**Design Patterns**

**Introduction:**

Design patterns are generalized solutions to repeated occurring problems in Software Design. We should think of design patterns not as a finished solution but a template to solve a more general problem. There are three main category of design patterns, Creational, Structural and Behavioral design patterns. Each of these categories has a set of specific design patterns that we will examine.

**Creational Design Patterns:**

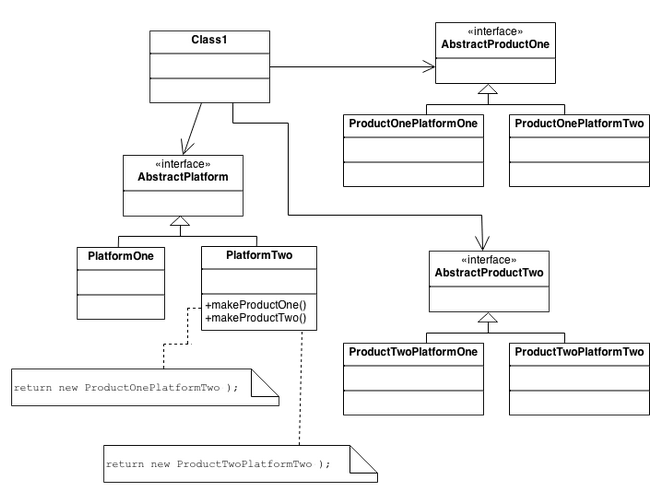
Creational patterns refer to designs that are specific to class instantiation and object creation. In this section we will take a look at the following creational design patterns:

* Abstract Factory
* Builder
* Factory Method
* Object Pool
* Prototype
* Singleton

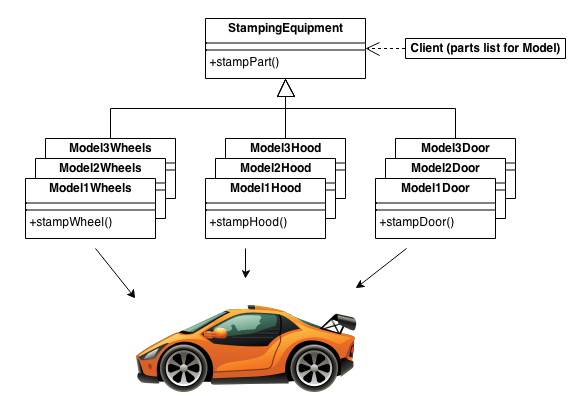
**Abstract Factory:**

The abstract factory design pattern is a pattern intended to provide an interface for creating related objects, without defining their concrete classes. It is typically good to use this design pattern when we are concerned about *portability*, e.g. across *different platform dependencies*, and encapsulating abstracted families of objects.

The abstract factory has factory methods used to create a concrete object part of the family of objects. Figure 1, shows an example of the general structure of an abstract factory design. A more specific example would be of a manufacture that is sheet metal stamping different parts of an automobile. For example, the wheels, the hood, the doors, etc. for different models of cars, which have different layouts of these general parts. We can see the design pattern visually in figure 2.



**Figure 1: An Example of a General Abstract Factory**



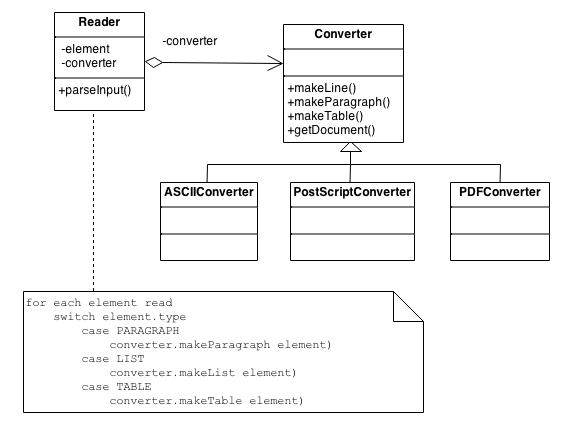
**Figure 2: A Specific Use-Case of an Abstract Factory Design Pattern**

**Builder:**

The builder design pattern is a creational pattern intended to separate the representation of an object from the construction of the object. This is so the pattern can use the same construction process for different representations of the object.

