CS 340 midterm study

- Software architecture layers benefits:
  1. Simple and easy to learn and implement
  2. Fewer dependencies so mocking and testing is easier
  3. You can extend or alter the application easily because of the clear pattern
- Observer Pattern
  - Intent: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
  - Structure of solution: Make abstract subject and observer classes. Bind abstract observer class to abstract subject. Make concrete subject and observer so when subject changes, notify() on subject will update all observers
  - When to use and benefits: When a change to one object requires a change to another object and the receiving objects have no requirement to be notified. Reduces dependencies. Reduces coupling between sender and receiver and observers can be added at any time.
  - Code:
- Template Method Pattern
  - Intent: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
  - Structure of solution: Base class implements invariable parts of algorithm and subclasses override abstract methods to complete the algorithm's steps in particular ways
  - When to use and benefits: When you have an algorithm which needs to vary in the implementation of some of its steps. Reduces code duplication and lets subclasses define behavior.
  - Code:
- Facade Pattern
  - Intent: Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
  - Structure of solution: Subsystems are put into an easy to understand interface so client classes can use them easily without knowing the details (compiler that handles parsing and other low level stuff).
  - When to use and benefits: When you want to hide lower level functionality but still make it usable
  - Code:
- Proxy Pattern
  - Intent: Provide a surrogate or placeholder for another object to control access to it
  - Structure of solution: Create a placeholder for an object that shouldn't be accessed directly.
  - When to use and benefits: Whenever you need something more complex than a simple pointer to an object. Remote proxy to have a local reference when actual object is elsewhere. Virtual if actual object is expensive and needs to be accessed on demand.

Protection controls access to object like when permissions are involved. Smart reference when you need to do additional operations on object when called like increment a counter or something.

- Code:

MVC vs MVP

Model-view-controller goes from view to controller to model. The Model-view-presenter pattern goes from view to presenter to model, back to presenter and back to view. This allows for more testable code. Because the view is dumber in the MVP and the presenter does all the brainwork without needing all the view's dependencies, it can test better.

MVVP vs MVP

Model-view-viewmodel is just like MVP except it uses data-binding in the view instead of actual function calls. This makes the view designable by UI designers who aren't super familiar with code.

Mockito and spies

- Why use fake objects?: It allows us to test code without also testing dependencies on that code which we already know work. Sometimes those dependencies have operations that will take forever as well. We can mock it so our code thinks its using the needed data types without actually creating them.
- Difference between mock and spy: A mock fakes an interface, just specifying what method is called and what the return value should be. A spy has access to the body of the methods, so the methods within the body are called as well. It's like having the class with the additional luxury of Mockito when and verify calls.
- Difference between Mockito when and verify: "When" specifies a return value for the mocked object when a method on it is called. "Verify" verifies that certain methods from the mock were even called. You can verify particular methods as well as parameters passed to those methods.

Design Principles

- Code duplication: no code duplication makes code more maintainable, so when a method that hold all that potentially duplicated code is called, it is changed in all places instead of having to change it in every place. It also complies faster with less written code, and takes the code writer less time to write it.
- Orthogonality: operations change just one thing without affecting others. Fewer dependencies help us achieve this for example. A change in one class does not require changes in 10 other classes.
- Decomposition: dividing a big problem into several smaller problems
  - Single responsibility principle
  - Size/length metrics: huge classes and methods should be decomposed further
  - Complexity: methods/classes should not be super complex even if they're small enough
- Information hiding: hide things as much as possible
  - Visibility: use private, protected, and public to hide as much as possible

    - Naming: don't use names that reveal low level operations like binary search when search itself will do
    - Separate interface from implementation: java interfaces, .h and .cpp files
    - Data hiding: private on a classes elements and accessible through public method interface
- Depend on abstractions, not concretions: using a List as the type when making an arraylist instead of making it type arraylist specifically.
    - When to use another class instead of primitive (string, int, etc):
        - Domain checking: phone numbers could be strings but most strings don't constitute a phone number. Make a phone number class to check for that instead
        - Additional Operations: URL can be a string but we want to use operations specific to URLs on those strings. Make a URL class
        - Code Readability: a string type wont tell you much about the variable but a named type could say a lot like type URL.
- High quality abstractions include good naming, information hiding, and avoiding primitive obsession.
- Isolated change principle: single responsibility principle taken further. Class should take care of one thing. Same with a method. That way if something is changed in that class or method, it doesn't affect other things.
- Error handling: Never ignore an error: When an error happens the code should do one of three things
    - Recover and continue execution
    - Pass the error to the caller
    - Log the error
- Algorithm and data structure selection: use algorithms and data structures that don't take too much time or space. Optimize it for whatever it is you're doing.
- How to achieve code reuse:
    - Parameterization: instead of defining variables in a method, pass those to the method as parameters. Generic types specifically
    - Implementation inheritance: class reuses code from a base class
    - Composition/delegation: composition is when a class is made of another class. Delegation is the using of that class to do operations of the class using it.
- Open-closed principle: software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification. That is, such an entity can allow its behavior to be extended without modifying its source code.

