code.

Let's create a simple example to represent a GUI hierarchy using the Composite pattern. We'll have a Window class as the composite, and Button and TextBox classes as leaves.

// Component interface

public abstract class UIComponent

{

public abstract void Display();

}

// Leaf class representing a Button

public class Button : UIComponent

{

private string label;

public Button(string label)

{

this.label = label;

}

public override void Display()

{

Console.WriteLine($"Button: {label}");

}

}

// Leaf class representing a TextBox

public class TextBox : UIComponent

{

private string content;

public TextBox(string content)

{

this.content = content;

}

public override void Display()

{

Console.WriteLine($"TextBox: {content}");

}

}

// Composite class representing a Window

public class Window : UIComponent

{

private List<UIComponent> components = new List<UIComponent>();

public void AddComponent(UIComponent component)

{

components.Add(component);

}

public override void Display()

{

Console.WriteLine("Window:");

foreach (var component in components)

{

component.Display();

}

}

}

In this example, we have a UIComponent abstract class representing the Component interface.

Button and TextBox are leaf classes that implement this interface.

The Window class is the composite class that can contain other UI components (both leaves and other composites).

In the Main method, we create instances of Button, TextBox, and Window, add buttons and a text box to the window, and then display the window.

The Display method of the window recursively calls the Display method on its child components, resulting in the entire hierarchy being displayed.

var button1 = new Button("OK");

var button2 = new Button("Cancel");

var textBox = new TextBox("Type something here");

// Creating composite element

var window = new Window();

window.AddComponent(button1);

window.AddComponent(button2);

window.AddComponent(textBox);

// Displaying the composite (window) which in turn displays its children

window.Display();

**Decorator**

The Decorator Pattern allows behavior to be added to an individual object, either statically or dynamically, without affecting the behavior of other objects from the same class.

It is often used to extend or augment the functionalities of classes in a flexible and reusable way.

In the Decorator Pattern, you have a set of decorator classes that are used to wrap concrete components.

These decorators have the same interface as the components they decorate, allowing them to be used interchangeably.

The key idea is to attach additional responsibilities to an object dynamically.

This is achieved by creating a set of decorator classes that are used to wrap concrete components.

Here's an example to illustrate the Decorator Pattern:

// Component interface

public interface ICoffee

{

int Cost();

}

// Concrete component

public class SimpleCoffee : ICoffee

{

public int Cost()

{

return 5;

}

}

// Decorator

public abstract class CoffeeDecorator : ICoffee

{

private readonly ICoffee \_coffee;

protected CoffeeDecorator(ICoffee coffee)

{

\_coffee = coffee;

}

public virtual int Cost()

{

return \_coffee.Cost();

}

}

// Concrete decorators

public class MilkDecorator : CoffeeDecorator

{

public MilkDecorator(ICoffee coffee) : base(coffee)

{

}

public override int Cost()

{

return base.Cost() + 2;

}

}

public class SugarDecorator : CoffeeDecorator

{

public SugarDecorator(ICoffee coffee) : base(coffee)

{

}

public override int Cost()

{

return base.Cost() + 1;

}

}

In this example, Coffee is the component interface, SimpleCoffee is the concrete component, and MilkDecorator and SugarDecorator are the decorators.

Decorators can be stacked or combined in various ways to create different combinations of behavior for the objects they decorate.

Here is code to use classes above:

ICoffee coffee = new SimpleCoffee();

Console.WriteLine("Cost of simple coffee: " + coffee.Cost());

coffee = new MilkDecorator(coffee);

Console.WriteLine("Cost of milk coffee: " + coffee.Cost());

coffee = new SugarDecorator(coffee);

Console.WriteLine("Cost of sugar and milk coffee: " + coffee.Cost());

**Facade**

The Facade pattern provides a simplified interface to a set of interfaces in a subsystem, making it easier to use and understand.

It involves creating a unified interface that hides the complexities of a subsystem and provides a higher-level interface for clients.

In software development, systems often consist of multiple subsystems with their own set of classes and interactions.

The Facade pattern aims to simplify the usage of these subsystems by providing a single, simplified interface.

This can be especially useful when the subsystems are complex or have a large number of components, and it becomes challenging for clients to interact with them directly.

Key components of the Facade pattern include:

**Facade:** This is the class or interface that clients interact with. It encapsulates the complexity of the subsystem by providing a simplified interface.

The facade delegates client requests to the appropriate objects within the subsystem.

**Subsystem Classes:** These are the individual classes or components that make up the subsystem.

They contain the actual implementation of the functionality, but their complexities are hidden from clients by the facade.

Here's a simple analogy to understand the Facade pattern:

Imagine a complex electronic device, such as a television.

The remote control acts as a facade for the television system.

Instead of dealing with the intricate details of turning on the TV, adjusting the volume, changing channels, and managing settings individually, users interact with the remote control, which provides a simplified interface. The remote control serves as a facade, hiding the complexity of the TV system.

In software development, the Facade pattern promotes a clean separation between subsystems and clients, making the codebase more maintainable and easier to comprehend.

It also helps in reducing dependencies between clients and subsystems, as clients interact only with the facade rather than the individual components of the subsystem.

Let's consider an example where we have a complex subsystem with multiple classes, and we'll create a facade to provide a simplified interface for clients.

In this example, we'll imagine a multimedia system with various components like the DVD player, the audio system, and the projector.

// Subsystem Classes

class DVDPlayer

{

public void TurnOn()

{

Console.WriteLine("DVD Player is turned on");

}

public void PlayMovie(string movie)

{

Console.WriteLine($"Playing movie: {movie}");

}

}

class AudioSystem

{

public void TurnOn()

{

Console.WriteLine("Audio System is turned on");

}

public void SetVolume(int volume)

{

Console.WriteLine($"Setting volume to {volume}");

}

}

class Projector

{

public void TurnOn()

{

Console.WriteLine("Projector is turned on");

}

public void Project()

{

Console.WriteLine("Projecting on the screen");

}

}

// Facade

class HomeTheaterFacade

{

private DVDPlayer dvdPlayer;

private AudioSystem audioSystem;

private Projector projector;

public HomeTheaterFacade()

{

dvdPlayer = new DVDPlayer();

audioSystem = new AudioSystem();

projector = new Projector();

}

public void WatchMovie(string movie)

{

Console.WriteLine("Get ready to watch a movie...");

dvdPlayer.TurnOn();

audioSystem.TurnOn();

projector.TurnOn();

dvdPlayer.PlayMovie(movie);

audioSystem.SetVolume(10);

projector.Project();

}

public void EndMovie()

{

Console.WriteLine("Shutting down the home theater...");

dvdPlayer.TurnOn(); // Assuming there's a method to turn off each component

audioSystem.TurnOn();

projector.TurnOn();

}

}

In this example, the HomeTheaterFacade class serves as the facade, providing a simplified interface for watching a movie and ending the movie.

