Head First Design Pattern Notes

[Note : Need to refer Head First Design Patterns Book]

[Git Repo : https://github.com/AbhishekMhamane/design-patterns-examples]

Chapter 1 : Welcome to Design Patterns: intro to Design Patterns

Design Principles

1. **Identify the aspects of your application that vary and separate them from what stays the same.**

   Take the parts that vary and encapsulate them, so that later you can alter or extend the parts that vary without affecting those that don't.

2. **Program to an interface, not an implementation.**
   E.g.
   Programming to an implementation would be,
   ```
   Dog d = new Dog();
   d.bark()
   ```

   Programming to an interface,
   ```
   Animal animal = new Dog();
   animal.makeSound();
   ```

3. **Favor composition over inheritance.**
   HAS-A can be better than IS-A

Chapter 2 : Keeping your Objects in the Know: the Observer Pattern

Observer Pattern

The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.It works like publisher and subscribers.

The subject and observers define the one-to-many relationship. We have one subject, who notifies many observers when something in the subject changes. The observers are dependent on the subject—when the subject's state changes, the observers are notified.

The Power of Loose Coupling

When two objects are loosely coupled, they can interact, but they typically have very little knowledge of each other. As we're going to see, loosely coupled designs often give us a lot of flexibility (more on that in a bit). And, as it turns out, the Observer Pattern is a great example of loose coupling. Let's walk through all the ways the pattern achieves loose coupling:

First, the only thing the subject knows about an observer is that it implements a certain interface (the Observer interface). It doesn't need to know the concrete class of the observer, what it does, or anything else about it.

We can add new observers at any time. Because the only thing the subject depends on is a list of objects that implement the Observer interface, we can add new observers whenever we want. In fact, we can replace any observer at runtime with another observer and the subject will keep purring along. Likewise, we can remove observers at any time.

We never need to modify the subject to add new types of observers. Let's say we have a new concrete class come along that needs to be an observer. We don't need to make any changes to the subject to accommodate the new class type; all we have to do is implement the Observer interface in the new class and register as an observer. The subject doesn't care; it will deliver notifications to any object that implements the Observer interface.

We can reuse subjects or observers independently of each other. If we have another use for a subject or an observer, we can easily reuse them because the two aren't tightly coupled.

Changes to either the subject or an observer will not affect the other. Because the two are loosely coupled, we are free to make changes to either, as long as the objects still meet their obligations to implement the Subject or Observer interfaces.

Design Principles

1. **Strive for loosely coupled designs between objects that interact.**
   Loosely coupled designs allow us to build flexible OO systems that can handle change because they minimize the interdependency between objects.

Chapter 3 : Decorating Objects: the Decorator Pattern

Design Principles

1.Classes should be open for extension, but closed for modification.
Our goal is to allow classes to be easily extended to incorporate new behavior without modifying existing code. It certainly sounds contradictory at first. After all, the less modifiable something is, the harder it is to extend, right? As it turns out, though, there are some clever OO techniques for allowing systems to be extended, even if we can't change the underlying code. Think about the Observer Pattern (in Chapter 2)...by adding new Observers, we can extend the Subject at any time, without adding code to the Subject. You'll see quite a few more ways of extending behavior with other OO design techniques.Many of the patterns give us time-tested designs that protect your code from being modified by supplying a means of extension.

Decorator Pattern

The Decorator Pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Chapter 4 : Baking with OO Goodness: the Factory Pattern

Factory Pattern is creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created.

Types of Factory Pattern

1. Simple Factory
2. Factory Method

   The Factory Method Pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

3. Abstract Factory

Design Principles

1.Depend upon abstractions. Do not depend upon concrete classes

At first, this principle sounds a lot like “Program to an interface, not an implementation,” right? It is similar; however, the Dependency Inversion Principle makes an even stronger statement about abstraction. It suggests that our high-level components should not depend on our low-level components; rather, they should both depend on abstraction.

Chapter 5 : One-of-a-Kind Objects: the Singleton Pattern

The Singleton Pattern ensures a class has only one instance, and provides a global point of access to it.

Chapter 6 : Encapsulating Invocation: the Command Pattern

The Command Pattern encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.