Final Study

- Dependency Inversion Principle
    - Intent: Make high level modules independent of low level modules so they can be reused.
    - Structure of solution: High level modules implement an interface, which allows low-level modules to perform operations on them. Button can turn on lamp that has implemented buttonServer. Make everything depend on abstractions, unless you must depend on a class. This class should be unchanging in the best case. Think: Spell checker constructor has parameters for fetcher, dictionary, and extractor, but the parameters ask for an interface instead of a concrete object, this way, any dictionary that implements the dictionary interface can be used instead of a single, specific dictionary class.
    - When to use and benefits: always use so changing low-level components does not affect the high-level needs. The high-level modules will depend on an interface that a variety of low-level components can be used to implement those methods.
    - Code:
- DynamoDB
    - How is it different from a relational database?
        - Allows easy scaling from tiny projects to huge applications
        - DynamoDB doesn't have a well-defined schema. You only need a primary key to uniquely define each item. Otherwise, each item can have whatever it wants in it. Other databases require items to match a rigid schema.
        - DynamoDB does not use SQL. It has it's own operations for data-access operations.
        - Relational databases are optimized for storage, not speed, so developers must optimize their queries, indexes, and table structures to achieve that speed. DynamoDB takes care of these details so you can focus on your application instead of database stuff.
        - Relational databases have upper limits on storage, but the way DynamoDB is set up, there are no limits. Users can specify what they need and DynamoDB will allocate space without increasing latency.
    - Why does a query include a value for the partition key?
        - The partition key allows data to be stored in the same place in physical memory so queries are quick. A table with items only identified by a partition key (no sort key), all items must have a different partition key. If there is a sort key, items can have the same partition key, but the sort key must be different.
- Data Access Object (DAO) Pattern
    - How did we use DAO pattern in project? What did your interfaces look like?
        - We made classes that interacted with the database. Each class only interacted with one table in the database. Each class also had an interface so if we wanted to write data to a different database or in a different way, we could pass that into our service classes without changing the service class.

- Data transfer object (DTO) Pattern
    - Intent: Reduce method calls by making a class that holds all the data and just sending that
    - Structure of solution: I believe an example of this are the requests and responses. They take necessary data from the domain objects in the tweeter application and only send the necessary information for the operation
    - When to use and benefits: Reduces method calls, especially remote calls, for increased speed and also organization
    - Code:
- Abstracting external dependencies
    - If we are using a database for example, don't make the project rely on that database. Make abstractions so different database implementations can be switched to easily, or the current code that connects to the current database doesn't have to be scrapped.
- Designing for Testability
    - Abstract Factory
        - Taking dependency inversion further. You can make an abstract factory that implements an array of interfaced components. Then passing it that one factory, you'll have switched over a whole theme of implementations to a class. The holiday decorations exercise had 3 different decoration types, all interfaces that could be implemented into something more specific, like Halloween or easter table decorations. In this case, it would make sense that all 3 decorations would have the same holiday theme, so we make an abstract factory interface that fetches the 3 decoration interfaces. The easter factory would fetch 3 easter decoration types. Then when making the main object that prints these things to the terminal, simply passing a different abstract factory to the constructor changed all those things. This allows for easy testing, reuse of classes, and ability to make more and easily use them without changing other classes.
    - Dependency Injection
        - Inject dependencies so classes don't have to create them itself. Think passing in the objects to a constructor, and the parameter is an interface. This easily allows those dependencies to be mocked when testing. If the class creates the objects, you couldn't mock them. They would've been made upon creation of the object.
- Asynchronous Messaging, SQS **HERE?**
    - How did you use this in M4 design? Why?
    - SQS – allowed posting of a status to be sent to 10,000 users' feeds. Sends things in batches.
    - Asynchronous messaging – things were done asynchronously so one operation waiting to be finished didn't pause all other operations.
- Inheritance vs Delegation/composition. How is composition/delegation used in all these:
    - State pattern
        - Object with different states is composed of a state object that implements the state interface. It delegates functions based on the state its in.
    - Proxy pattern