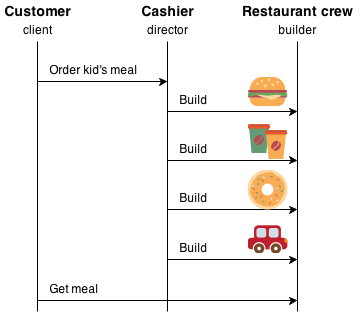
The “Director” calls the services of the “builder” to create parts of the complex object piece by piece. When the “director” is finished the client can retrieve the results of the complete object from the “builder”. This pattern allows for finer control over the construction process, which is unlike other creational patterns that construct the end product in on shot.

The structure of the builder design pattern can be seen below in figure 3, where the Reader object encapsulates the parsing of the common input. The Builder allows for many different possible polymorphic creations of many peculiar representations of targets.



**Figure 3: Structure of Builder Design Pattern**

An example that can highlight the usefulness of this pattern is, a fast-food restaurant that constructs different children’s meals. The children’s meals typically have a main item, side item, a drink, and a toy. Where we have a number of different main items, side items, drinks and toys. This example highlights the point that the creation process of the children’s meal is the same, however the contents in each children’s meal can vary significantly. Figure 4 displays a visual representation of this design pattern.

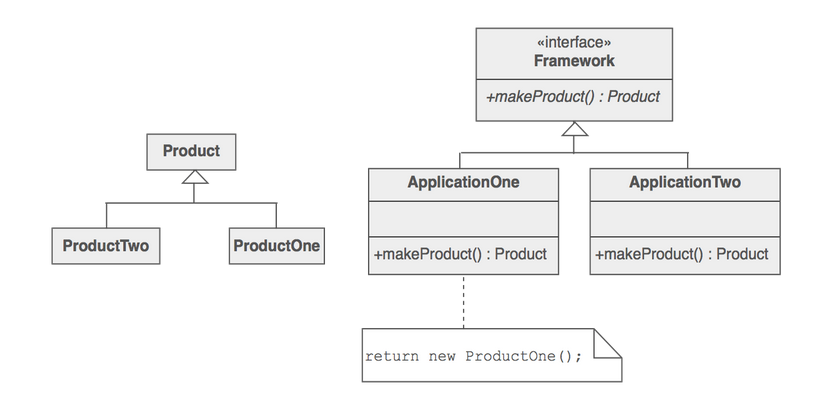


**Figure 4: Example of the Builder Design Pattern**

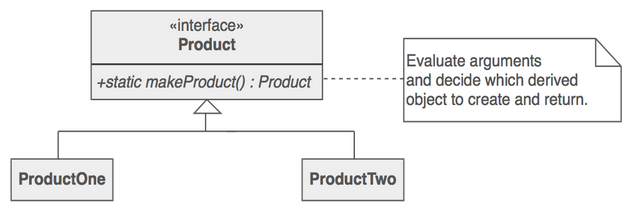
**Factory Method:**

The factory method design pattern is an interface used to create an object, however leaves the responsibility to the subclasses to decide which class to instantiate. A superclass (interface) will specify the standard and generic behavior of the different types of objects and then allow the subclasses to define the details that are supplied by the client. The factory method is very similar to the Abstract Factory except that there is not as much of an emphasis on a family of objects.

The structure of the Factory Method design pattern can be seen below in figure 5. A more popular approach of the factory method is to use a static method of the class that returns an object of that class’ type, figure 6.



**Figure 5: Structure of the Factory Method Design Pattern**



**Figure 6: Example of a Static Factory Method**

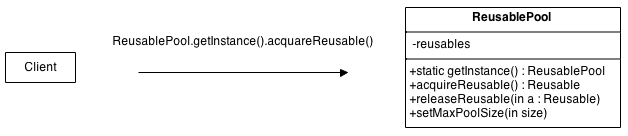
**Object Pool:**

The Object Pool design pattern is designed to help manage objects. This will help the performance in situations here the cost of initializing a class is high and the rate of instantiation of a class is high. When a client has access to an object pool (or resource/reusable pool) they can avoid creating a new object by simply accessing the pool for one of the already instantiated objects in the pool. The pool is typically build as a singleton, in order to manage all of the objects.

The object pool allows clients to “Check out” an object and once it is no longer need by the client it is returned to the pool. It is important to note that we don’t want a process to wait for an object to be returned to the pool, so the object pool will create new objects as they are needed if that object is “Checked out” or has not been created.

