Design Pattern Bullet Points

Overall

- Knowing the OO basics does not make you a OO designer
- Good OO designs are **reusable, extensible, maintainable**
- Patterns show you how to build systems with good OO design qualities
- Patterns are proven object-oriented experience
- Patterns do not give you code, they give you general solutions to design problems. You apply them to your specific application
- Patterns are not invented, they are discovered
- Most patterns and principles address issues of change in software
- Most patterns allow some part of system to vary independently of all other parts
- We often try to take what varies in a system and encapsulate it
- Patterns provide a shared language that can max the value of your communication with other developers

Observer pattern

- The observer pattern defines a one-to-many relationship between objects
- Subjects or as we also know them, observables, update observers using a common interface
- Observers are **loosely** coupled in that the observable knows nothing about them, other than that they implement the observer interface
- You can push or pull data from observable when using the pattern
- Do not depend on a specific order of notification for your observers
- Java has several implementations of the observer pattern, including the general purpose java.util.Observable
- Watch out for issues with the java.util.Observable implementation
- Do not be afraid to create your own Observable implementation if needed
- Swing makes heavy use of the observer pattern, as do many GUI frameworks
- You will also find the pattern in many other places, including JavaBeans and RMI

Factory pattern

- All factories encapsulate object creation
- Simple factory, while not a bona fide design pattern, is a simple way to decouple your clients from concrete classes
- Factory method replies on inheritance: object creation is delegated to subclasses which implement the factory interface
- Abstract factory relies on object composition:

    - Object creation is implemented in methods exposed in the factory interface
- All factory patterns promote loose coupling by reducing the dependency of your application on concrete classes
- The intent of factory method is to allow a class to defer instantiation to its subclasses
- The intent of abstract factory is to create families of related objects without having to depend on their concrete classes
- The dependency inversion principle guides us to avoid dependencies on concrete types and to strive for abstractions
- Factories are a powerful technique for coding to **abstractions**, **not concrete classes**

Singleton pattern

- At most one instance of a class in your application
- provides a **global access** point to that instance
- A **private** constructor, a **static** method combined with a **static** variable
- Examine your performance and resource constraints and carefully choose an appropriate singleton implementation for multithreaded applications
- Beware of the double-checked locking implementation; it is not thread-safe in versions before java 2, version 5
- Be careful if you are using multiple class loaders; this could defeat the singleton implementation and result in multiple instances
- If you are using a JVM earlier than 1.2, you will need to create a registry of singleton to defeat the garbage collector

Command pattern

- decouples an object, making a request from the one that knows how to perform it.
- A command object is at the center of this decoupling and encapsulates a receiver with an action (or set of actions).
- An invoker makes a request of a Command object by calling its execute() method, which invokes those actions on the receiver.
- Invokers can be parameterized with Commands, even dynamically at runtime.
- Commands may support undo by implementing an undo method that restores the object to its previous state before the execute() method was last called.

- Macro Commands are a simple extension of Commands to be invoked. Likewise, Macro Commands can easily support undo().
- In practice, it is not uncommon for “smart” Command objects to implement the request themselves rather than delegating to a receivers.
- Commands may also be used to implement logging and transactional systems.

Adapter & Facade pattern

- When you need to use an **existing** class and its interface is not the one you need, use an adapter.
- An adapter changes an interface into one a client expects.
- Implementing an adapter may require little work or a great deal of work depending on the size and complexity of the target interface.
- There are two forms of the Adapter Pattern: object and class adapters. Class adapters require multiple **inheritance.**

- When you need to **simplify** and **unify** a large interface or complex set of interfaces, use a facade.
- A facade decouples a client from a complex subsystem.
- Implementing a facade requires that we compose the facade with its subsystem and use delegation to perform the work of the facade.
- You can implement more than one facade for a subsystem.

- An adapter wraps an object to change its interface, a decorator wraps an object to add new behaviors and responsibilities, and a facade “wraps” a set of objects to simplify.