Bullet points

- The 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.
- MacroCommands are a simple extension of the Command Pattern that allow multiple commands to be invoked. Likewise, MacroCommands can easily support undo().
- In practice, it’s not uncommon for “smart” Command objects to implement the request themselves rather than delegating to a receiver.
- Commands may also be used to implement logging and transactional systems.

Chapter 7 : Being Adaptive: the Adapter and Facade Patterns

The Adapter Pattern converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.

Now, we know this pattern allows us to use a client with an incompatible interface by creating an Adapter that does the conversion. This acts to decouple the client from the implemented interface, and if we expect the interface to change over time, the adapter encapsulates that change so that the client doesn't have to be modified each time it needs to operate against a different interface.
Real World Examples

1. **Iterators**
   The more recent Collection classes use an Iterator interface that, like the Enumeration interface, allows you to iterate through a set of items in a collection, and adds the ability to remove items.

2. **Enumerators**
   If you've been around Java for a while, you probably remember that the early collection types (Vector, Stack, Hashtable, and a few others) implement a method, elements(), which returns an Enumeration. The Enumeration interface allows you to step through the elements of a collection without knowing the specifics of how they are managed in the collection.

The Facade Pattern provides a unified interface to a set of interfaces in a subsystem. Facade defines a higherlevel interface that makes the subsystem easier to use.A facade not only simplifies an interface, it decouples a client from a subsystem of components.Facades and adapters may wrap multiple classes, but a facade's intent is to simplify, while an adapter's is to convert the interface to something different.Facade Pattern allows us to avoid tight coupling between clients and subsystems, and, as you will see shortly, also helps us adhere to a new object-oriented principle.

So the way to tell the difference between the Adapter Pattern and the Facade Pattern is that the adapter wraps one class and the facade may represent many classes? A: No! Remember, the Adapter Pattern changes the interface of one or more classes into one interface that a client is expecting. While most textbook examples show the adapter adapting one class, you may need to adapt many classes to provide the interface a client is coded to. Likewise, a Facade may provide a simplified interface to a single class with a very complex interface. The difference between the two is not in terms of how many classes they "wrap," it is in their intent. The intent of the Adapter Pattern is to alter an interface so that it matches one a client is expecting. The intent of the Facade Pattern is to provide a simplified interface to a subsystem.

Design Principles

- **Principle of Least Knowledge: talk only to your immediate friends**
  But what does this mean in real terms? It means when you are designing a system, for any object, be careful of the number of classes it interacts with and also how it comes to interact with those classes. This principle prevents us from creating designs that have a

large number of classes coupled together so that changes in one part of the system cascade to other parts. When you build a lot of dependencies between many classes, you are building a fragile system that will be costly to maintain and complex for others to understand.

Bullet Points

- When you need to use an existing class and its interface is not the one you need, use an adapter.
- When you need to simplify and unify a large interface or complex set of interfaces, use a facade.
- An adapter changes an interface into one a client expects.
- A facade decouples a client from a complex subsystem.
- Implementing an adapter may require little work or a great deal of work depending on the size and complexity of the target interface.
- Implementing a facade requires that we compose the facade with its subsystem and use delegation to perform the work of the facade.
- There are two forms of the Adapter Pattern: object and class adapters. Class adapters require multiple inheritance.
- 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

---

Chapter 8 : Encapsulating Algorithms: theTemplate Method Pattern

The Template Method Pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.This pattern is all about creating a template for an algorithm. What's a template? As you've seen it's just a method; more specifically, it's a method that defines an algorithm as a set of steps. One or more of these steps is defined to be abstract and implemented by a subclass. This ensures the algorithm's structure stays unchanged, while subclasses provide some part of the implementation**.**
Hooked on Template Method... A hook is a method that is declared in the abstract class, but only given an empty or default implementation. This gives subclasses the ability to "hook into" the algorithm at various points, if they wish; a subclass is also free to ignore the hook.