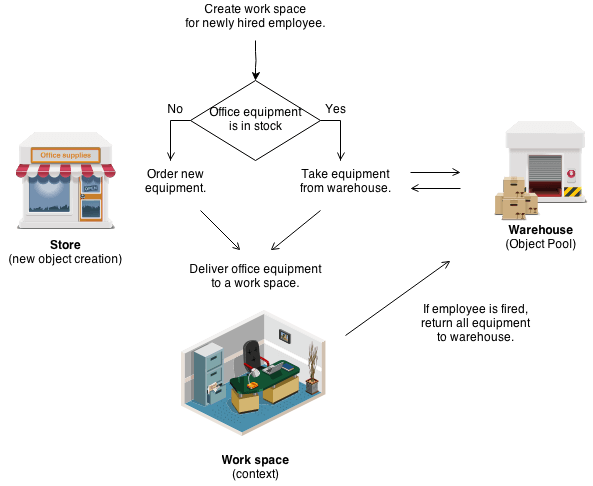
The Structure of the Object Pool (or resource/reusable pool) can be seen in figure 7. The design pattern is made up of a resource pool and a client or clients that have access to the pool. Since it is desirable for the resource pool to manage all of the unused objects, we create the resource pool as a singleton, which has the constructor as a private function. This forces other classes to call getInstance to get an instance of the resource pool. When a client wants access to an object in the pool it will make a call to the acquireReusable method. The pool will return the object and remove it from the pool, if the pool is empty then the acquireReusable method creates a Reusable object.

If the client wants to return a resource to the pool it passes it through the releaseReusable method. By returning the reusable object back to the pool it allows another client or the same client to access it again at a later time.



**Figure 7: Object Pool Structure**

An example of the Object Pool design pattern is an office warehouse, figure 8 will be used as a visual tool for this example. In this example the warehouse will be the resource pool, where office supplies are held. When a new employee is hired, the warehouse (resource pool) is checked for specific office equipment (reusable objects). If the warehouse (resource pool) does not have any the office equipment (reusable objects), the equipment is bought (object creation). If an employee is fire or moved to a different area the office equipment (reusable objects) is returned to the warehouse (resource pool).

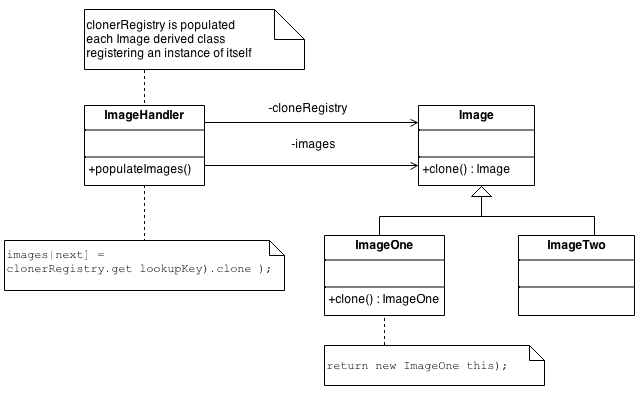


**Figure 8: An abstract Example of the Object Pool Design Pattern**

**Prototype:**

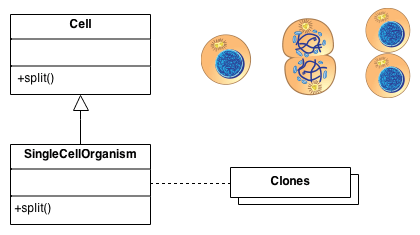
The Prototype design pattern is intended to hide the complexity of the creating new instances of different related objects from the client. We do not want the client to be calling the *new* operator. Instead of creating new instances of objects from scratch we look to clone (or copy) existing objects. This design should be used if there are potentially a number of classes that we only want to use at runtime. This design patter should also be considered if the creation of an object is expensive and the copy of that same object is more efficient than creating that object from scratch.

The Prototype design has an abstract base class with a pure virtual of a clone method and a dictionary of all the “cloneable” concrete derived classes. Any class that needs a “polymorphic instance” capability: derives itself from the abstract base class, registers its prototypical instance and implements the clone operation. The client will no longer need to call the *new* operator. Instead the client will just call the *clone* operation on the abstract base class, supplying an enum or string to the *clone* operation, which is used to return a copy of the concrete derive class. The structure of the prototype design pattern can be seen below in figure 9.



