**Creational Patterns**

Creational patterns are ones that create objects for you, rather than having you instantiate objects directly. This gives your program more flexibility in deciding which objects need to be created for a given case.

* **Abstract factory:** provide an interface for creating families of related or dependent objects without specifying their concrete classes.
* **Builder:** separate the construction of a complex object from its representation, allowing the same construction process to create various representations.
* **Factory method:** define an interface for creating a single object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
* **Prototype:** specify the kinds of objects to create using a prototypical instance, and create new objects from the 'skeleton' of an existing object, thus boosting performance and keeping memory footprints to a minimum.
* **Singleton:** ensure a class has only one instance, and provide a global point of access to it.
* [**Object Pool**](https://sourcemaking.com/design_patterns/object_pool)  
  Avoid expensive acquisition and release of resources by recycling objects that are no longer in use

**Structural Patterns**

These concern class and object composition. They use inheritance to compose interfaces and define ways to compose objects to obtain new functionality.

* **Adapter:** allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.
* **Bridge:** decouples an abstraction from its implementation so that the two can vary independently.
* **Composite:** composes zero-or-more similar objects so that they can be manipulated as one object.
* **Decorator:** dynamically adds/overrides behaviour in an existing method of an object.
* **Facade:** provides a simplified interface to a large body of code.
* **Flyweight:** reduces the cost of creating and manipulating a large number of similar objects.
* **Proxy:** provides a placeholder for another object to control access, reduce cost, and reduce complexity.
* [**Private Class Data**](https://sourcemaking.com/design_patterns/private_class_data)  
  Restricts accessor/mutator access

**Behavioral Patterns**

Most of these design patterns are specifically concerned with communication between objects.

* **Chain of responsibility:** delegates commands to a chain of processing objects.
* **Command:** creates objects which encapsulate actions and parameters.
* **Interpreter:** implements a specialized language.
* **Iterator:** accesses the elements of an object sequentially without exposing its underlying representation.
* **Mediator:** allows loose coupling between classes by being the only class that has detailed knowledge of their methods.
* **Memento:** provides the ability to restore an object to its previous state (undo).
* **Observer:** is a publish/subscribe pattern which allows a number of observer objects to see an event.
* **State:** allows an object to alter its behavior when its internal state changes.
* **Strategy:** allows one of a family of algorithms to be selected on-the-fly at runtime.
* **Template:** method defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.
* **Visitor:** separates an algorithm from an object structure by moving the hierarchy of methods into one object
* [**Null Object**](https://sourcemaking.com/design_patterns/null_object)  
  Designed to act as a default value of an object