Clients can interact with the home theater system without needing to know the details of each subsystem component.

The facade encapsulates the complexity of turning on/off the DVD player, audio system, and projector, making the code more manageable and understandable.

Here is the client code:

// Using the Facade to simplify interaction with the subsystem

HomeTheaterFacade homeTheater = new HomeTheaterFacade();

homeTheater.WatchMovie("Inception");

// After watching the movie, you can use the facade to clean up

homeTheater.EndMovie();

**Flyweight**

The Flyweight Pattern is used to minimize memory usage or computational expenses by sharing as much as possible with related objects; it is a way to use objects in large numbers when a simple repeated representation would use an unacceptable amount of memory.

The key idea behind the Flyweight Pattern is to separate the intrinsic (shared) and extrinsic (unique) parts of an object.

The intrinsic state is the part that can be shared among multiple objects, while the extrinsic state is the part that varies and must be stored separately for each object.

Here are the main components of the Flyweight Pattern:

**Flyweight Interface/Abstract Class:** This defines the interface for the flyweight objects. It usually includes a method to receive and act on the extrinsic state.

**ConcreteFlyweight:** This is a concrete implementation of the flyweight interface and represents the shared intrinsic state. Objects of this class are shared and reused.

**UnsharedConcreteFlyweight:** This class represents the unshared flyweight. Instances of this class have intrinsic state that cannot be shared.

**FlyweightFactory:** This is a factory class that manages the creation and sharing of flyweight objects. It ensures that flyweight objects are shared and not duplicated unnecessarily. It often includes a mechanism to retrieve an existing flyweight or create a new one if it doesn't exist.

**Client:** The client is responsible for maintaining the extrinsic state of the flyweight objects. It uses the flyweight objects but does not need to know about the internal details of how they are shared.

Here's an example to illustrate the concept.

Let's say you have a text editor, and you want to represent characters in the text efficiently.

Instead of creating a separate object for each character, you can use a flyweight pattern.

The intrinsic state would be the character itself, and the extrinsic state would be the position of the character in the text.

By sharing the intrinsic state (characters) among multiple positions, you can reduce memory usage.

The Flyweight Pattern is particularly useful in situations where a large number of similar objects need to be created and memory usage is a concern.

It's important to note that this pattern is most effective when the shared (intrinsic) state is considerable and the extrinsic state is relatively small.

Below is a simple example of a text editor using the Flyweight Pattern:

//Flyweight interface

interface ICharacter

{

void Display(int position);

}

//ConcreteFlyweight

class Character : ICharacter

{

private char \_symbol;

public Character(char symbol)

{

\_symbol = symbol;

}

public void Display(int position)

{

Console.WriteLine($"Character {\_symbol} at position {position}");

}

}

//FlyweightFactory

class CharacterFactory

{

private Dictionary<char, ICharacter> \_characters = new Dictionary<char, ICharacter>();

public ICharacter GetCharacter(char symbol)

{

if (!\_characters.ContainsKey(symbol))

{

\_characters[symbol] = new Character(symbol);

}

return \_characters[symbol];

}

}

//Client

class TextEditor

{

private List<Tuple<int, char>> \_characters = new List<Tuple<int, char>>();

private CharacterFactory \_characterFactory = new CharacterFactory();

public void InsertCharacter(char symbol, int position)

{

ICharacter character = \_characterFactory.GetCharacter(symbol);

\_characters.Add(new Tuple<int, char>(position, symbol));

character.Display(position);

}

public void DisplayText()

{

foreach (var tuple in \_characters)

{

Console.WriteLine($"Character {tuple.Item2} at position {tuple.Item1}");

}

}

}

In this example:

ICharacter is the Flyweight interface.

Character is the ConcreteFlyweight representing a character in the text.

CharacterFactory is the FlyweightFactory responsible for creating and managing character objects.

TextEditor is the Client that inserts characters into the text and displays the text.

The key idea is that characters are shared through the CharacterFactory, and the TextEditor maintains the extrinsic state (position) for each character.

This way, the memory usage is minimized, and characters are reused when they already exist in the factory.

Here is the code to use the example above:

TextEditor textEditor = new TextEditor();

//Insert characters into the text editor

textEditor.InsertCharacter('H', 0);

textEditor.InsertCharacter('e', 1);

textEditor.InsertCharacter('l', 2);

textEditor.InsertCharacter('l', 3);

textEditor.InsertCharacter('o', 4);

//Display the text

textEditor.DisplayText();

**Behavioural patterns.**

Behavioural patterns are design patterns that deal with object collaboration and delegation of responsibilities.

These patterns focus on defining a set of algorithms, encapsulating each algorithm, and making them interchangeable.

Behavioral patterns help in defining how objects interact and communicate with each other.

**Interpreter**

The Interpreter Design Pattern is used to define a grammar for a language and provides a way to evaluate and interpret expressions written in that language.

It is commonly used in scenarios where you have a language to parse and execute, such as a mathematical expression parser or a query language interpreter.

Here's a simplified explanation of the Interpreter Pattern components:

**Abstract Expression**: Defines an abstract interface for interpreting expressions. This interface typically includes an Interpret method.

**Terminal Expression**: Implements the AbstractExpression interface for terminal symbols in the grammar. Terminal expressions represent the lowest-level elements in the language.

**Non-Terminal Expression**: Implements the AbstractExpression interface for non-terminal symbols in the grammar. Non-terminal expressions combine one or more terminal and non-terminal expressions to form more complex expressions.

**Context**: Contains information that needs to be interpreted. It can also maintain a stack or other data structures to manage the parsing and evaluation of expressions.

