Overview of Design Patterns

Fundamental Design Patterns

Delegation (When not to use Inheritance)

Delegation is a way of extending and reusing a class by writing another class with additional functionality that uses instances of the original class to provide the original functionality.

Interface

Keep a class that uses data and services provided by instances of other classes independent of those classes by having it access those instances through an interface.
Related patterns are:

- Delegation
  The Delegation and Interface patterns are often used together.

Immutable

The Immutable pattern increases the robustness of objects that share references to the same object and reduces the overhead of concurrent access to an object. It accomplishes this by not allowing an object's state information to change after it is constructed. The Immutable pattern also avoids the need to synchronize multiple threads of execution that share an object.
Related patterns are:

- Single Threaded Execution
  The Single Threaded Execution pattern is the pattern most frequently used to coordinate access by multiple threads to a shared object. The Immutable object pattern can be used to avoid the need for the Single Threaded Execution pattern or any other kind of access coordination.

Marker Interface

The Marker Interface pattern uses interfaces that declare no methods or variables to indicate semantic attributes of a class. It works particularly well with utility classes that must determine something about objects without assuming they are an instance of any particular class.
Related patterns are

- Snapshot
  The Marker Interface pattern is used as part of the Snapshot pattern to allow serialization of objects.

Proxy

The Proxy pattern forces method calls to an object to occur indirectly through a proxy object that acts as a surrogate for the other object, delegating method calls to that object.

Classes for proxy objects are declared in a way that usually eliminates client object's awareness that they are dealing with a proxy. Proxy is a very general pattern that occurs in many other patterns, but never by itself in its pure form.
Related patterns are

- Access Proxy
  The Access Proxy pattern uses a proxy to enforce a security policy on access to a service providing object.
- Façade
  The Façade pattern uses a single object as a front end to a set of interrelated objects.
- Remote Proxy
  The Remote Proxy pattern uses a proxy to hide the fact that a service object is located on a different machine than the client objects that want to use it.
- Virtual Proxy
  This pattern uses a proxy to create the illusion that a service providing object exists before it has actually been created. It is useful if the object is expensive to create and its services may not be needed.
- Decorator
  The Decorator pattern is structurally similar to the Proxy pattern in that it forces access to a service providing object to be done indirectly through another object. The difference is a matter of intent. Instead of trying to manage the service, the indirection object in some way enhances the service.

Creational Patterns

Abstract Factory

Given a set of related abstract classes, the Abstract Factory pattern provides a way to create instances of those abstract classes from a matched set of concrete subclasses. The Abstract Factory pattern is useful for allowing a program to work with a variety of complex external entities such as different windowing systems with similar functionality.
Related patterns are

- Factory Method
  The Abstract Factory may uses the Factory Method pattern.
- Singleton
  Concrete Factory classes are usually implemented as Singleton classes.

Builder

The Builder pattern allows a client object to construct a complex object by specifying only its type and content. The client is shielded from the details of the object's construction.
Related patterns are

- Interface
  The Builder pattern uses the Interface pattern to hide the actual class of the object it builds.

- Composite
  The object built using the Builder pattern is typically a Composite.
- Factory Method
  The Builder pattern uses the Factory Method pattern to decide which concrete class to instantiate to build the desired type of object.
- Layered Initialization
  The Builder pattern uses the Layered Initialization pattern to create objects that build the desired type of object.
- Null Object
  The Null Object pattern may be used by the Builder pattern to provide "do nothing" implementations of methods.
- Visitor
  The Visitor pattern allows the creation requesting object to be more closely coupled to the construction of the new complex object. Instead of describing the content of the objects to be built through a series of method calls, the information is presented in bulk as a complex data structure.