Design Principles

- **The Hollywood Principle Don't call us, we'll call you.**
  Easy to remember, right? But what has it got to do with OO design? The Hollywood Principle gives us a way to prevent "dependency rot." Dependency rot happens when you have high-level components depending on low-level components depending on

high-level components depending on sideways components depending on low-level components, and so on. When rot sets in, no one can easily understand the way a system is designed. With the Hollywood Principle, we allow low-level components to hook themselves into a system, but the high-level components determine when they are needed, and how. In other words, the high-level components give the low-level components the “don’t call us, we’ll call you” treatment.

Bullet Points

- A template method defines the steps of an algorithm, deferring to subclasses for the implementation of those steps.
- The Template Method Pattern 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 subclass.
- To prevent subclasses from changing the algorithm in the template method, declare the template method as final.
- The Hollywood Principle guides us to put decision making in highlevel modules that can decide how and when to call low-level modules.
- You’ll see lots of uses of the Template Method Pattern in real-world code, but (as with any pattern) don’t expect it all to be designed “by the book.”
- The Strategy and Template Method Patterns both encapsulate algorithms, the first by composition and the other by inheritance.
- Factory Method is a specialization of Template Method.

Chapter 9 : the Iterator and Composite Patterns Well-Managed Collections

The **Iterator design** pattern is a behavioral design pattern that provides a way to access the elements of an aggregate object (like a list) sequentially without exposing its underlying representation. It defines a separate object, called an iterator, which encapsulates the details of traversing the elements of the aggregate, allowing the aggregate to change its internal structure without affecting the way its elements are accessed. Iterator Pattern is a relatively simple and frequently used design pattern. There are a lot of data structures/collections available in every language. Each collection must provide an iterator that lets it iterate through its objects. However, while doing so it should make sure that it does not expose its implementation.

The **Composite Pattern** allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.The Composite Pattern allows us to build structures of objects in the form of trees that contain both compositions of objects and individual objects as nodes. Using a composite structure, we can apply the same operations over both composites and individual objects. In other words, in most cases we can ignore the differences between compositions of objects and individual objects.

Chapter 10 : The State of Things: the State Pattern

The State Pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
The first part of this description makes a lot of sense, right? Because the pattern encapsulates state into separate classes and delegates to the object representing the current state, we know that behavior changes along with the internal state. The Gumball Machine provides a good example: when the gumball machine is in the NoQuarterState and you insert a quarter, you get different behavior (the machine accepts the quarter) than if you insert a quarter when it's in the HasQuarterState (the machine rejects the quarter). What about the second part of the definition? What does it mean for an object to "appear to change its class"? Think about it from the perspective of a client: if an object you're using can completely change its behavior, then it appears to you that the object is actually instantiated from another class. In reality, however, you know that we are using composition to give the appearance of a class change by simply referencing different state objects.

Bullet points

ƒ The State Pattern allows an object to have many different behaviors that are based on its internal state.
ƒ Unlike a procedural state machine, the State Pattern represents each state as a full-blown class.
ƒ The Context gets its behavior by delegating to the current state object it is composed with.
ƒ By encapsulating each state into a class, we localize any changes that will need to be made. ƒ The State and Strategy Patterns have the same class diagram, but they differ in intent.
ƒ The Strategy Pattern typically configures Context classes with a behavior or algorithm.
ƒ The State Pattern allows a Context to change its behavior as the state of the Context changes.
ƒ State transitions can be controlled by the State classes or by the Context classes.
ƒ Using the State Pattern will typically result in a greater number of classes in your design.
ƒ State classes may be shared among Context instances.

Chapter 11 : Controlling Object Access: the Proxy Pattern

The **Proxy Pattern** provides a surrogate or placeholder for another object to control access to it. Use the Proxy Pattern to create a representative object that controls access to another object, which may be remote, expensive to create, or in need of securing.
Well, we've seen how the Proxy Pattern provides a surrogate or placeholder for another object. We've also described the proxy as a "representative" for another object. But what about a proxy controlling access? That sounds a little strange. No worries. In the case of the gumball machine, just think of the proxy controlling access to the remote object. The proxy needed to control access because our client, the monitor, didn't know how to talk to a remote object. So in some sense the remote proxy controlled access so that it could handle the network details for us. As we just discussed, there are many variations of the Proxy Pattern, and the variations typically revolve around the way the proxy "controls access." We're going to talk more about this later, but for now here are a few ways proxies control access:
ƒ As we know, a remote proxy controls access to a remote object.
ƒ A virtual proxy controls access to a resource that is expensive to create.
ƒ A protection proxy controls access to a resource based on access rights.