**Client**: Builds the abstract syntax tree of the expressions and invokes the interpreter to evaluate them.

Here is a simple example:

// Abstract Expression

public interface IExpression

{

int Interpret(Dictionary<string, int> context);

}

// Terminal Expression

public class NumberExpression : IExpression

{

private readonly int \_number;

public NumberExpression(int number)

{

\_number = number;

}

public int Interpret(Dictionary<string, int> context)

{

return \_number;

}

}

// Non-Terminal Expression

public class AddExpression : IExpression

{

private readonly IExpression \_left;

private readonly IExpression \_right;

public AddExpression(IExpression left, IExpression right)

{

\_left = left;

\_right = right;

}

public int Interpret(Dictionary<string, int> context)

{

return \_left.Interpret(context) + \_right.Interpret(context);

}

}

// Non-Terminal Expression for subtraction

public class SubtractExpression : IExpression

{

private readonly IExpression \_left;

private readonly IExpression \_right;

public SubtractExpression(IExpression left, IExpression right)

{

\_left = left;

\_right = right;

}

public int Interpret(Dictionary<string, int> context)

{

return \_left.Interpret(context) - \_right.Interpret(context);

}

}

// Terminal Expression for variables

public class VariableExpression : IExpression

{

private readonly string \_variableName;

public VariableExpression(string variableName)

{

\_variableName = variableName;

}

public int Interpret(Dictionary<string, int> context)

{

if (context.ContainsKey(\_variableName))

{

return context[\_variableName];

}

else

{

throw new InvalidOperationException($"Variable {\_variableName} not found in the context.");

}

}

}

Here is and client code:

// Context with variable values

var context = new Dictionary<string, int>

{

{ "a", 5 },

{ "b", 3 }

};

// Create the expression tree: a + b - 2

IExpression expression = new AddExpression(new VariableExpression("a"), new VariableExpression("b"));

expression = new SubtractExpression(expression, new NumberExpression(2));

// Interpret the expression

int result = expression.Interpret(context);

Console.WriteLine("Result: " + result); // Output: Result: 6

In this example, we have created a simple expression language that can evaluate arithmetic expressions involving addition, subtraction, and variables.

The Interpret method recursively evaluates the expression tree, considering the context of variables.

Let's consider another practical example of the Interpreter Pattern in C#.

We'll create a simple date expression language that can interpret date-related expressions, such as adding or subtracting days from a given date.

Here below the example:

// Abstract Expression

public interface IDateExpression

{

DateTime Interpret(DateTime context);

}

// Terminal Expression for a specific date

public class DateExpression : IDateExpression

{

private readonly DateTime \_date;

public DateExpression(DateTime date)

{

\_date = date;

}

public DateTime Interpret(DateTime context)

{

return \_date;

}

}

// Non-Terminal Expression for adding days

public class AddDaysExpression : IDateExpression

{

private readonly IDateExpression \_expression;

private readonly int \_daysToAdd;

public AddDaysExpression(IDateExpression expression, int daysToAdd)

{

\_expression = expression;

\_daysToAdd = daysToAdd;

}

public DateTime Interpret(DateTime context)

{

var baseDate = \_expression.Interpret(context);

return baseDate.AddDays(\_daysToAdd);

}

}

// Non-Terminal Expression for subtracting days

public class SubtractDaysExpression : IDateExpression

{

private readonly IDateExpression \_expression;

private readonly int \_daysToSubtract;

public SubtractDaysExpression(IDateExpression expression, int daysToSubtract)

{

\_expression = expression;

\_daysToSubtract = daysToSubtract;

}

public DateTime Interpret(DateTime context)

{

var baseDate = \_expression.Interpret(context);

return baseDate.AddDays(-\_daysToSubtract);

}

}

Here is the client side code:

// Context: a specific date

var context = new DateTime(2023, 10, 1);

// Create the expression: (context - 7) + 14

IDateExpression expression = new SubtractDaysExpression(new AddDaysExpression(new DateExpression(context), -7), 14);

// Interpret the expression

DateTime result = expression.Interpret(context);

Console.WriteLine("Result: " + result.ToString("yyyy-MM-dd")); // Output: Result: 2023-10-8

Let's consider another practical example.

We'll create a simple date expression language that can interpret date-related expressions, such as adding or subtracting days from a given date.

// Abstract Expression

public interface IDateExpression

{

DateTime Interpret(DateTime context);

}

// Terminal Expression for a specific date

public class DateExpression : IDateExpression

{

private readonly DateTime \_date;

public DateExpression(DateTime date)

{

\_date = date;

}

public DateTime Interpret(DateTime context)

{

return \_date;

}

}

// Non-Terminal Expression for adding days

public class AddDaysExpression : IDateExpression

{

private readonly IDateExpression \_expression;

private readonly int \_daysToAdd;

public AddDaysExpression(IDateExpression expression, int daysToAdd)

{

\_expression = expression;

\_daysToAdd = daysToAdd;

}

public DateTime Interpret(DateTime context)

{

var baseDate = \_expression.Interpret(context);

return baseDate.AddDays(\_daysToAdd);

}

}

// Non-Terminal Expression for subtracting days

public class SubtractDaysExpression : IDateExpression

{

private readonly IDateExpression \_expression;

private readonly int \_daysToSubtract;

public SubtractDaysExpression(IDateExpression expression, int daysToSubtract)

{

\_expression = expression;

\_daysToSubtract = daysToSubtract;

}

public DateTime Interpret(DateTime context)

{

var baseDate = \_expression.Interpret(context);

return baseDate.AddDays(-\_daysToSubtract);

}

}

Here below the client code:

// Context: a specific date

var context = new DateTime(2023, 10, 1);

// Create the expression: (context - 7) + 14

IDateExpression expression = new SubtractDaysExpression(new AddDaysExpression(new DateExpression(context), -7), 14);

// Interpret the expression

DateTime result = expression.Interpret(context);

Console.WriteLine("Result: " + result.ToString("yyyy-MM-dd")); // Output: Result: 2023-10-8

In this example, we've created a date expression language.

You can create date expressions to add or subtract days from a given date, and the Interpreter Pattern is used to interpret and calculate the final date result.

This example demonstrates how you can apply the Interpreter Pattern to a different domain, in this case, working with dates and date expressions.

**Mediator.**

The Mediator pattern is a pattern that promotes loose coupling between components in a software system by centralizing communication between them.

It is often used to simplify complex communication between multiple objects or classes.

In this pattern, a mediator object encapsulates the interactions between these objects, allowing them to communicate without being directly aware of each other.

Here's a description of the Mediator pattern and an example in C#:

**Mediator Pattern Components:**

**Mediator:** This is an interface or abstract class that defines the contract for communication between various components (colleagues).

**Concrete Mediator:** This is the actual implementation of the mediator interface. It manages the interactions and relationships between the colleagues.

**Colleague:** These are the individual components that need to communicate with each other. Colleagues are typically unaware of each other and communicate through the mediator.

**Concrete Colleagues:** These are the specific implementations of colleagues. They rely on the mediator to coordinate their interactions.

Here below a practical example:

Let's say we have a chat application where users can send messages to each other. We can use the Mediator pattern to facilitate communication between users without them knowing about each other.