Factory Method

You write a class for reuse with arbitrary data types. You organize this class so that it can instantiate other classes without being dependent on any of the classes it instantiates. The reusable class is able to remain independent of the classes it instantiates by it can delegate the choice of which class to instantiate to another object and referring to the newly created object through a common interface.
Related patterns are

- Abstract Factory
  The Factory Method pattern is useful for constructing individual objects for a specific purpose without the construction requester knowing the specific classes being instantiated. If you need to create a matched set of such objects, then the Abstract Factory pattern is a more appropriate pattern.
- Template Method
  The Factory Method pattern is often used with the Template Method pattern.

Prototype

The Prototype pattern provides an alternate way for an object to work with other objects without knowing the details of their construction.
The Prototype pattern allows an object to create customized objects without knowing their class or any details of how to create them. It works by giving prototypical objects to an object that initiates object creation. The creation initiating object then creates objects by asking the prototypical objects to make copies of themselves.
Related patterns are

- Composite
  The Prototype pattern is often used with the Composite pattern. Prototypical objects are often composite objects.
- Abstract Factory

The Abstract Factory pattern can be a good alternative to the Prototype pattern if there is no need for the dynamic changes that the Prototype pattern allows to a palette of prototypical objects.

- Façade
  The class that manages a collection of prototypical objects commonly acts as façade that separates the other classes that participate in the Prototype pattern from the rest of the program.
- Factory Method
  The Factory Method pattern may be an alternative to the Prototype pattern when the palette of prototypical objects never contains more than one object.
- Decorator
  The Prototype pattern is often used with the Decorator pattern.

Singleton

The Singleton pattern ensures that only one instance of a class is created. All objects that use an instance of that class use the same instance. You can use the Singleton pattern with many other patterns. In particular, it is often used with the Abstract Factory , Builder and Prototype patterns. The Singleton pattern has some similarity to the Cache Management pattern. A Singleton is functionally similar to a Cache that only contains one object. If multiple threads will be getting the instance of a singleton class, you can use the Double Checked Locking coding pattern to ensure that only one instance is created while avoiding the overhead of unnecessary thread synchronization after the instance is created.
Related patterns are

- Shared Object
  The Singleton pattern describes classes that have a single instance that may or may not be shared. The Shared Object pattern describes objects that are shared and may have multiple instances.
- Object Pool
  Manage the reuse of objects for a type of object that is expensive to create or only a limited number of a kind of object can be created.
  Related patterns are
- Cache Management
  The Cache Management pattern manages the reuse of specific instances of a class. The Object Pool pattern manages and creates instances of a class that can be used interchangeably.
- Factory Method
  The Factory Method pattern can be used to encapsulate the creation logic for objects. However, it does not manage them after their creation.
- Singleton
  Objects that manage object pools are usually singletons.

Partitioning Patterns

Layered Initialization

When you need multiple implementations of an abstraction, you usually define a class to encapsulate common logic and subclasses to encapsulate different specialized logic. That does not work when common logic must be used to decide which specialized subclass to create. The Layered Initialization pattern solves this problem by encapsulating the common and specialized logic to create an object in unrelated classes.
Related patterns are

- Builder
  The Builder pattern uses the Layered Initialization pattern to create a specialized object for representing data in a specific form.
- Façade
  The Layered Initialization pattern uses the Façade pattern by hiding all of the other objects participating in the pattern from clients of service objects.
- Factory Method
  When the choice of which kind of object to create does not involve any significant preprocessing of data, the Factory Method pattern may be a better choice. The Layered Initialization pattern may use the Factory Method pattern after it has decided what kind of specialized logic is needed.
- Layered Initialization
  The Layered Initialization pattern recognizes a division of responsibilities into layers during design.
- Composite
  When more than two layers of initialization are needed for initialization you can combine the Layered Initialization pattern with the Composite pattern to perform initialization in as many layers as needed.

