## Principles of object oriented design

* Program to an interface, not an implementation
* Favor object composition over class inheritance
  + Inheritance and Parameterization can’t change at runtime, composition can

# Patterns Summary

This part tries to explain the commonly used design patterns

## Creational Patterns

### Abstract Factory

**Intent:** Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

**Own words**: Abstract factory interface is used by concrete factories to create certain concrete types of an abstract type

**A screenshot of a vehicle registration form

Description automatically generatedOwn example:**

In this example we have a car and a motorcycle, both implement the IVehicle interface. To create these vehicles we have a separate CarFactory and MotorcycleFactory, but since they both implement IVehicleFactory, We can use the IVehicleFactory and don’t need to think about what kind of vehicle we are dealing with. This is useful for example in a class with dependency injection. A vehicle factory is received in the constructor and we can create vehicles regardless of what vehicle. Another benefit is scalability. If tomorrow I want to add a boat which is also an IVehicle, as long as my boat factory implements the IVehicleFactory, The rest of my code will keep working.

**C#:** Interfaces are a built in feature of c# so nothing notable

### Builder

**Intent:** Separate the construction of a complex object from its representation so that the same construction process can create different representations.

**Own words:** A class can be so large and complex, that using a constructor to create it is very messy. For example the class has a lot of values that might or might not be implemented but creating sub classes for all of these values would be overkill. A builder allows you to build objects step by step and adding what is needed only.

**Director object:** This is an optional part of the builder pattern, The director class defines the order in which to execute the building steps, while the builder provides the implementation for those steps.

**Own example:**

A diagram of a car

Description automatically generatedIn this example, a dirtbike is made. A hondaDirtBikeBuilder is passed on to our director which creates the frame, engine and wheels in the correct order. We can then get the dirbike from our builder.

IDirtbikeBuilder builder = new HondaDirtbikeBuilder;

Director director = new Director(HondaDirtbikeBuilder);

director.ConstructBike(hondaBuilder);

Dirtbike hondaDirtbike = hondaBuilder.GetDirtbike();

**C#:**  this pattern is used by default to build the host in the program.cs file of a .Net webapi

### Factory Method

**Intent:** Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

**Own words:** The Factory Method design pattern provides an interface for creating objects in a superclass, allowing subclasses to alter the type of objects that will be created. It allows for the creation of objects without specifying the exact class of object that will be created, providing flexibility in object creation. This pattern is especially useful when a class can't anticipate the class of objects it must create.

### Prototype

**Intent:** Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

**Own words:** Creational design patterns enable you to duplicate existing objects without tying your code to specific classes. However, directly copying an object without knowledge of its exact type or private fields can be challenging. To ensure objects can be copied, they are designed to inherit an interface with a clone method, allowing them to return a duplicate of themselves.

### Singleton

**Intent:** Ensure a class only has one instance, and provide a global point of access to it.

**Own words:** Having only one instance of a class can be advantageous, such as ensuring a database is accessed through a single point. This is commonly achieved by having a static instance of the class within the class itself, and then accessing this instance through a static method.

## Structural patterns

### Adapter

**Intent:** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

**Own words:** The Adapter Pattern allows incompatible interfaces to work together by creating a bridge between them. It involves a wrapper class, known as the "adapter," that converts the interface of one class into another interface that a client expects. This pattern enables classes with incompatible interfaces to work together without modifying their source code, promoting code reusability and flexibility in integrating different systems.

### Bridge

**Intent:** Decouple an abstraction from its implementation so that the two can vary independently.

**Own words:** Sometimes certain fields or logic within a class can be considered a separate entity, in which case it might be better to separate it into its own class and have the original class contain a reference to it. This way, the two classes can evolve separately while remaining connected. The Bridge Pattern achieves this by creating an abstraction (often an abstract class or interface) that contains a reference to the implementor (another class or interface), allowing them to vary independently.

### Composite

**Intent:** Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

**Own words:** To create a tree of objects, ensure they all share a common interface for their basic functionality. By allowing objects to "own" other objects, you can form a tree structure where compositions of objects and individual objects can be handled uniformly. This approach enables the manipulation of complex structures as if they were individual objects, simplifying interactions within the tree.

### Decorator

Also known as Wrapper

**Intent:** Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

**Own Words:** The Decorator or Wrapper Pattern works by dynamically adding new behaviors or responsibilities to objects without altering their existing structure. It involves creating a set of decorator classes that are used to wrap concrete components. These decorators implement the same interface as the components they wrap, allowing them to seamlessly substitute for the original objects. Each decorator can add new functionality before or after calling the original component's methods, creating a chain of decorators that can modify the behavior of the base component. This pattern promotes flexibility and extensibility, as new behaviors can be added by creating new decorators rather than modifying the original classes.