// Mediator interface

public interface IChatMediator

{

void SendMessage(string message, IUser user);

}

// Concrete Mediator

public class ChatMediator : IChatMediator

{

private List<IUser> users = new List<IUser>();

public void AddUser(IUser user)

{

users.Add(user);

}

public void SendMessage(string message, IUser sender)

{

foreach (var user in users)

{

// Don't send the message back to the sender

if (user != sender)

{

user.ReceiveMessage(message);

}

}

}

}

// Colleague interface

public interface IUser

{

void SendMessage(string message);

void ReceiveMessage(string message);

}

// Concrete Colleague

public class ChatUser : IUser

{

private readonly IChatMediator mediator;

private string name;

public ChatUser(IChatMediator mediator, string name)

{

this.mediator = mediator;

this.name = name;

}

public void SendMessage(string message)

{

Console.WriteLine($"{name} sends message: {message}");

mediator.SendMessage(message, this);

}

public void ReceiveMessage(string message)

{

Console.WriteLine($"{name} receives message: {message}");

}

}

Here the client side code:

IChatMediator chatMediator = new ChatMediator();

IUser user1 = new ChatUser(chatMediator, "User 1");

IUser user2 = new ChatUser(chatMediator, "User 2");

IUser user3 = new ChatUser(chatMediator, "User 3");

chatMediator.AddUser(user1);

chatMediator.AddUser(user2);

chatMediator.AddUser(user3);

user1.SendMessage("Hello, everyone!");

user2.SendMessage("Hi, User 1!");

In this example, the ChatMediator manages communication between ChatUser instances.

Users send messages through the mediator, and the mediator forwards the messages to other users.

This allows users to communicate without needing to know about each other directly, demonstrating the Mediator pattern's use for decoupling components in a system.

**Iterator.**

The Iterator Pattern is used to provide a way to access elements of an aggregate object (such as a collection) sequentially without exposing the underlying representation of the object.

It allows you to iterate over the elements of a collection without needing to know the internal details of how the collection is structured.

Here's how the Iterator Pattern works:

**Iterator Interface (or Abstract Class):** This defines the interface for accessing and traversing the elements of the collection. It typically includes methods like Next(), HasNext(), and CurrentItem().

**Concrete Iterator:** This class implements the Iterator interface and keeps track of the current position within the collection. It provides the actual implementation for iterating over the collection.

**Aggregate (Collection):** This is the collection of objects that you want to iterate over. It may also provide a method to create an Iterator object.

**Concrete Aggregate:** This class implements the Aggregate interface and provides the concrete collection of objects. It also has a method that creates a Concrete Iterator for the collection.

Let’s see an example:

// Step 1: Create the Iterator interface

public interface IIterator<T>

{

bool HasNext();

T Next();

}

// Step 2: Create the Concrete Iterator

public class StudentIterator : IIterator<Student>

{

private List<Student> \_students;

private int \_position = 0;

public StudentIterator(List<Student> students)

{

\_students = students;

}

public bool HasNext()

{

return \_position < \_students.Count;

}

public Student Next()

{

if (HasNext())

{

Student nextStudent = \_students[\_position];

\_position++;

return nextStudent;

}

else

{

return null;

}

}

}

// Step 3: Create the Aggregate interface

public interface IAggregate<T>

{

IIterator<T> CreateIterator();

}

// Step 4: Create the Concrete Aggregate

public class StudentList : IAggregate<Student>

{

private List<Student> \_students = new List<Student>();

public void AddStudent(Student student)

{

\_students.Add(student);

}

public IIterator<Student> CreateIterator()

{

return new StudentIterator(\_students);

}

}

// Student class representing individual student records

public class Student

{

public string Name { get; set; }

public int StudentID { get; set; }

public Student(string name, int studentID)

{

Name = name;

StudentID = studentID;

}

}

Here the client side code:

StudentList studentList = new StudentList();

studentList.AddStudent(new Student("Alice", 1001));

studentList.AddStudent(new Student("Bob", 1002));

studentList.AddStudent(new Student("Charlie", 1003));

IIterator<Student> iterator = studentList.CreateIterator();

while (iterator.HasNext())

{

Student student = iterator.Next();

Console.WriteLine($"Student Name: {student.Name}, Student ID: {student.StudentID}");

}

In this revised example, we've created a student management system where we can add student records to the StudentList collection and use the StudentIterator to iterate through these records.

The Iterator Pattern allows us to iterate over the students without exposing the internal structure of the StudentList class, providing a clean way to manage and access student data.

By the way this pattern permit you to change the iterable object without affecting the client code, infact here is used a list, but it's possible to replace it with an array or other.

This code could be used only to iterate through iterable of type student, let's generalize it using generic.

Here below an example:

//the Iterator interface

public interface IIterator<T>

{

bool HasNext();

T Next();

}

//the Concrete Iterator

public class BookIterator : IIterator<Book>

{

private List<Book> \_books;

private int \_position = 0;

public BookIterator(List<Book> books)

{

\_books = books;

}

public bool HasNext()

{

return \_position < \_books.Count;

}

public Book Next()

{

if (HasNext())

{

Book nextBook = \_books[\_position];

\_position++;

return nextBook;

}

else

{

return null;

}

}

}

//the Aggregate interface

public interface IAggregate<T>

{

IIterator<T> CreateIterator();

}

//the Concrete Aggregate

public class BookCatalog : IAggregate<Book>

{

private List<Book> \_books = new List<Book>();

public void AddBook(Book book)

{

\_books.Add(book);

}

public IIterator<Book> CreateIterator()

{

return new BookIterator(\_books);

}

}

// Book class

public class Book

{

public string Title { get; set; }

public string Author { get; set; }

public Book(string title, string author)

{

Title = title;

Author = author;

}

}

Here below the code to text the example above:

BookCatalog catalog = new BookCatalog();

catalog.AddBook(new Book("Book 1", "Author 1"));

catalog.AddBook(new Book("Book 2", "Author 2"));

catalog.AddBook(new Book("Book 3", "Author 3"));

IIterator<Book> iterator = catalog.CreateIterator();

while (iterator.HasNext())

{

Book book = iterator.Next();

Console.WriteLine($"Title: {book.Title}, Author: {book.Author}");

}

In this use case, the Iterator Pattern allows library patrons to iterate through the books in the catalog without exposing the internal structure of the BookCatalog class, providing a clean and encapsulated way to browse the library's collection.

**Memento.**

The Memento pattern is used to capture and externalize the internal state of an object so that the object can be restored to that state later.