Filter

The Filter pattern allows objects that perform different transformations and computations on streams of data and have compatible interfaces to be dynamically connected to perform combinations of operations on streams of data.
Related patterns are

- Composite
  The Composite pattern can be an alternative to the Filter pattern when the objects involved do not have a consistent interface and they can be composed statically.
- Layered Architecture
  The Layered Architecture pattern (described in volume 2) is similar to the Filter pattern. The most important difference is that the objects involved in the layered Architecture pattern correspond to different levels of abstraction.
- Decorator
  The Filter pattern is a special case of the Decorator pattern, where a data source or data sink object is wrapped to add logic to the handling of a data stream.

Composite

The Composite pattern allows you to build complex objects by recursively composing similar objects in a tree-like manner. The Composite pattern also allows the objects in the tree to be manipulated in a consistent manner, by requiring all of the objects in the tree to have a common superclass or interface.
Related patterns are

- Chain of Responsibility
  The Chain of Responsibility pattern can be combined with the Composite pattern by adding child to parent links so that children can get information from an ancestor without having to know which ancestor the information came from.
- High Cohesion
  The High Cohesion pattern (described in volume 2) discourages putting specialized methods in general purpose classes, which is something that the Composite pattern encourages.
- Visitor
  You can use the Visitor pattern to encapsulate operations in a single class that would otherwise be spread across multiple classes.

Structural Patterns

Adapter

An Adapter class implements an interface known to its clients and provides access to an instance of a class not know to its clients. An adapter object provides the functionality promised by an interface without having to assume what class is being used to implement that interface.
Related patterns are

- Anonymous Adapter
  This is a coding pattern that uses anonymous adapter objects to handle events.
- Façade
  The Adapter class provides an object that acts as an intermediary for method calls between client objects and one other object not known to the client objects. The Façade pattern provides an object that acts as an intermediary for method calls between client objects and multiple objects not know to the client objects.
- Iterator
  The Iterator pattern is a specialized form of the Adapter pattern for sequentially accessing the contents of collection objects.
- Proxy
  The Proxy pattern, like the Adapter pattern, uses an object that is a surrogate for another object. However, a Proxy object has the same interface as the object for which it is a surrogate.
- Strategy

  The Strategy pattern is structurally similar to the Adapter pattern. The difference is in the intent. The Adapter pattern allows a Client object to carry out its originally intended function in collaboration by calling method of objects the implement a

particular interface. The Strategy pattern provides objects that implement a particular interface for the purpose of altering or determining the behavior of a Client object.

Iterator

The Iterator pattern defines an interface that declares methods for sequentially accessing the objects in a collection. A class that accesses a collection only through such an interface remains independent of the class that implements the interface.
Related patterns are

- Adapter
  The Iterator pattern is a specialized form of the Adapter pattern for sequentially accessing the contents of collection objects.
- Factory Method
  Some collection classes may use the Factory Method pattern to determine what kind of iterator to instantiate.
- Null Object
  Null iterators are sometimes used to implement the Null Object pattern.

Bridge

The Bridge pattern is useful when there is a hierarchy of abstractions and a corresponding hierarchy of implementations. Rather than combining the abstractions and implementations into many distinct classes, the Bridge pattern implements the abstractions and implementations as independent classes that can be combined dynamically.
Related patterns are

- Layered Architecture
  The Bridge design pattern is a way of organizing the entities identified using the Layered Architecture pattern (described in volume 2) into classes.
- Abstract Factory
  The Abstract Factory pattern can be used by the Bridge pattern to decide which implementation class to instantiate for an abstraction object.

Façade

The Façade pattern simplifies access to a related set of objects by providing one object that all objects outside the set use to communicate with the set.
Related patterns are

- Interface
  The Interface pattern can be used with the Façade pattern to allow different sets of façade and implementation classes to be used without client classes having to be aware of the different classes.
- Law of Demeter

A conceptual model that uses the Law of Demeter pattern often gives rise to a design that follows the Façade pattern.