**Figure 9: Structure of Prototype Design Pattern**

A natural example of the Prototype design pattern is the mitotic division of a cell, resulting in two identical cells, where the cell copies an identical version of itself. When the cell splits it creates an identical cell with the same genotype. Figure 10 highlights this example visually.

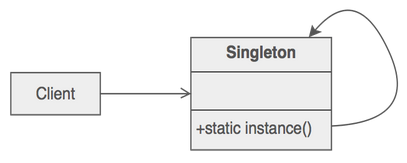


**Figure 10: A Real World Example of the Prototype Design Pattern**

**Singleton:**

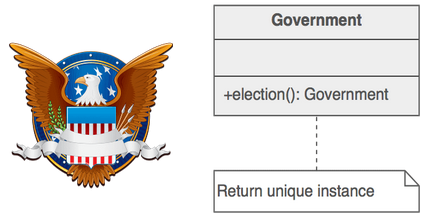
The purpose of the Singleton design pattern is to ensure that a class only has one instance. If we want to ensure only a single instance of the class, the class needs to be responsible for creation, initialization, access, and enforcement. This is done by declaring the instance of the object as a static private member of the class. The class also needs to contain a public static member function that encapsulates all the initialization of the object as well as provides access to the static member instance. The “static member function accessor” approach will not support subclassing of the singleton class. It is important to note that deleting a Singleton class is a non-trivial design problem.

The structure of a Singleton object is seen in figure 11. The instance needs to have the property of “initialization on first use”.



**Figure 11:** **Structure of Singleton Design Pattern**

A real world example of the Singleton design pattern could be the idea of the Presidency of the United States. There is only a Single President of the United States, figure 12.



**Figure 12: Example of Singleton Object**

**Structural Design Patterns:**

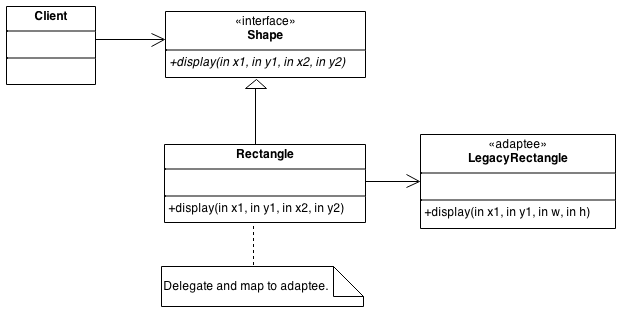
Structural patterns focus on object composition and interface design. In this section we will take a look at the following structural design patterns:

* Adapter
* Bridge
* Composite
* Decorator
* Façade
* Flyweight
* Proxy

**Adapter:**

The Adapter design pattern allows you to convert an existing interface into a different interface. This is done by wrapping an existing class with a new interface. This design pattern is helpful when we have an existing class, possibly from a previous project, that has some of the functionality we are looking for but some parts of the class need to be changed. The adapter pattern is about creating an intermediary abstraction that maps the old class with the new system.

The structure of the adapter function can be seen in figure 13. In figure 13 we have a legacyRectangle class that displayed information using the following parameters: x, y, w (width), and h (height). However, the new client wants to pass the “Upper left x and y” and the “Lower right x and y”, which is highlighted with the new Rectangle class, which contains a legacyRectangle object.

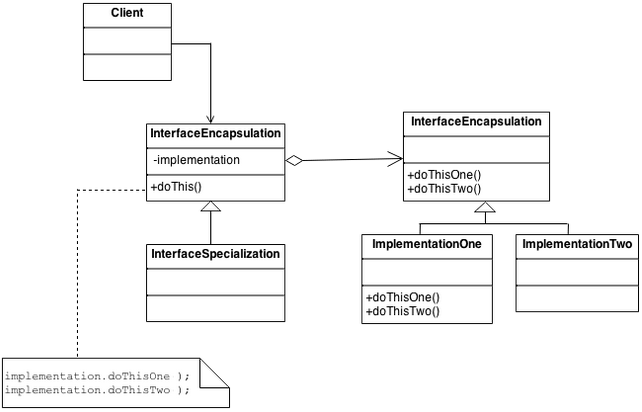


**Figure 13: Structure of Adapter Design Pattern**

**Bridge:**

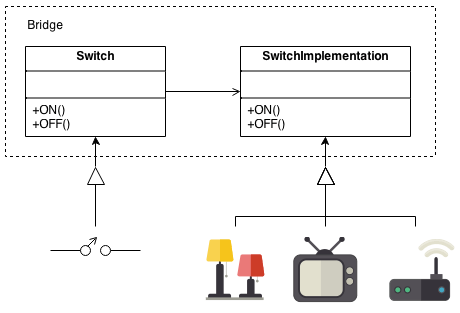
The intent of the bridge design pattern is to decouple an abstraction for the implementation, so that we can vary the two components independently. The interface class should have a pointer to an abstract implementation class. The bridge pattern is a synonym for the “Handle-Body” idiom.