This pattern is useful in scenarios where you need to implement features like undo functionality, or you want to save and restore the state of an object without exposing its internal details.

The Memento pattern typically involves three main roles:

Originator: This is the object whose state you want to save and restore.

It creates and returns Memento objects that represent its internal state.

Memento: This is an object that stores the state of the Originator.

It has two main responsibilities: storing the state and allowing the Originator to restore its state from the Memento.

Caretaker: This is the client code that uses the Originator and Memento to save and restore the state.

The Caretaker is responsible for keeping track of the Memento objects and applying them to the Originator when needed.

Here's an example of the Memento pattern in C#:

// Originator class

class TextEditor

{

private string text;

public string Text

{

get { return text; }

set { text = value; }

}

public TextEditorMemento Save()

{

return new TextEditorMemento(text);

}

public void Restore(TextEditorMemento memento)

{

text = memento.SavedText;

}

public override string ToString()

{

return text;

}

}

// Memento class

class TextEditorMemento

{

public string SavedText { get; private set; }

public TextEditorMemento(string text)

{

SavedText = text;

}

}

// Caretaker class

class TextEditorHistory

{

public TextEditorMemento Memento { get; set; }

}

Here is client side code:

TextEditor textEditor = new TextEditor();

TextEditorHistory history = new TextEditorHistory();

// Editing and saving state

textEditor.Text = "First draft";

history.Memento = textEditor.Save();

textEditor.Text = "Second draft";

// Restoring state

textEditor.Restore(history.Memento);

Console.WriteLine("Current text: " + textEditor);

In this example, we have an TextEditor class representing the Originator, a TextEditorMemento class representing the Memento, and a TextEditorHistory class representing the Caretaker.

The Caretaker keeps track of the Memento object, which stores the state of the TextEditor.

You can see how we save and restore the state of the TextEditor object using Memento objects.

**Null object**

The Null Object Pattern is a design pattern used in object-oriented programming to handle the absence of an object or the need for a default behavior when dealing with objects.

It is particularly useful in situations where you want to avoid explicit null checks or conditional statements that often lead to code that is difficult to maintain and error-prone.

Here's an explanation of the Null Object Pattern:

**Problem:** In many programming scenarios, you encounter situations where you need to deal with objects that may or may not exist or have a valid value.

For example, you might have a reference to an object, and you want to call methods or access properties on it, but there's a possibility that the object is null or undefined.

This can lead to code cluttered with if-else checks to handle these cases, making the code harder to read and maintain.

**Solution:** The Null Object Pattern suggests creating a special "Null" or "Default" object that behaves like a regular object but implements default behavior for the methods and properties you might call on it when the actual object is absent.

Here are the key elements of the Null Object Pattern:

**Interface or Base Class**: Define an interface or base class that represents the common interface for all objects in the hierarchy.

**Concrete Classes**: Create concrete classes that implement this interface or inherit from the base class. These are the real objects in your system.

**Null Object**: Create a special Null Object class that also implements the same interface but provides default or do-nothing implementations for the methods and properties.

This Null Object is used when an actual object is absent.

**Client Code**: In your client code, instead of checking for null or undefined objects, you simply use the Null Object.

Since the Null Object adheres to the same interface, it can be used interchangeably with the real objects without causing runtime errors.

**Benefits of Null Object Pattern:**

**Reduced Conditional Statements**: The pattern eliminates the need for explicit null checks, resulting in cleaner and more readable code.

**Default Behavior**: It provides a consistent default behavior for missing or null objects, reducing the risk of unexpected errors.

**Simplifies Client Code**: Client code becomes more straightforward and doesn't need to handle special cases for null objects.

**Encapsulation**: The pattern encapsulates the null-checking logic within the Null Object itself, making the client code cleaner and more maintainable.

Let’s see a concrete example, suppose you have a customer management system, and you want to implement a customer notification system.

Customers can have different notification preferences, such as email or SMS. However, some customers might not have specified any preference.

In this case, you want to use a Null Notification object to handle the absence of a notification preference.

Here's the C# example:

// Define the INotification interface

public interface INotification

{

void SendNotification(string message);

}

// Concrete notification implementations

public class EmailNotification : INotification

{

private string email;

public EmailNotification(string email)

{

this.email = email;

}

public void SendNotification(string message)

{

// Implement sending an email notification logic here

Console.WriteLine($"Sending email to {email}: {message}");

}

}

public class SMSNotification : INotification

{

private string phoneNumber;

public SMSNotification(string phoneNumber)

{

this.phoneNumber = phoneNumber;

}

public void SendNotification(string message)

{

// Implement sending an SMS notification logic here

Console.WriteLine($"Sending SMS to {phoneNumber}: {message}");

}

}

// Null Notification implementation

public class NullNotification : INotification

{

public void SendNotification(string message)

{

// Do nothing (no-op) when using the Null Notification

}

}

// Customer class

public class Customer

{

public string Name { get; }

private INotification notificationPreference;

public Customer(string name, INotification notificationPreference = null)

{

Name = name;

// Use the Null Notification if no preference is specified

this.notificationPreference = notificationPreference ?? new NullNotification();

}

public void Notify(string message)

{

notificationPreference.SendNotification(message);

}

}

Here is the client code:

// Create customers with and without notification preferences

var customer1 = new Customer("Carl", new EmailNotification("alice@example.com"));

var customer2 = new Customer("Bob", new SMSNotification("+123456789"));

var customer3 = new Customer("Charlie"); // No preference, uses Null Notification

// Notify customers

customer1.Notify("Your order has shipped.");

customer2.Notify("Payment received.");

customer3.Notify("Special offer!");

// Customer 3 won't receive any notifications due to Null Notification.

In this example:

We define the INotification interface for sending notifications.

Concrete notification classes (EmailNotification and SMSNotification) implement the INotification interface with specific notification methods.

The NullNotification class is a Null Object that implements the INotification interface with no-op methods.

The Customer class has a name and an optional notification preference.

If no preference is specified during customer creation, it defaults to the Null Notification.

The Notify method of the Customer class uses the notification preference to send notifications.

If the preference is the Null Notification, no actual notification is sent.

In the client code (Program class), we create customers with and without notification preferences and demonstrate how the Null Object Pattern handles the absence of notification preferences gracefully.

Customers with notification preferences receive notifications, while customer3 (Charlie) doesn't have a specified preference and uses the Null Notification, effectively ignoring notifications.

**Observer**

The Observer Pattern is a behavioral design pattern that defines a one-to-many dependency between objects so that when one object (the subject or observable) changes state, all its dependents (observers) are notified and updated automatically.