Template method pattern

- Defines the steps of an algorithm, deferring to subclasses for the implementation of those steps
- Gives us an important technique for code reuse
- The template method’s abstract class may define **concrete methods, abstract methods and hooks**
- Abstract methods are implemented by subclasses
- Hooks are methods that do nothing or default behavior in the abstract class, but may be overridden in the subclasses
- To prevent subclasses from changing the algorithm in the template method, declare template method as **final**

- The **hollywood principle** guides is to put decision-making in high-level modules that can decide how and when to call low level modules
- ~~You will see lots of uses of the template method pattern in real world code, but do not expect it all to be designed by the book~~
- The strategy and template method pattern both encapsulate algorithms, one by inheritance and one by composition
- The factory method is a specialization of template method

Iterator pattern

- Allows access to an aggregate's elements **without exposing its internal structure**
- Takes the job of iterating over an aggregate and encapsulates it in another object
- When using an iterator we relieve the aggregate of the responsibility of supporting operations for traversing its data
- An iterator provides a common interface for traversing the items of an aggregate, allowing you to use polymorphism when writing code that makes use of the items of the aggregate
- We should strive to assign only one responsibility to each class

Composite pattern

- Provides a structure to hold both individual objects and composites
- Allows clients to treat composites and individual objects uniformly
- A component is any object in a composite structure. Components may be other composites or leaf nodes
- There are many design trade-off in implementing composite. You need to balance transparency and safety with your needs

State & Strategy pattern

- Allows an object to have many different behaviors that are based on its internal state.
- Unlike a procedural state machine, the State Pattern represents state as a full-blown class.
- The context gets its behavior by delegating to the current state object it is composed with.
- By encapsulating each state into a class, we localize any changes that will need to be made.

- The State and Strategy Patterns have the same class diagram, but they differ in intent.
- Strategy Pattern typically configures Context classes with a behavior or algorithm.
- State Pattern allows a Context to change its behavior as the state of the Context changes.
- State transitions can be controlled by the State classes or by the Context classes.
- Using the State Pattern will typically result in a greater number of classes in your design.
- State classes may be shared among Context instances.

Proxy pattern

- Provides a representative for another object in order to control the client's access to it. There are a number of ways it can manage that access.
- A **Remote** Proxy manages interaction between a client and a remote object.
- A **Virtual** Proxy controls access to an object that is expensive to instantiate.
- A **Protection** Proxy controls access to the methods of an object based on the caller.
- Many other variants of the Proxy Pattern exist including caching proxies, firewall proxies, copy-on-write proxies, and so on.
- Proxy is structurally similar to Decorator, but the two differ in their purpose.
- The Decorator Pattern adds behavior to an object, while a Proxy **controls access**.
- Java's built-in support for Proxy can build a dynamic proxy class on demand and dispatch all calls on it to a handler of your choosing.
- Like any wrapper, proxies will increase the number of classes and objects in your designs.

MVC

- The model view controller pattern is a **compound** pattern consisting of the **observer, strategy and composite** patterns
- The model makes use of the observer pattern so that it can keep observers updated yet stay decoupled from them
- The controller is the strategy for the view. The view can use different implementations of the controller to get different behavior

- The view uses the composite pattern to implement the user interface, which usually consists of nested components like panels, frames and buttons
- These patterns work together to decouple the three players in the MVC model, which keeps designs clear and flexible
- The adapter pattern can be used to adapt a new model to an existing view and controller
- Model 2 is an adaption of MVC for web applications
- In model 2, the controller is implemented as a servlet and JSP & HTML implement the view

Main topics Gary mentioned in class:

- Review old quizzes
- Review design patterns and design principles
    - Know the definitions and basics of functionality
- Review end of the chapter bullet point
- Review testing
- Review cohesion and coupling
- Review agile and waterfall methodologies
- Review financial lecture focusing on high-level ideas(cover in quiz review)