Flyweight

If instances of a class that contain the same information can be used interchangeably, the Flyweight pattern allows a program to avoid the expense of multiple instances that contain the same information by sharing one instance.
Related patterns are

- Composite
  The Flyweight pattern is often combined with the Composite pattern to represent the leaf nodes of a hierarchical structure with shared objects.
- Factory Method
  The Flyweight pattern uses the factory method pattern to create new flyweight objects.
- Immutable
  Shared flyweight objects are often immutable.

Dynamic Linkage

Allow a program, upon request, to load and use arbitrary classes that implement a known interface.
Related patterns are

- Virtual Proxy
  The Virtual Proxy sometimes uses the Dynamic Linkage pattern to load the class that it needs to create its underlying object.

Virtual Proxy

If an object is expensive to instantiate and may not be needed, it may be advantageous to postpone its instantiation until the object is needed. The Virtual Proxy pattern hides the fact that an object may not yet exist from its clients, by having them access the object indirectly through a proxy object that implements the same interface as the object that may not exist. The technique of delaying the instantiation of an object until it is actually needed is sometimes called lazy instantiation.
Related patterns are

- Façade
  The Façade pattern can be used with the Virtual Proxy pattern to minimize the number of proxy classes that are needed.
- Proxy
  The Virtual Proxy pattern is a specialized form of the Proxy pattern.

Decorator

The Decorator pattern extends the functionality of an object in a way that is transparent to its clients. It does that by using an instance of a subclass of the original class that delegates operations to the original object.
Related patterns are

- Delegation
  The Decorator pattern is a structured way of applying the Delegation pattern.
- Filter

The Filter pattern is a specialized version of the Decorator pattern that focuses on manipulating a data stream.

- Strategy
  The Decorator pattern is useful for arranging for things to happen before or after the methods of another object are called. If you want to arrange for different things to happen in the middle of calls to a method, consider using the Strategy pattern.

Template Method

The Template Method pattern is an alternative to the Decorator pattern that allows variable behavior in the middle of a method call instead of before or after it.
Cache Management
The Cache Management pattern allows fast access to objects that would otherwise take a long time to access. It involves keeping a copy of objects that are expensive to construct after the immediate need for the object is over. The object may be expensive to construct for any number of reasons, such as requiring a lengthy computation or being fetched from a database.
Related patterns are

- Façade
  The Cache Management pattern uses the Façade pattern.
- Publish-Subscribe
  You can use the Publish-Subscribe pattern to ensure the read consistency of a cache.
- Remote Proxy
  The Remote Proxy provides an alternative to the Cache Management pattern by working with objects that exist in a remote environment rather than fetching them into the local environment.
- Template Method
  The Cache Management pattern uses the Template Method pattern to keep its Cache class reusable across application domains.
- Virtual Proxy
  The Cache Management pattern is often used with the Virtual Proxy pattern to make the cache transparent to objects that access object in the cache.

Behavioral Patterns

Chain of Responsibility

The Chain of Responsibility pattern allows an object to send a command without knowing what object or objects will receive it. It accomplishes that by passing the command to a chain of objects that is typically part of a larger structure. Each object in the chain may handle the command, pass the command on to the next object in the chain or do both.
Related patterns are

- Composite

When the chain of objects used by the Chain of Responsibility pattern is part of a larger structure, that larger structure is usually built using the Composite pattern.

- Command
  The Chain of Responsibility pattern makes the particular object that executes a command indefinite. The Command pattern makes the object that executes a command explicit and specific.
- Template Method
  When the objects that make up a chain of responsibility are part of a larger organization built using the Composite pattern, the Template Method pattern is often used to organize the behavior of individual objects.

Command

Encapsulate commands in objects so that you can control their selection, sequencing, queue them, undo them and otherwise manipulate them.
Related patterns are

- Factory Method
  The Factory Method pattern is sometimes used to provide a layer of indirection between a user interface and command classes.
- Little Language
  You can use the Command Pattern to help implement the Little Language pattern.
- Snapshot
  You can use the Snapshot pattern to provide a coarse grained undo mechanism that saves the entire state of an object rather than a command by command account of how to reconstruct previous states.