### Façade

**Intent:** Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use

**Own words:** The Facade Pattern provides a simplified interface to a complex system, acting as a single entry point for client interaction. It involves creating a facade class that delegates client requests to various components of the system, hiding their complexities. This pattern encapsulates the system's complexities behind a unified interface, making it easier for clients to use and reducing dependencies between the client code and subsystems. It promotes loose coupling and improves code readability by providing a high-level interface that shields clients from the system's implementation details.

### Flyweight

**Intent:** Flyweight is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.

**Own words:** When dealing with large numbers of objects, you need to minimise RAM usage. This is possible by separating extrinsic (unique) and intrinsic (shared) properties into different objects. This way the intrinsic objects can reuse the extrinsic ones and there is no need to save duplicates.

### Proxy

**Intent:** This pattern lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

**Own words:** The Proxy Pattern provides a surrogate or placeholder for another object to control access to it, this placeholder implements the same interface as the original object so the code using it can stay the same. It is used when you want to add a level of indirection to control access to an object, such as for lazy initialization, access control, logging, or caching.

## Behavioural patterns

### Chain of responsibility

**Intent:** Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

**Own words:** Classes can often be owned by a parent class, and own a child class themselves, creating a chain of classes. With the Chain of Responsibility pattern, you create a linked list of handlers where each handler has a reference to the next handler in the chain. When a request is made, it is passed through the chain until a handler is found that can process it. This allows for dynamic and flexible handling of requests as any handler can either handle the request or pass it to the next handler in the chain, promoting loose coupling and easy addition of new handlers.

### Command

Also known as Action or Transaction

**Intent:** Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

**Own words:** The Command Pattern is usually implemented by creating a Command interface or abstract class that defines a method like execute(). Concrete command classes then implement this interface, each encapsulating a specific action. A client object holds a reference to a command, and when it needs the action to be executed, it calls the execute() method on the command. This decouples the sender of a request from the receiver, allowing for parameterization of clients with different commands, undoable operations, and queuing of requests.

### Interpreter

Will not discuss this pattern because it is too specific to certain applications.

### Iterator

**Intent:** Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

**Own words:** In more complicated structures like a tree, the traversal behavior of the Iterator Pattern is separated into an iterator object by defining an iterator interface. This interface typically includes methods like hasNext() and next(). Each node in the tree implements a method to create an iterator specific to its traversal logic (e.g., in-order, pre-order). The concrete iterator classes then implement these methods to navigate the tree according to the specified traversal logic. This abstraction allows the client to traverse complex tree structures without knowing the details of traversal, promoting flexibility and code reusability.

### Mediator

**Intent:** Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

**Own words:** When using the Mediator Pattern, requests or method calls between classes happen through a mediator class, which will encapsulate the communication logic. This pattern eliminates mutual dependencies between classes that use each other’s functionality. Each class only needs to know about the mediator, reducing direct dependencies and promoting loose coupling. The mediator coordinates the interactions between classes, acting as a central hub where objects can communicate without being directly aware of each other.

### Momento

Also known as snapshot

**Intent:** Memento is a behavioral design pattern that lets you save and restore the previous state of an object without revealing the details of its implementation.

**Own words:** This pattern is implemented by creating a snapshot object which saves the state of your originator object, with methods to get and set the state. Usually a caretaker object is also created, which has methods to create the snapshot and to restore the originator object back to the snapshot. This caretaker often implements the command pattern.

### Observer

Also known as publish-subscribe

**Intent**: Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they’re observing.

**Own words:** The Observer Pattern works by establishing a one-to-many relationship between objects. When the state of one object (the subject) changes, all its dependents (observers) are notified and updated automatically. The pattern involves two main components: the Subject, which maintains a list of observers and notifies them of any state changes, and the Observer, which defines an interface that concrete observers implement to receive updates from the subject.

### State

**Intent:** Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

**Own words:** This pattern involves extracting the logic of a class into separate state objects. Each state object contains the original object and assumes its functionality, allowing for different state objects to modify the behavior of the original object. By delegating functionality to different state objects, the original object's behavior can change dynamically based on its current state.

### Strategy

**Intent:** Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

**Own words:** In a class that performs an algorithm or logic that can be isolated, applying the Strategy Pattern involves isolating this logic into separate strategy objects. By doing so, the class can easily switch between different algorithms or logics by storing a selected strategy as a field and executing it when needed. This approach promotes flexibility and allows the class to adapt to varying requirements without altering its core implementation.

### Template method

**Intent:** Template Method is a behavioral design pattern that defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.

**Own words:** This pattern is implemented by splitting an algorithm into multiple methods or steps that are contained in an object. This object now can be extended by different implementations of the algorithms that can overwrite certain steps. This way none of the steps need to be duplicated within the code.

### Visitor

**Intent:** Visitor is a behavioral design pattern that lets you separate algorithms from the objects on which they operate.