Remote Proxy, With the Remote Proxy, the proxy acts as a local representative for an object that lives in a different JVM. A method call on the proxy results in the call being transferred over the wire and invoked remotely, and the result being returned back to the proxy and then to the Client.
Virtual Proxy, The Virtual Proxy acts as a representative for an object that may be expensive to create. The Virtual Proxy often defers the creation of the object until it is needed; the Virtual Proxy also acts as a surrogate for the object before and while it is being created. After that, the proxy delegates requests directly to the RealSubject.

Imp Links : Proxy design pattern example

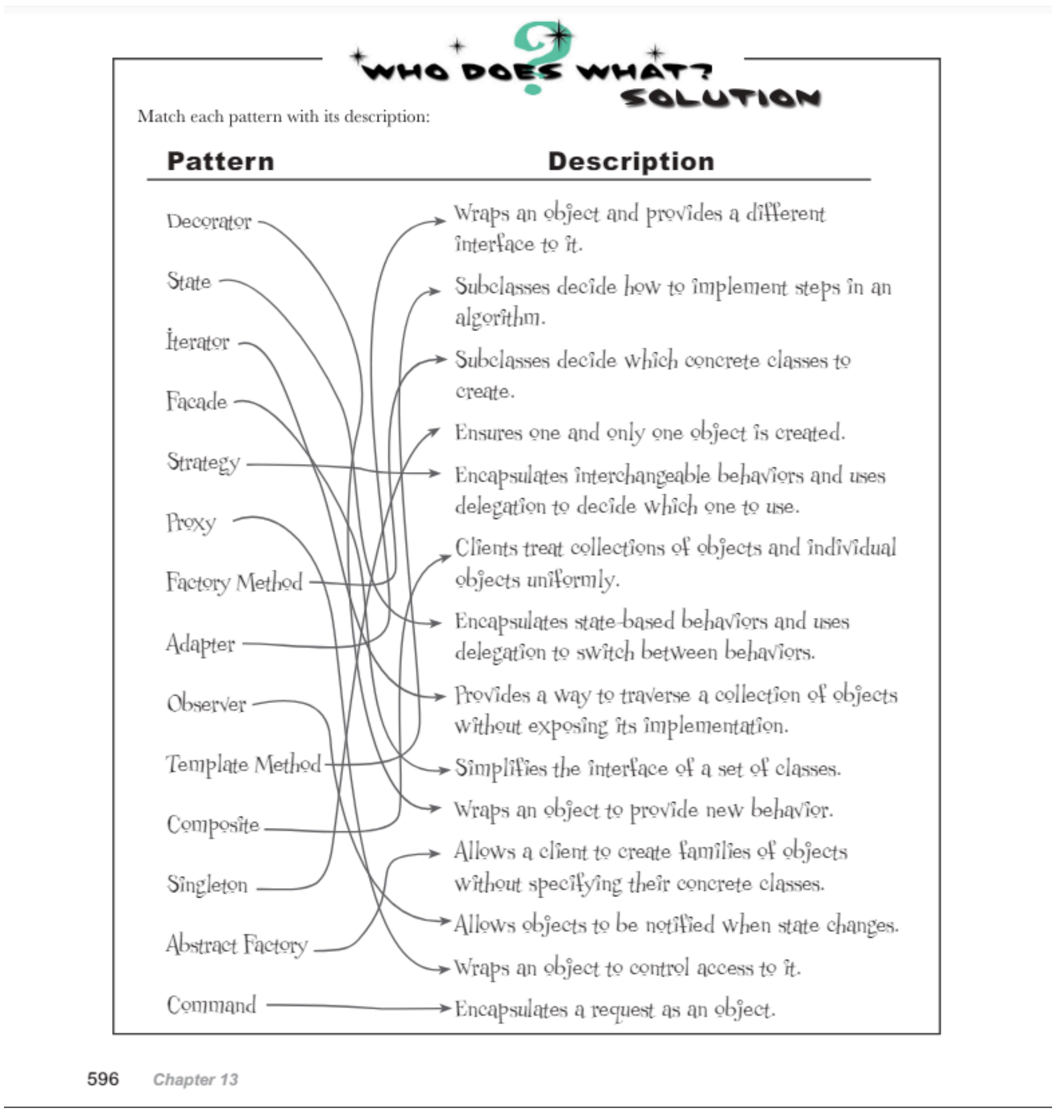
Chapter 12 : Patterns of Patterns: compound patterns