Template Method

The Template Method pattern can be used to implement the top level undo logic of the Command pattern.
Little Language / Interpreter
Suppose you need to solve many similar problems and you notice that the solutions to these problems can be expressed as different combinations of a small number of elements or operations. The simplest way to express solutions to these problems may be to define a little language. Common types of problems you can solve with little languages are searches of common data structures, creation of complex data structures and formatting of data.
Related patterns are

- Composite
  A parse tree is organized with the Composite pattern.
- Visitor
  The Visitor pattern allows you to encapsulate logic for simple manipulations of a parse tree in a single class.

Mediator

The Mediator pattern uses an object to coordinate state changes between other objects. Putting the logic in one object to manage state changes of other objects, instead of

distributing the logic over the other objects, results in a more cohesive implementation of the logic and decreased coupling between the other objects.
Related patterns are:

- Adapter
  Mediator classes often use adapter objects to receive notifications of state changes.
- Interface
  The Mediator pattern uses the Interface pattern to keep the Colleague classes independent of the Mediator class.
- Low Coupling/High Cohesion
  The Mediator pattern is a good example of an exception to the advice of the Low Coupling/High Cohesion pattern.

Snapshot

Capture a snapshot of an object's state so that the object's state can be restored later. The object that initiates the capture or restoration of the state does not need to know anything about the state information. It only needs to know that the object whose state it is restoring or capturing implements a particular interface.
Related patterns are

- Command
  The Command pattern allows state changes to be undone on a command by command basis without having to make a snapshot of an object’s entire state after every command.

Observer

Allow objects to dynamically register dependencies between objects, so that an object will notify those objects that are dependent on it when its state changes.
Related patterns are

- Adapter
  The Adapter pattern can be used to allow objects that do not implement the required interface to participate in the Observer pattern.
- Delegation
  The Observer pattern uses the Delegation pattern.
- Mediator
  The Mediator pattern is sometimes used to coordinate state changes initiated by multiple objects to an Observable object.

State

Encapsulate the states of an object as discrete objects, each belonging to a separate subclass of an abstract state class.
Related patterns are

- Flyweight
  You can use the Flyweight pattern to share state objects.
- Mediator

The State pattern is often used with the Mediator pattern when implementing user interfaces.

Singleton

You can implement non-parametric states using the Singleton pattern.
Null Object
The Null Object pattern provides an alternative to using null to indicate the absence of an object to delegate an operation to. Using null to indicate the absence of such an object requires a test for null before each call to the other object's methods. Instead of using null, the Null Object pattern uses a reference to an object that doesn't do anything.
Related patterns are

- Singleton
  If instances of a class whose methods do nothing contain no instance specific information, then you can save time and memory by implementing it as a singleton class.
- Strategy
  The Null Object pattern is often used with the Strategy pattern.

Strategy

Encapsulate related algorithms in classes that are subclasses of a common superclass. This allows the selection of algorithm to vary by object and also allows it to vary over time.
Related patterns are

- Adapter
  The Adapter pattern is structurally similar to the Strategy pattern. The difference is in the intent. The Adapter pattern allows a client object to carry out its originally intended function by calling methods of objects that implement a particular interface. The Strategy pattern provides objects that implement a particular interface for the purpose of altering or determining the behavior of a client object.
- Flyweight
  If there are many client objects, they may share strategy objects if they are implemented as Flyweights.
- Null Object
  The Strategy pattern is often used with the Null Object pattern.

Template Method

The Template method pattern manages alternate behaviors through subclassing rather than delegation.
Template Method
Write an abstract class that contains only part of the logic needed to accomplish its purpose. Organize the class so that its concrete methods call an abstract method where the missing logic would have appeared. Provide the missing logic in subclass' methods that override the abstract methods.
Related patterns are

- Strategy

The Strategy pattern modifies the logic of individual objects. The Template Method pattern modifies the logic of an entire class.