        - Main class is composed of a proxy class. The proxy class is composed of the real class and delegates responsibilities to that class when the proxy's requirements are met. The main class has essentially delegated the responsibility of knowing when the operation is allowed to be performed to the proxy class, while the actual operation is delegated to the real class by the proxy.
    - Strategy pattern
        - A class is composed of an object that extends an algorithm's interface. It delegates the execution of the algorithm to that object.
    - Decorator pattern
        - The main class is composed of string sources and the decorators, which are also string source implementations. It delegates the function of making the strings to the string source classes and manipulating the strings to the decorators.
    - Why is it used? – single responsibility principle. A single class or method is responsible for one thing. Without it, the classes here would be doing many things instead of delegating specific functions to classes made for those functions.
    - Advantages/disadvantages of inheritance (extends) vs composition (has instance of class)
- Strategy Pattern
    - Strategy and template solve same problem. How solutions Differ
        - Template pattern has an abstract class that defines an algorithm. Some methods are specified and others need to be overridden by the class that extends it. The strategy pattern is different because the classes that extend it are many; not just one. A separate class is composed of an object that extends/implements the algorithm's interface, and it uses delegation to run the algorithm. For example, I could have a main class that uses an algorithm to write a string, but depending on the time of day, it wants to use a different implementation of that algorithm. The main class could check the time of day and pass in the appropriate algorithm implementation class to the class that contains it so when it runs, it uses the appropriate algorithm implementation. This means the template method pattern implementation is chosen at compile time while the strategy pattern allows the algorithm implementation to be chosen at run time, based on variables or states. Template = rigid. Strategy = flexible.
    - Which to use? Strategy or template?
        - Use strategy pattern when the implementation of a particular algorithm needs to be dynamic based on the state of the program or you want to be more flexible with the algorithm implementation you want to use. Use the template method pattern if you want fewer classes and the algorithm implementation doesn't need to be flexible. Bubble sort algorithm.
    - Intent:
    - Structure of solution:
    - When to use and benefits:
    - Code:
- Adapter Pattern

    - Implement an existing interface with another class (example online)
    - Intent: Makes a class that allows you to use one class with another, which couldn't be done previously.
    - Structure of solution: The exercise had a contact manager class and a table class, but you couldn't put the contacts into the table. Made a table data interface that specified operations the table needed to display data. Then made an adapter class that would implement the interface and could take a contact Manager in the constructor and make it usable in the table. The table would take in a TableData object in the constructor, and that adapter class was it.
    - When to use and benefits:
    - Code: Make adapter class that implements stack class. Adapter has arrayList object of type T. The overridden method "push" just uses .add() on the item passed to it, and the pop method returns the last element in the arrayList and deletes it.
- Decorator Pattern
    - Extend followers endpoint to include optional compression kinds. How to use decorator pattern
    - Intent: Decorate something
    - Structure of solution: Pass original into a decorator and it'll do some operation on it to meet a desired change. The compression example would have different decorator classes that would compress the data.
    - When to use and benefits: When you want to alter data conditionally and in particular ways. Instead of having to do it specifically each time, you can use a decorator.
    - Code:
- State Pattern **HERE**
    - How to use this to implement an elevator? Bubble gum machine?
        - Have 4 states based on if there are gumballs in the machine or a quarter inserted. You can insert a quarter, remove a quarter, add gumballs, or turn the handle to get a gumball.
            - Gumball in machine/Quarter inserterd
                - Turn handle – get a gumball and decrement amount in machine; increase profit from machine; if there are still gumballs in the machine, change state to no gumballs and no quarter; else, change state to gumball in machine and no quarter.
                - Insert quarter – already quarter, so no change
                - Remove quarter – change state to gumball in machine/no quarter
                - Add gumball – increment gumballs
            - Gumball in machine/no quarter
                - Turn handle – no quarter so no change
                - Insert quarter – change state to gumball in machine/quarter inserted
                - Remove quarter – there is no quarter so no change
                - Add gumball – increment gumballs
            - No gumball in machine/quarter inserted

        - Turn handle – same as gumball in machine/quarter inserted except no gumball comes out and it still goes to no gumball in machine/no quarter
        - Insert quarter – quarter already there so no change
        - Remove quarter – change state to no gumball in machine/no quarter
        - Add gumball – change state so gumball is in machine, increment gumballs
    - No gumball in machine/no quarter
        - Turn handle – nothing happens
        - Insert quarter – change state to no gumball in machine/quarter inserted
        - Remove quarter – nothing happens
        - Add gumball – change state so gumball is in machine, increment gumballs

  - Intent: Have a class with functions that do different things depending on the state its in
  - Structure of solution: Have different states defined through a state interface, and override the methods for each state. Basically each state will be able to do all the things as the other, but they'll react differently. The actual object that has different states will change state appropriately
  - When to use and benefits:
  - Code:
- Command Pattern **HERE**
  - Intent:
  - Structure of solution:
  - When to use and benefits:
  - Code: