**SOLID Design Principles**

* Single Responsability Principle
  + A class should only have one reason to change.
  + Separation of concerns – different classes handling different, independent tasks/problems.
* Open-Closed Principle
  + Classes should be open for extension but closed for modification.
* Liskov Substitution Principle
* You should be able to substitute a base type for a subtype
* Interface Segregation Principle
  + Don’t put too much into an interface; split into separate interfaces
  + YAGNI – You Ain’t Going to Need It
* Dependency Inversion Principle
  + High-level modules should not depend upon low-level ones; use abstractions

**Gamma Categorization**

* Design Patterns are typically split into three categories.
* This is called Gamma Categorization after Erich Gamma, one of GoF authors.
* Creational Patters
  + Deal with the creation (construction) of objects.
  + Explicit (constructor) vs implicit (DI, reflection, etc.)
  + Wholesale (single statement) vs piecewise (step-by-step)
* Structural Patterns
  + Concerned with the structure (e.g., class members)
  + Many patterns are wrappers that mimic the underlying class’ interface
  + Stress the importance of good API design
* Behavioral Patterns
  + They are all different; no central theme

**Builder**

* When piecewise object construction is complicated, provide an API for doing it succinctly.
* A builder is a separate component for building an object.
* Can either give builder an initializer or return it via a static function.
* To make builder fluent, return self.
* Different facets of an object can be built with different builders working in tandem via a base class.

**Factory**

* A factory method is a static method that creates objects.
* A factory is any entity that can take care of objects creation.
* A factory can be external or reside inside the objects as an inner class.
* Hierarchies of factories can be used to create related objects.

**Prototype**

* A partially or fully initialized object that you copy (clone) and make use of.
* To implement a prototype, partially construct an object and store it somewhere.
* Deep copy the prototype.
* Customize the resulting instance.
* A factory provides a convenient API for using prototypes.

**Singleton**

* A component which is instantiated only once.
* A constructor can choose what to return; we can keep returning same instance.
* Monostate: many instances shared data.
* Directly depending on the Singleton is a bad idea: introduce a dependency instead.

**Adapter**

* A constructor which adapts an existing interface X to conform to the required interface Y.
* Implementing an Adapter is easy.
* Determinate the API you have and the API you need.
* Create a component which aggregates (has a reference to, …) the adapted.
* Intermediate representations can pile up: caching and other optimizations.

**Bridge**

* A mechanism that decouples an interface (hierarchy) from an implementation (hierarchy).
* Decouple abstraction from implementation.
* Both can exist as hierarchies.
* A stronger form of encapsulation.

**Composite**

* A mechanism for treating individual (scalar) objects and compositions of objects in a uniform manner.
* Objects can use ither objects via inheritance/composition.
* Some composed and singular objects need similar/identical behaviors.
* Composite design pattern lets us treat both types of objects uniformly.
* JavaScript supports iteration with Symbol.iterator
* A single object can make itself iterable by yielding this

**Decorator**

* Facilities the addition of behaviors to individual objects without inheriting from them.
* A decorator keeps the reference to the decorated object(s).
* Adds utility fields and methods to augment the object’s features.
* May or may not forward calls to the underlying object.

**Façade**

* Provides a simple, easy to understand/user interface over a large and sophisticated body of code.
* Build a Façade to provide a simplified API over a set of classes.
* May wish to (optionally) expose internals through the façade.
* May allow users to ‘escalate’ to use more complex APIs if they need to.

**Flyweight**

* A space optimization technique that lets us use less memory by storing externally the data associated with similar objects.
* Store common data externally.
* Specify an index or a reference into the external data store.
* Define the idea of ‘ranges’ on homogeneous collections and store data related to those ranges.

**Proxy**

* A class that functions as an interface to a particular resource. That resource may be remote, expensive to construct, or may require logging or some other added functionality.
* A proxy has the same interface as the underlying object.
* To create a proxy, simply replicate the existing interface of an object.
* Add relevant functionality to the redefined member functions.
* Different proxies (communication, logging, caching, etc.) have completely different behaviors.

**Chain of Responsibility**

* A chain of components who all get a chance to process a command or a query, optionally having default processing implementation and an ability to terminate the processing chain.
* Command Query Separation
  + Command = asking an action or change (e.g., please set your attack value to 2).
  + Query = asking for information (e.g., please give me your attack value).
  + CQS = having separate means of sending commands and queries to e.g., direct field access.
* Chain of Responsibility can be implemented as a chain of references or a centralized construct.
* Enlist objects in the chain, possibly controlling their order/priority.
* In a linked-list implementation, one member can impede further processing.
* Support removal of objects from the chain (lifetime control).

**Command**

* An object which represents an instruction to perform a particular action. Contains all the information necessary for the action to be taken.
* Encapsulate all details of an operation in a separate object.
* Define instruction for applying the command (either in the command itself, or elsewhere).
* Optionally define instructions for undoing the command.
* Can create composite commands (a.k.a. macros).

**Interpreter**

* A component that processes structured text data. Does so by turning it into separate lexical tokens (lexing) and then interpreting sequences of said tokens (parsing).
* Barring simple cases, an interpreter acts in two stages.
* Parsed data can then be traversed.

**Iterator**

* An object that facilities the traversal of a data structure.
* An iterator specified how you can traverse an object.
* Stateful iterators cannot be recursive.
* yield allows for much more succinct iteration.

**Mediator**

* A component that facilities communication between other components without them necessarily being aware of each other or having direct (reference) access to each other.
* Create the mediator and have each object in the system refer to it.
* Mediator engages in bidirectional, communication with its connected components.
* Mediator has functions the components can call.
* Components have functions the mediator can call.

**Memento**

* A token/handle representing the system state. Let’s us roll back to the state when the token was generated. May or may not directly expose state information.
* Mementos are used to roll back states arbitrarily.
* A memento is simply a token/handle class with (typically) no functions of its own.
* A memento is not required to expose directly the state(s) to which it reverts the system.
* Can be used to implement undo/redo.

**Observer**

* An observer is an object that wishes to be informed about events happening in the system. The entity generating the events is an observable.
* Observer is an intrusive approach: an observable must provide an event to subscribe to.
* Subscription and unsubscription handled with addition/removal of items in a list.
* Property notifications are easy; dependent property notifications are tricky.

**State**

* A pattern in which the object’s behavior is determined by its state. An object transitions from one state to another (something needs to trigger a transition).
* A formalized construct which manages state and transitions is called state machine.
* Given sufficient complexity, it pays to formally define possible states and events/triggers.
* Can define:
  + State entry/exit behaviors.
  + Action when a particular event causes a transition.
  + Guard conditions enabling/disabling a transition,
  + Default action when no transitions are found for an event.

**Strategy**

* Enables the exact behavior of a system to be selected at run-time.
* Define an algorithm at a high level.
* Define the interface you expect each strategy to follow.
* Provide for dynamic composition of strategies in the resulting object.

**Template Method**

* Allows us to define the ‘skeleton’ of the algorithm, with concrete implementations define in subclasses.
* Define an algorithm at a high level.
* Define constituent parts as empty methods/properties.
* Inherit the algorithm class, providing necessary overrides.

**Visitor**

* A component (visitor) that knows how to traverse a data structure composed of (possibly related) types.
* Propagate an accept (Visitor v) method throughout the entire hierarchy.
* Create a visitor with visitFoo(Foo), visitBar(Bar), … for each element in the hierarchy.
* Each accept() simply calls visitor.visitXxx(this)