This pattern is often used to implement distributed event handling systems, in which one object (the subject) maintains a list of dependent objects (observers) that are notified of state changes.

Let's create a concrete example of the Observer Pattern in C# for a real-world scenario.

We'll implement a stock market monitoring system where multiple investors (observers) are interested in receiving updates about the changes in the stock prices (subject).

When the stock price changes, all investors should be notified and updated with the new stock price.

//Define the Observer interface

public interface IInvestor

{

void Update(string stockSymbol, decimal stockPrice);

}

//Define the Subject (Observable)

public class StockMarket

{

private Dictionary<string, decimal> stocks = new Dictionary<string, decimal>();

private List<IInvestor> investors = new List<IInvestor>();

public void AddStock(string symbol, decimal price)

{

stocks[symbol] = price;

}

public void Attach(IInvestor investor)

{

investors.Add(investor);

}

public void Detach(IInvestor investor)

{

investors.Remove(investor);

}

public void UpdateStockPrice(string symbol, decimal newPrice)

{

if (stocks.ContainsKey(symbol))

{

stocks[symbol] = newPrice;

NotifyInvestors(symbol, newPrice);

}

}

private void NotifyInvestors(string symbol, decimal newPrice)

{

foreach (var investor in investors)

{

investor.Update(symbol, newPrice);

}

}

}

//Define Concrete Observers (Investors)

public class Investor : IInvestor

{

private string name;

public Investor(string name)

{

this.name = name;

}

public void Update(string stockSymbol, decimal stockPrice)

{

Console.WriteLine($"{name} received an update for {stockSymbol} - New Price: {stockPrice:C}");

}

}

Now you can use this pattern in your client in the following way:

//Create the stock market and investors

StockMarket stockMarket = new StockMarket();

Investor investor1 = new Investor("John");

Investor investor2 = new Investor("Alice");

// Step 5: Attach investors to the stock market

stockMarket.Attach(investor1);

stockMarket.Attach(investor2);

// Step 6: Add stocks to the market

stockMarket.AddStock("AAPL", 150.0m);

stockMarket.AddStock("GOOGL", 2800.0m);

// Step 7: Update stock prices

stockMarket.UpdateStockPrice("AAPL", 155.0m);

stockMarket.UpdateStockPrice("GOOGL", 2850.0m);

In this example:

StockMarket is the observable object representing the stock market.

It maintains a list of stocks and a list of investors. It provides methods to add stocks, attach and detach investors, and update stock prices.

Investor is the concrete observer class that implements the IInvestor interface.

Each investor is interested in receiving updates about specific stocks.

When a stock price changes, the Update method is called to notify the investor with the new stock price.

In the Main method, we create a stock market, two investors, attach them to the stock market, add some stocks, and then update the stock prices.

When the stock prices are updated, both investors are notified and display the new prices.

**State**

The State Pattern is a behavioral design pattern that allows an object to alter its behavior when its internal state changes.

The pattern achieves this by defining a set of state classes that encapsulate the behavior associated with different states and allowing the context (the object whose behavior changes) to switch between these states seamlessly.

Here's a breakdown of the key components of the State Pattern:

**Context**: This is the object that maintains a reference to the current state object and delegates the behavior to that state object.

The context can change its state by transitioning to a different state object.

**State**: This is an interface or an abstract class that defines a set of methods that represent the different states the context can be in.

Each concrete state class implements these methods to provide specific behavior for that state.

**Concrete State**: These are the classes that implement the State interface or inherit from the State abstract class. Each concrete state class defines the behavior associated with a particular state of the context.

Now, let's see an example of the State Pattern in C#:

Suppose we are building a simple audio player that can be in different playback states: Playing, Paused, and Stopped. We'll use the State Pattern to manage these states.

// State interface

public interface IState

{

void Play();

void Pause();

void Stop();

}

// Concrete State classes

public class PlayingState : IState

{

public void Play()

{

Console.WriteLine("Already playing.");

}

public void Pause()

{

Console.WriteLine("Pausing the playback.");

}

public void Stop()

{

Console.WriteLine("Stopping the playback.");

}

}

public class PausedState : IState

{

public void Play()

{

Console.WriteLine("Resuming the playback.");

}

public void Pause()

{

Console.WriteLine("Already paused.");

}

public void Stop()

{

Console.WriteLine("Stopping the playback.");

}

}

public class StoppedState : IState

{

public void Play()

{

Console.WriteLine("Starting playback.");

}

public void Pause()

{

Console.WriteLine("Cannot pause, the player is stopped.");

}

public void Stop()

{

Console.WriteLine("Already stopped.");

}

}

// Context class

public class AudioPlayer

{

private IState currentState;

public AudioPlayer()

{

currentState = new StoppedState(); // Initial state is Stopped

}

public void ChangeState(IState newState)

{

currentState = newState;

}

public void Play()

{

currentState.Play();

}

public void Pause()

{

currentState.Pause();

}

public void Stop()

{

currentState.Stop();

}

}

Now you can use your state pattern in your client application like that:

AudioPlayer player = new AudioPlayer();

player.Play(); // Output: Starting playback.

player.Pause(); // Output: Cannot pause, the player is stopped.

player.Stop(); // Output: Already stopped.

player.Play(); // Output: Starting playback.

player.Pause(); // Output: Pausing the playback.

In this example, we have the AudioPlayer context class that can be in different states (Playing, Paused, Stopped).

The IState interface defines the methods that represent the possible actions the player can take in each state.

The concrete state classes (PlayingState, PausedState, and StoppedState) implement these methods to provide specific behavior for each state.

By changing the state of the AudioPlayer using the ChangeState method, we can control its behavior based on its current state.

This is the essence of the State Pattern, where the behavior of an object changes based on its internal state.

Let’s see another example: a traffic light implementation.

A traffic light can be in several states: Red, Yellow, and Green.