**Own words:**  When you have an algorithm that needs to operate on different types of objects, you can separate the algorithm from the objects by creating a visitor object. This visitor object contains the algorithm's implementations for each type of object. Next, each type of object will have a method, usually called accept, that takes the visitor object as a parameter. This accept method allows the object to receive the visitor and lets the visitor perform its operations on the object.

# Sources:

* Book: <https://www.javier8a.com/itc/bd1/articulo.pdf>
* Refactoring guru: <https://refactoring.guru/>
* <https://medium.com/@gustavorestani/the-most-used-design-patterns-in-net-development-80d76f9fb6b>
* <https://www.dofactory.com/net/>

[1] Gamma, E., Vlissides, J., Helm, R., & Johnson, R. (n.d.). Design Patterns: Elements of Reusable Object-Oriented Software.

[2] Rahman, S. (2019, July 24). The 3 Types of Design Patterns All Developers Should Know (with code examples of each). Retrieved March 16, 2024, from freeCodeCamp.org website: <https://www.freecodecamp.org/news/the-basic-design-patterns-all-developers-need-to-know/>

[3] GfG. (2023, September 27). Gang of Four (GOF) Design Patterns. Retrieved March 16, 2024, from GeeksforGeeks website: <https://www.geeksforgeeks.org/gang-of-four-gof-design-patterns/>

[4] YouTube. (2020, March 27). Design patterns in C#. YouTube. <https://www.youtube.com/watch?v=bzQeMqcgILY&list=PLBEm2Vv2nD-Ppk8U_LaR8wXl47kgCI8Dl>

[5] Design Patterns. (2014). Retrieved March 16, 2024, from Refactoring.guru website: <https://refactoring.guru/design-patterns>

[6] Alle, M. (2023). Singleton Design Pattern In C#. Retrieved March 17, 2024, from C-sharpcorner.com website: <https://www.c-sharpcorner.com/UploadFile/8911c4/singleton-design-pattern-in-C-Sharp/>

[7] BillWagner. (2023, April 28). lock statement - synchronize thread access to a shared resource - C#. Retrieved March 17, 2024, from Microsoft.com website: <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/lock>

[8] dotnet-bot. (2024). Lazy Class (System). Retrieved March 17, 2024, from Microsoft.com website: <https://learn.microsoft.com/en-us/dotnet/api/system.lazy-1?view=net-8.0>

[9] Rick-Anderson. (2023, November 7). Dependency injection in ASP.NET Core. Retrieved March 17, 2024, from Microsoft.com website: <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-8.0>

‌[10] Tutor, L. (2023, August 17). Observer Pattern in C#: From Basics to Advanced - Laks Tutor - Medium. Retrieved March 24, 2024, from Medium website: <https://medium.com/@lexitrainerph/observer-pattern-in-c-from-basics-to-advanced-ea4b2d748e>

## Ai prompts:

explain in about 3 sentences what the factory method design pattern does

make the content and meaning of this text clearer "With this creational design patterns, you copy existing objects without making your code dependant on your classes. But because copying an object without knowing its concrete type or private fields can be impossible. So to make sure you can copy objects you make them inherit an interface that enforces a clone method, in which they return a copy of themselves. "

make the content and meaning of this text clearer "Sometimes it can be beneficial to only have one instance of a class, for example to make sure a database only gets accessed trough one point. This is usually done by having a static instance of the class in the class, and returning this trough a static method. "

correct this text about the bridge pattern "Sometimes certain fields or logic within a class can be considered a separate entity, then it might be better to separate it into its own class, and just have the original class contain a reference to it. That way the 2 classes can evolve separately while still being connected. This is usually done by creating an abstract class."

make the content and meaning of this text clearer "A tree of objects can be created by first making sure that all objects use the same basic interface with their common functionality, and then by letting objects own other objects you can create a tree structure where compositions of objects and objects can be treated the same. "

explain in 4 sentences how a facade pattern works

Classes can often be owned by a parent class, and own a child class themselves, creating a chain of classes. With the chain of responsibility patterns you create... complete

how does the iterator pattern work in case of a tree data structure

This pattern extracts the logic of a class into state objects. A state object contains the original objects and takes over its functionality. Different state objects can modify the functionality of the original object this way.

improve/clarify this explanation of the strategy pattern "In a class which executes some kind of algorithm or logic that can be isolated, isolating this logic allows the class to implement different algorithms/logics easily by saving one strategy as a field and executing that strategy."

If an algorithm needs to work for multiple types of objects, separate the object from the algorithm by making a visitor object that contains the algorithm and its implementations for different types of objects. Then have the different types of objects contain a method that takes your visitor object as parameter. " rewrite this to make it clearer

what can you tell me about the observer pattern in c#

what is the goal of the strategy pattern in 4 sentences

give me examples of anti-patterns