Visitor

One way to implement an operation that involves objects in a complex structure is to provide logic in each of their classes to support the operation. The Visitor pattern provides an alternative way to implement such operations that avoids complicating the classes of the objects in the structure by putting all of the necessary logic in a separate visitor class. The Visitor pattern also allows the logic to be varied by using different visitor classes.
Related patterns are

- Iterator
  The Iterator pattern is an alternative to the Visitor pattern when the object structure to be navigated has a linear structure.
- Little Language
  In the Little Language pattern, you can use the Visitor Pattern to implement the interpreter part of the pattern.
- Composite
  The Visitor pattern is often used with object structures that are organized according to the Composite pattern.

Concurrency Patterns

Single Threaded Execution

Some methods access data or other shared resources in a way that produces incorrect results if there are concurrent calls to a method and both calls access the data or other resource at the same time. The Single Threaded Execution pattern solves this problem by preventing concurrent calls to the method from resulting in concurrent executions of the method.

Guarded Suspension

Suspend execution of a method call until a precondition is satisfied.
Related patterns are

- Balking
  The Balking pattern provides a different strategy for handling method calls to objects that are not in an appropriate state to execute the method call.
- Two-Phase Termination
  Because the implementation of the Two-Phase Termination pattern usually involves the throwing and handling of InterruptedException, its implementation usually interacts with the Guarded Suspension pattern.

Balking

If an object's method is called when the object is not in an appropriate state to execute that method, have the method return without doing anything.
Related patterns are

- Double Checked Locking
  The Double Checked Locking coding pattern (described in volume 2) is structurally simillar to the Balking pattern. Its intention is different. The balking pattern avoids executing code when an object is in the wrong state. The Double Checked Locking pattern avoids executing code to avoid unnecessary work.
- Guarded Suspension
  The Guarded Suspension pattern provides an alternate way to handle method calls to objects that are not in an appropriate state to execute the method call.
- Single Threaded Execution
  The Balking pattern is often combined with the Single Threaded Execution pattern to coordinate changes to an object's state.

Scheduler

Control the order in which threads are scheduled to execute single threaded code using an object that explicitly sequences waiting threads. The Scheduler pattern provides a mechanism for implementing a scheduling policy. It is independent of any specific scheduling policy.
Related patterns are

- Read/Write Lock
  Implementations of the Read/Write Lock pattern usually use the Scheduler pattern to ensure fairness in scheduling.

Read/Write Lock

Allow concurrent read access to an object but require exclusive access for write operations.
Related patterns are

- Single Threaded Execution
  The Single Threaded Execution pattern is a good and simpler alternative to the Read/Write Lock pattern when most of the accesses to data are write accesses.

Scheduler

The Read/Write Lock pattern is a specialized form of the Scheduler pattern.
Producer-Consumer

Coordinate the asynchronous production and consumption of information or objects.
Related patterns are

- Guarded Suspension
  The Producer-Consumer pattern uses the Guarded Suspension pattern to manage the situation of a Consumer object wanting to get an object from an empty queue.
- Pipe
  The Pipe pattern is a special case of the Producer-Consumer pattern that involves only one Producer object an only one Consumer object. The Pipe pattern usually

refers to the Producer object as a data source and the Consumer object as a data sink.

- Scheduler
  The Producer-Consumer pattern can be viewed as a special form of the Scheduler pattern that has scheduling policy with two notable features.
  The scheduling policy is based on the availability of a resource.
  The scheduler assigns the resource to a thread but does not need to regain control of the resource when the thread is done so it can reassign the resource to another thread.

Two-Phase Termination

Provide for the orderly shutdown of a thread or process through the setting of a latch. The thread or process checks the value of the latch at strategic points in its execution.