// State interface

public interface ITrafficLightState

{

void Handle(TrafficLight trafficLight);

}

// Concrete State classes

public class RedState : ITrafficLightState

{

public void Handle(TrafficLight trafficLight)

{

Console.WriteLine("Traffic light is red. Stop!");

Thread.Sleep(2000); // Simulate red light duration

trafficLight.ChangeState(new GreenState());

}

}

public class YellowState : ITrafficLightState

{

public void Handle(TrafficLight trafficLight)

{

Console.WriteLine("Traffic light is yellow. Prepare to stop.");

Thread.Sleep(1000); // Simulate yellow light duration

trafficLight.ChangeState(new RedState());

}

}

public class GreenState : ITrafficLightState

{

public void Handle(TrafficLight trafficLight)

{

Console.WriteLine("Traffic light is green. Go!");

Thread.Sleep(2000); // Simulate green light duration

trafficLight.ChangeState(new YellowState());

}

}

// Context class

public class TrafficLight

{

private ITrafficLightState currentState;

public TrafficLight()

{

currentState = new RedState(); // Initial state is Red

}

public void ChangeState(ITrafficLightState newState)

{

currentState = newState;

}

public void Request()

{

currentState.Handle(this);

}

}

And here the code to simulate the change of traffic light’s state:

TrafficLight trafficLight = new TrafficLight();

for (int i = 0; i < 5; i++)

{

trafficLight.Request();

}

In this example, we have a TrafficLight context class that can be in three states: RedState, YellowState, and GreenState.

Each state implements the ITrafficLightState interface and defines the behavior associated with that state.

The TrafficLight class uses the State Pattern to change its state based on the current state of the traffic light.

In the Main method, we create a TrafficLight object and simulate the behavior of a traffic light by repeatedly calling the Request method.

The traffic light transitions between red, yellow, and green states based on the specified durations, simulating a real-world traffic light.

**Strategy**

The Strategy Pattern is a behavioral design pattern that defines a family of algorithms, encapsulates each one, and makes them interchangeable.

It allows you to select an algorithm at runtime based on the context or the specific requirements of a task.

This pattern promotes the principle of "composition over inheritance" by favoring object composition over static inheritance.

Here's how the Strategy Pattern works:

Define a family of algorithms (strategies) that perform a specific task.

Create an interface or an abstract class that declares a method to execute the strategy.

Implement concrete strategy classes that provide different implementations of the strategy interface.

In the context class, maintain a reference to the strategy interface and use it to execute the selected strategy.

Here's a simple example of the Strategy Pattern in C#:

Suppose you are building a payment processing system where you can apply different payment methods (e.g., credit card, PayPal, Bitcoin) to make payments.

You want to make the payment method interchangeable at runtime.

Define the strategy interface:

public interface IPaymentStrategy

{

void ProcessPayment(double amount);

}

Implement concrete strategy classes for different payment methods:

public class CreditCardPayment : IPaymentStrategy

{

public void ProcessPayment(double amount)

{

Console.WriteLine($"Paid {amount:C} using Credit Card.");

// Add logic to process payment via credit card here

}

}

public class PayPalPayment : IPaymentStrategy

{

public void ProcessPayment(double amount)

{

Console.WriteLine($"Paid {amount:C} using PayPal.");

// Add logic to process payment via PayPal here

}

}

public class BitcoinPayment : IPaymentStrategy

{

public void ProcessPayment(double amount)

{

Console.WriteLine($"Paid {amount:C} using Bitcoin.");

// Add logic to process payment via Bitcoin here

}

}

Create a context class that allows you to change the payment strategy:

public class ShoppingCart

{

private IPaymentStrategy paymentStrategy;

public ShoppingCart(IPaymentStrategy strategy)

{

this.paymentStrategy = strategy;

}

public void Checkout(double totalAmount)

{

paymentStrategy.ProcessPayment(totalAmount);

}

public void SetPaymentStrategy(IPaymentStrategy strategy)

{

this.paymentStrategy = strategy;

}

}

Now, you can use the Strategy Pattern in your client:

var cart = new ShoppingCart(new CreditCardPayment());

// Make a payment

cart.Checkout(100.00);

// Change the payment method to PayPal

cart.SetPaymentStrategy(new PayPalPayment());

// Make another payment with the new payment method (PayPal)

cart.Checkout(50.00);

// Change the payment method to Bitcoin

cart.SetPaymentStrategy(new BitcoinPayment());

// Make another payment with the new payment method (Bitcoin)

cart.Checkout(200.00);

**Template method.**

The Template Method Pattern is used in object-oriented programming to define the skeleton or outline of an algorithm in a method, while allowing some of the steps within the algorithm to be implemented by subclasses.

This pattern promotes code reusability and helps in defining a common structure for a set of related algorithms.

Here's how the Template Method Pattern works:

A superclass (or abstract class) defines a template method that outlines the steps of an algorithm but leaves certain steps to be implemented by concrete subclasses.

This template method is often declared as final to prevent subclasses from altering the algorithm's structure.

The template method consists of a series of method calls, which are either abstract or concrete methods.

The abstract methods are placeholders for the steps that subclasses need to implement, while the concrete methods provide default or shared behavior that can be reused by all subclasses.

Subclasses inherit from the superclass and provide concrete implementations for the abstract methods, customizing the algorithm to their specific needs.

They can also choose to override any of the concrete methods to tailor the behavior as required.

When a client code interacts with the template method, it uses the method provided by the superclass.

This ensures that the algorithm's structure remains consistent across different subclasses while allowing for customization of specific steps.

Benefits of the Template Method Pattern are:

**Code Reusability:** The common steps of an algorithm are implemented in the template method, promoting code reuse among subclasses.

**Consistency:** The pattern enforces a consistent structure for related algorithms, making the code easier to understand and maintain.

**Customization:** Subclasses can customize specific steps of the algorithm while keeping the overall structure intact.

**Reduced Duplication:** The pattern reduces code duplication by encapsulating shared behavior in the superclass.

**Extension without Modification:** You can add new subclasses with different implementations without modifying the existing code in the superclass.

Example use cases for the Template Method Pattern include implementing sorting algorithms (where the comparison and swapping steps are customized by subclasses), document processing (where different document types may have unique processing steps), and game frameworks (where game loops or player actions may vary but share common structures).