The client does not care about the platform-dependent details of the class, so the bridge pattern encapsulates the platform-dependent details behind an abstraction “Wrapper”. The structure of this design pattern can be seen in figure 14.



**Figure 14: Structure of Bridge Design Pattern**

A real world example of the bridge design pattern is a household switch which controls lights, ceiling fans, etc. The purpose of the switch is to turn a device on or off, however the actual switch can be implemented as a pull chain (e.g. ceiling fan), a simple two position switch, or a dimmer switch. All are examples of different implementations of the abstract idea off a switch, which is just used to turn on and off a device, figure 15.

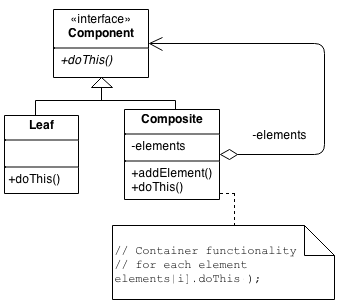


**Figure 15: Real-World Example of Bridge Design Pattern**

**Composite:**

The Composite design pattern is a tree structure of objects that are composed of smaller objects, this is called a whole-part hierarchy. The composite design pattern allows clients to treat the individual object and the composition of objects, uniformly.

The Composite design requires an abstract base class (or the component class) that specifies the behavior that is used uniformly across all the primitive (leaf) and composite objects. The primitive (leaf) and composite classes are subclasses of the component class, as seen in figure 16. The composite classes are managers of child composites or primitives. These composite classes, will normally contain AddChild() and RemoveChild() functions

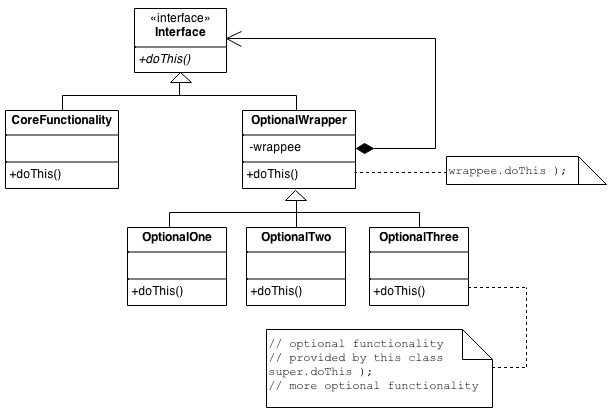


**Figure 16: Structure of Composite Design Pattern**

There are multiple examples of the whole-part hierarchies. One example is a Row-Column GUI layout managers that setup the layout of different widgets, where a widget could be another Row-Column GUI layout manager. Another example could be file directories, where we can have a single file inside of a file directory and/or we can have more file directories inside of the top-level file directory. Another example could be different containers that contain different elements, which those elements themselves can be containers.

**Decorator:**

The Decorator design pattern aims to add additional responsibilities to objects dynamically. The problem that the Decorator design pattern tries to solve is if the client wants to add behavior or state to an individual object or objects at run-time. The Decorator design pattern involves encapsulating the original object along with the potential decorator objects inside an abstract interface. The interface should be inherited by both the core object and the decorator objects and should use recursive composition to allow for an unlimited number of decorator “layers. The structure of this pattern can be seen in figure 17.



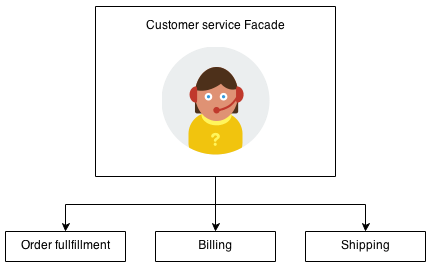
**Figure 17:** **Structure of Decorator Design Pattern**

A real-world example of the Decorator design pattern could be a Christmas tree where decorations are added to the tree. The decorator objects could be lights, garland, candy canes, glass ornaments, etc. The ornaments themselves do not change the tree itself. The additional functionality that can be added to the tree is “lighting up” the Christmas Tree, with the lights decorator object.