Patterns are often used together and combined within the same design solution. A compound pattern combines two or more patterns into a solution that solves a recurring or general problem. E.g, MVC(Model-View-Controller)

The Model View Controller (MVC) Pattern is a compound pattern consisting of the Observer, Strategy, and Composite Patterns. ƒ The model makes use of the Observer Pattern so that it can keep observers updated yet stay decoupled from them. ƒ The controller is the Strategy for the view. The view can use different implementations of the controller to get different behavior. ƒ The view uses the Composite Pattern to implement the user interface, which usually consists of nested components like panels, frames, and buttons. ƒ These patterns work together to decouple the three players in the MVC model, which keeps designs clear and flexible. ƒ The Adapter Pattern can be used to adapt a new model to an existing view and controller. ƒ MVC has been adapted to the web. ƒ There are many web MVC frameworks with various adaptations of the MVC pattern to fit the client/server application structure.

Chapter 13 : Patterns in the Real World: better living with patterns

A Pattern is a solution to a problem in a context. The context is the situation in which the pattern applies. This should be a recurring situation. The problem refers to the goal you are trying to achieve in this context, but it also refers to any constraints that occur in the context. The solution is what you're after: a general design that anyone can apply that resolves the goal and set of constraints.



WHO DOES WHAT? SOLUTION

Match each pattern with its description:

Pattern

- Decorator
- State
- Iterator
- Facade
- Strategy
- Proxy
- Factory Method
- Adapter
- Observer
- Template Method
- Composite
- Singleton
- Abstract Factory
- Command

Description

- Wraps an object and provides a different interface to it.
- Subclasses decide how to implement steps in an algorithm.
- Subclasses decide which concrete classes to create.
- Ensures one and only one object is created.
- Encapsulates interchangeable behaviors and uses delegation to decide which one to use.
- Clients treat collections of objects and individual objects uniformly.
- Encapsulates state-based behaviors and uses delegation to switch between behaviors.
- Provides a way to traverse a collection of objects without exposing its implementation.
- Simplifies the interface of a set of classes.
- Wraps an object to provide new behavior.
- Allows a client to create families of objects without specifying their concrete classes.
- Allows objects to be notified when state changes.
- Wraps an object to control access to it.
- Encapsulates a request as an object.

596 Chapter 13

Creational Patterns involve object instantiation and all provide a way to decouple a client from the objects it needs to instantiate.
Any pattern that is a **Behavioral Pattern** is concerned with how classes and objects interact and distribute responsibility.
Structural Patterns let you compose classes or objects into larger structures.
Class Patterns describe how relationships between classes are defined via inheritance. Relationships in class patterns are established at compile time.
Object Patterns describe relationships between objects and are primarily defined by composition. Relationships in object patterns are typically created at runtime and are more dynamic and flexible.

Bullet Points

ƒ Let Design Patterns emerge in your designs; don't force them in just for the sake of using a pattern.
ƒ Design Patterns aren't set in stone; adapt and tweak them to meet your needs.
ƒ Always use the simplest solution that meets your needs, even if it doesn't include a pattern.
ƒ Study Design Patterns catalogs to familiarize yourself with patterns and the relationships among them.
ƒ Pattern classifications (or categories) provide groupings for patterns. When they help, use them.
ƒ You need to be committed to be a patterns writer: it takes time and patience, and you have to be willing to do lots of refinement.
ƒ Remember, most patterns you encounter will be adaptations of existing patterns, not new patterns.