Design Pattern Notes

* Memento
  + Use for tracking the history of an object
  + You have 3 classes
    - Originator – The primary class that you are tracking the state history of
    - Memento – a state of the object at that particular time
    - Caretaker – Also known as the history, will keep a list of memento’s to return should the developer require this
* State
  + When you want to track the state of a particular object without needing to constantly update that object
  + You can implement an interface that the primary class will use and call the same “use this object” function on it
  + That way you can just keep creating new tools/objects but the primary class does not need to change
  + Diagram

    Description automatically generated
  + Example is painting software and tracking what tool the user is currently using
    - You just need to have each tool inherit a “tool” interface that the primary class can call functions on
      * Click to start
      * Release to finish etc
* Iterator
  + When you want to iterate through something like a browser history
  + You don’t want to for, loop through the list in the main function because if you happen to change the list/array you would need to change the main functionality as well
  + To resolve this we can create an iterator the browser history can use that will have iterator functionality
  + This way we can iterate, and still change the list properties without needing to change our main functionality
  + Diagram

    Description automatically generated
* Strategy
  + We have an image storage that has a compressor and filter attached to it
  + We want to make sure that if we change or add a compressor or filter, it doesn’t change what the image storage has to do for that
  + the polymorphism principle to variable types
  + Creating a new class that implements an interface so we don’t have to change anything
  + Diagram

    Description automatically generatedDiagram

    Description automatically generated
* Template Method Pattern
  + Diagram

    Description automatically generated
  + If we want certain task logic to happen on all sub tasks we can use an abstract implementation to always handle the primary task logic, and then let the sub tasks (classes that inherit task) continue the rest of their necessary logic after
* Command Strategy
  + Diagram

    Description automatically generated
  + Diagram

    Description automatically generated
  + Assign delegates to an interface so that the invoker doesn’t need to know the specific implementation of the receiver
  + This also allows us to track commands as a history
  + Diagram

    Description automatically generated
* Observer
  + You add an observer interface that will communicate with a data source
  + Diagram

    Description automatically generated
  + Diagram

    Description automatically generated
  + Diagram

    Description automatically generated
* Mediator
  + Dialogue box or form that needs to collaborate
  + When you have a series of objects that need to communicate, instead of them just all talking to each other, they talk to a primary source
  + Diagram

    Description automatically generated
  + Diagram

    Description automatically generated
* Chain of Responsibility
  + Essentially a linked list of handlers to determine the flow of actions
  + Diagram

    Description automatically generated
* Visitor Pattern
  + Adding a new function to an interface means we have to change ALL the other functions on the classes inheriting that interface
  + Uses method overloading to change the behavior of the operations
  + Use if your object structure is stable, but want to add new operations later
  + HTML is a good example because you have a static amount of nodes, but may need to add more operations over time
  + Diagram

    Description automatically generated

Structural

* Composite pattern
  + When you want to treat any object elements the same way
  + Use an interface with the same function to run their own functionality
  + Diagram

    Description automatically generated
* Adapter
  + When we want to use an interface to adapt incoming libraries to our own code
  + Diagram

    Description automatically generated
* Decorator
  + Add additional behavior to an object, you can compound the amount of stream interfaces onto the object
  + Essentially adding a linked list of streams to continue a stream of function calls
  + Diagram

    Description automatically generated
* Façade Pattern
  + When you primary logic relies too much on multiple classes you introduce a façade that will handle the multiple objects
  + Diagram

    Description automatically generated
* Flyweight Pattern
  + Uses object pooling/factory to share an object
* Bridge Pattern
  + Diagram

    Description automatically generated
* Proxy pattern
  + Talk to a target through a proxy
  + Can perform tasks, intermediate tasks etc
  + One thing you can do is use a proxy object to lazy load something only when requested
  + Diagram

    Description automatically generated
  + Diagram

    Description automatically generated
  + Diagram

    Description automatically generated
* Prototype pattern
  + Diagram

    Description automatically generated
* Singleton
  + Ensuring a class only ever has one instance is easier to locate (get instance)
  + Diagram

    Description automatically generated
* Factory Pattern
  + Diagram

    Description automatically generated
  + Defer the creation of an object to a subclass
  + Should be implemented by inheritance
  + NOT using statics
* Abstract pattern
  + Provides and interface for creating families of related objects
  + Diagram

    Description automatically generated
* Builder Pattern
  + Separate the construction of an object from its representation
  + Diagram

    Description automatically generated