**Façade:**

The Façade design pattern provides an interface to a set of interfaces within a subsystem, which allows for the easier use of the subsystem. The purpose of the Façade design pattern is to provide a client with a simplified interface to utilize the functionality of a complex subsystem.

An example of the façade design pattern is a customer service representative, figure 18. The customer service representative acts as a Façade, by providing an interface to the order fulfillment department, the billing department, and the shipping department.

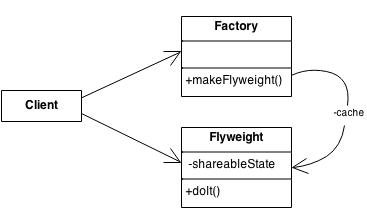


**Figure 18: Example of Façade Design Pattern**

**Flyweight:**

The Flyweight design pattern utilizes sharing to support **large** numbers of fine-grained objects efficiently. The Flyweight design pattern is conceptually broken up into two pieces: the state-dependent (extrinsic) part, and the state-independent (intrinsic) part. The intrinsic state is the shared portion of the flyweight object. The extrinsic part is stored or computed by the client and passed to the flyweight object.

The structure of the flyweight design pattern can be seen in figure 19. Typically, the flyweights are stored in a factory’s repository, which restrains the client from creating any objects. The client must request the object from the Flyweight, which will do the object creation when necessary. Any attribute that makes the object sharing impossibly must be supplied by the client.



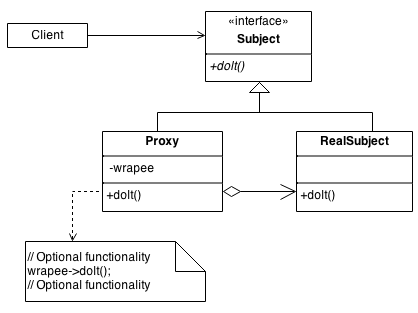
**Figure 19: Structure of Flyweight Design Pattern**

**Proxy:**

The Proxy design patter provides a class that limits the access to another class. The Proxy design pattern acts as a surrogate for another object, to protect the underlying object from undue or unexpected complexity. The proxy object will instantiate the real object the first time the client makes a request of the proxy object. Then all subsequent requests to the proxy are forwarded to the encapsulated real object. The structure of the proxy design pattern can be seen in figure 20.

Here are four common situations where the proxy pattern is applicable:

1. A *virtual proxy* is a placeholder for objects that are expensive to create. If the virtual proxy is called or needed then it will create the object, when the proxy is requested.
2. A *remote proxy* is a local representative of an object that resides in a different address space (i.e. another computer or another program running on the same computer). This could be the “stub code” you might see in remote procedure calls (RPC).
3. A *protective proxy* controls access to different objects, that the client should not have full access too. It may check the access permission of the client before forwarding requests to the actual object.
4. A *smart proxy* interposes additional actions when an object is accessed. This could include things like counting the number of references to the object so that it can be freed (e.g. a smart pointer). Another example is checking the real object is locked before it is accessed (e.g. a mutex).



**Figure 20: Structure of Proxy Design Pattern**

A real-world example of the proxy design pattern can be a check or bank draft. The check or bank draft acts as a surrogate to the funds inside the bank account. We don’t want to give someone full access to all of the funds in the account, so we provide a placeholder to a specific amount.

*Adapter* provides a different interface to its subject. *Proxy* provides the same interface. *Decorator* provides an enhanced interface.

*Decorator* and *Proxy* have different purposes but similar structures. Both describe how to provide a level of indirection to another object, and the implementations keep a reference to the object to which they forward requests.

**Behavioral Design Patterns:**

Behavior patterns focus mainly on the intercommunication between different objects. In this section we will take a look at the following behavioral design patterns:

* Chain of Responsibility
* Command
* Interpreter
* Iterator
* Mediator
* Memento
* Null Object
* Observer
* State
* Strategy
* Template Method
* Visitor

**References:**

1. <https://sourcemaking.com/design_patterns>
2. Design Patterns (Gang of four: Gamma, Helm, Johnson, Vlissides)
3. Data Structures and Problem Solving Using C++, 2nd-ed, Mark Weiss
4. <https://www.tutorialspoint.com/design_pattern/index.htm>