Here's an example of the Template Method Pattern in C#. In this example, we'll create a template for building different types of sandwiches:

// Abstract class defining the template method

abstract class Sandwich

{

public void MakeSandwich()

{

PrepareBread();

AddIngredients();

AddCondiments();

WrapSandwich();

}

protected abstract void PrepareBread();

protected abstract void AddIngredients();

protected abstract void AddCondiments();

protected virtual void WrapSandwich()

{

Console.WriteLine("Wrap the sandwich in paper.");

}

}

// Concrete subclass 1 for creating a BLT sandwich

class BLTSandwich : Sandwich

{

protected override void PrepareBread()

{

Console.WriteLine("Prepare white bread for BLT.");

}

protected override void AddIngredients()

{

Console.WriteLine("Add bacon, lettuce, and tomato.");

}

protected override void AddCondiments()

{

Console.WriteLine("Add mayonnaise and mustard.");

}

}

// Concrete subclass 2 for creating a Veggie sandwich

class VeggieSandwich : Sandwich

{

protected override void PrepareBread()

{

Console.WriteLine("Prepare whole wheat bread for Veggie sandwich.");

}

protected override void AddIngredients()

{

Console.WriteLine("Add lettuce, tomato, cucumber, and bell peppers.");

}

protected override void AddCondiments()

{

Console.WriteLine("Add hummus and balsamic dressing.");

}

protected override void WrapSandwich()

{

Console.WriteLine("Wrap the Veggie sandwich in foil.");

}

}

You can use Template method pattern in your code like this:

Console.WriteLine("Making a BLT Sandwich:");

Sandwich bltSandwich = new BLTSandwich();

bltSandwich.MakeSandwich();

Console.WriteLine("\nMaking a Veggie Sandwich:");

Sandwich veggieSandwich = new VeggieSandwich();

veggieSandwich.MakeSandwich();

In this C# example:

Sandwich is an abstract class that defines the template method MakeSandwich().

It also declares three abstract methods PrepareBread(), AddIngredients(), and AddCondiments(), which must be implemented by concrete subclasses.

It also provides a default implementation for the WrapSandwich() method, which can be overridden by subclasses.

BLTSandwich and VeggieSandwich are concrete subclasses that inherit from Sandwich.

They provide specific implementations for the abstract methods to create different types of sandwiches.

The VeggieSandwich class overrides the WrapSandwich() method to customize the wrapping behavior.

In the Main method, we create instances of BLTSandwich and VeggieSandwich and then make sandwiches using the MakeSandwich() method.

The template method ensures that the sandwich-making process follows a consistent structure, with bread preparation, ingredient addition, and condiment application, while allowing for customization in each sandwich type.

When you run this C# code, you will see output demonstrating the use of the Template Method Pattern to create different types of sandwiches with a shared preparation process but varying ingredients and wrapping.

**Visitor Pattern.**

The Visitor pattern is used in object-oriented programming to separate the algorithm from the structure of an object.

It allows you to add new operations or behaviors to a set of objects without modifying their class definitions.

The primary purpose of the Visitor pattern is to achieve the following goals:

**Separation of Concerns:** The Visitor pattern separates the behavior (or algorithm) from the elements (or objects) that it operates on.

This separation makes it easier to add new behaviors to a set of objects without modifying their classes, which can be especially useful when working with classes that you don't have control over or that are part of a complex inheritance hierarchy.

**Open-Closed Principle:** The pattern adheres to the open-closed principle, which states that software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. With the Visitor pattern, you can introduce new behaviors (visitor implementations) without altering the existing codebase.

**Double Dispatch:** The Visitor pattern allows for double dispatch, which is a mechanism that enables the selection of a specific method or operation based on both the type of the object being visited and the type of the visitor.

This enables more flexible and dynamic behavior selection.

Common scenarios where the Visitor pattern is useful include:

When you have a complex object structure with various classes, and you want to perform different operations on these objects without modifying their class hierarchy.

When you need to add new behaviors or operations to a set of classes without altering their source code, especially in situations where you can't modify the existing classes.

When you want to achieve dynamic dispatch based on both the type of the object being visited and the type of the visitor.

In essence, the Visitor pattern provides a way to achieve polymorphism in situations where traditional inheritance and method overriding may not be the most suitable solution.

It allows you to extend the functionality of classes in a more modular and maintainable manner.

However, it does introduce some complexity and can make the code harder to understand if not used judiciously, so it should be applied with care.

// Define the abstract Element class that will be visited by the Visitor.

abstract class Element

{

public abstract void Accept(IVisitor visitor);

}

// Concrete TextElement class representing text content.

class TextElement : Element

{

public string Text { get; set; }

public override void Accept(IVisitor visitor)

{

visitor.VisitTextElement(this);

}

}

// Concrete HyperlinkElement class representing hyperlinks.

class HyperlinkElement : Element

{

public string Url { get; set; }

public string Text { get; set; }

public override void Accept(IVisitor visitor)

{

visitor.VisitHyperlinkElement(this);

}

}

// Define the Visitor interface.

interface IVisitor

{

void VisitTextElement(TextElement textElement);

void VisitHyperlinkElement(HyperlinkElement hyperlinkElement);

}

// Concrete implementation of HTML generation visitor.

class HtmlVisitor : IVisitor

{

public void VisitTextElement(TextElement textElement)

{

Console.WriteLine($"<p>{textElement.Text}</p>");

}

public void VisitHyperlinkElement(HyperlinkElement hyperlinkElement)

{

Console.WriteLine($"<a href=\"{hyperlinkElement.Url}\">{hyperlinkElement.Text}</a>");

}

}

// Concrete implementation of plain text generation visitor.

class PlainTextVisitor : IVisitor

{

public void VisitTextElement(TextElement textElement)

{

Console.WriteLine(textElement.Text);

}

public void VisitHyperlinkElement(HyperlinkElement hyperlinkElement)

{

Console.WriteLine($"{hyperlinkElement.Text} ({hyperlinkElement.Url})");

}

}

You can use Visitor pattern in your code like this:

// Create a list of elements in a document.

List<Element> elements = new List<Element>

{

new TextElement { Text = "This is a paragraph of text." },

new HyperlinkElement { Url = "https://www.example.com", Text = "Visit Example.com" }

};

// Create visitors (HTML and plain text) and apply them to the elements.

IVisitor htmlVisitor = new HtmlVisitor();

IVisitor plainTextVisitor = new PlainTextVisitor();

Console.WriteLine("HTML Output:");

foreach (var element in elements)

{

element.Accept(htmlVisitor);

}

Console.WriteLine("\nPlain Text Output:");

foreach (var element in elements)

{

element.Accept(plainTextVisitor);

}

In this example:

We define two concrete element classes: TextElement representing text content and HyperlinkElement representing hyperlinks.

Both classes inherit from the abstract Element class and implement the Accept method.

We define an IVisitor interface with methods for visiting each type of element (e.g., VisitTextElement and VisitHyperlinkElement).

We implement two concrete visitor classes: HtmlVisitor for generating HTML output and PlainTextVisitor for generating plain text output.

Both visitors implement the IVisitor interface and define how to process each type of element.

In the Main method, we create a list of elements in a document and two visitors.

We then iterate through the elements, applying each visitor, which generates the desired output format based on the type of element.

This example demonstrates how the Visitor pattern allows you to separate the behavior (output generation) from the structure of the document elements, enabling you to add new output formats (visitors) without modifying the element classes.