**Creational**

1. **Singleton** - The singleton pattern is used to limit creation of a class to only one object.

<https://www.tutorialspoint.com/design_pattern/singleton_pattern.htm>

<https://refactoring.guru/design-patterns/singleton>

1. **Factory Method** - In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

<https://refactoring.guru/design-patterns/factory-method>

<https://www.tutorialspoint.com/design_pattern/factory_pattern.htm>

1. **Abstract Factory** is a creational design pattern, which solves the problem of creating entire product families without specifying their concrete classes. Also called factory of factories.

<https://refactoring.guru/design-patterns/abstract-factory>

1. **Builder** is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code. Builder lets you run some additional construction steps before fetching the product.
2. **Prototype** - Implementing correct copy strategy for complex objects.

<https://www.dofactory.com/net/prototype-design-pattern>

this.MemberwiseClone -> Shallow Copy (all value types without references)

this.MemberwiseClone + references setup -> Deep copy

Direct assignment keeps the references and up on changing them, the newly created object changes it as well. We want to avoid that by copying directly and ensuring the copy will not change.

**Structural**

1. **Adapter** is a structural design pattern that allows objects with incompatible interfaces to collaborate. [Adapter](https://refactoring.guru/design-patterns/adapter) is commonly used with an existing app to make some otherwise-incompatible classes work together nicely. We should use the Adapter class whenever we want to work with the existing class but its interface is not compatible with the rest of our code. Basically, the Adapter pattern is a middle-layer which serves as a translator between the code implemented in our project and some third party class or any other class with a different interface.

<https://www.codeproject.com/Articles/774259/Adapter-Design-Pattern-Csharp>

1. **Bridge** pattern decouples the abstraction from the implementation so they can work independently. It is used for applying great extensibility on both the class and the abstract class. They can be changed independently without breaking one another. Message, message sender.
2. **Composite** is a structural design pattern that lets you compose objects into **tree structures** and then work with these structures as if they were individual objects. Implement single interface for all objects. Implement Composite object (elements that have children) and leaf elements that share the same interface.
3. **Decorator** is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.  Decorator pattern allows a user to add new functionality to an existing object without altering its structure. This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact. Use the Decorator pattern when you need to be able to assign extra behaviors to objects at runtime without breaking the code that uses these objects.
4. **Facade** is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes. Use the Facade pattern when you need to have a limited but straightforward interface to a complex subsystem. Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system. This pattern involves a single class which provides simplified methods required by client and delegates calls to methods of existing system classes.
5. **Flyweight** pattern is primarily used to reduce the number of objects created and to decrease memory footprint and increase performance.
6. **Proxy** is a structural design pattern that 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.

**Behavioral**

1. **Command** pattern is a data driven design pattern and falls under behavioral pattern category. A request is wrapped under an object as command and passed to invoker object. Invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command.
2. **State** is a behavioral design pattern that lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.  Use the State pattern when you have an object that behaves differently depending on its current state, the number of states is enormous, and the state-specific code changes frequently.  Use the pattern when you have a class polluted with massive conditionals that alter how the class behaves according to the current values of the class’s fields.  Use State when you have a lot of duplicate code across similar states and transitions of a condition-based state machine.
3. **Chain of Responsibility** is a behavioral design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain. In this pattern, normally each receiver contains reference to another receiver. If one object cannot handle the request then it passes the same to the next receiver and so on.
4. **Iterator** patternis a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.). The main idea of the Iterator pattern is to extract the traversal behavior of a collection into a separate object called an iterator. Use the Iterator pattern when your collection has a complex data structure under the hood, but you want to hide its complexity from clients (either for convenience or security reasons).
5. **Mediator** is a behavioral design pattern that lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object. Mediator pattern is used to reduce communication complexity between multiple objects or classes. This pattern provides a mediator class which normally handles all the communications between different classes and supports easy maintenance of the code by loose coupling.
6. **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. Memento pattern is used to restore state of an object to a previous state. Use the Memento pattern when you want to produce snapshots of the object’s state to be able to restore a previous state of the object.
7. **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.
8. **Strategy** implements similar classes that only differ in the way they execute some behavior based on the context. Use the Strategy pattern when you want to use different variants of an algorithm within an object and be able to switch from one algorithm to another during runtime.
9. **Template** patternis 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. In Template pattern, an abstract class exposes defined way(s)/template(s) to execute its methods. Its subclasses can override the method implementation as per need.
10. **Visitor** is a behavioral design pattern that lets you separate algorithms from the objects on which they operate.