




Notes – Design Patterns

- I. Concept
 - a. Design Patterns is a catalog of designs that work well in particular situations.
 - b. They're answers to the question "how do I..."
 - c. Useful as a way of describing designs since everybody knows what they are. ("This is a factory method.")
 - d. Designed by the Gang of Four: Gamma, Helm, Johnson, Vlissides
 - e. Over thirty well-defined, well-known designs exist.
 - f. There's no governing body, just a forum of people who decide when a new design gets added.
 - i. Is it generally useful?
 - ii. Is it distinct?
 - g. Senior developers will know these, since they're just the set of "stuff that's been run into before."
 - h. Now everybody can know them just by getting the book and studying.
- II. Factory Methods
 - a. The rule says: Always write in terms of a superclass.
 - b. You do need to use the correct name when you say "new" though.
 - c. For example, your application may not care what operating system is used, but you still have to instantiate the right thing.
 - d. You may not even know until runtime what you need.
 - e. Consider `java.util.Calendar` by example
 - i. Describes dates
 - ii. Many different calendar types can be represented.
 - iii. Don't want to close the set of possible subclasses or break encapsulation.
 - iv. Nobody cares which type of calendar is being used at any moment.
 - f. The Solution
 - i. Have the superclass instantiate the appropriate subclass from a static method.
 - ii. Create a `getCalendar()` method called a factory method.
 - g. Factory classes are just classes with many factory methods.
 - i. May instantiate many different types of class.
 - ii. If three objects need to work together, but could be {A, B, C}, {P, Q, R}, or {X, Y, Z} but not any other combination, a factory class may be great.
 - iii. Calling three different methods to yield the three different objects would solve that problem.
 - h. Use factory methods whenever multiple choices are classes are available, and where a central piece of code can figure out which one is needed.
 - i. Whenever multiple objects like those described above need to interact, make a factory class.
 - j. Iterators are the most broadly used example of this. Nobody ever knows what class an iterator really is – Iterator itself is just an interface!
- III. Flyweight Pattern
 - a. You usually want EVERYTHING to be an object.
 - b. Objects have a lot of overhead though, which is why int, float, et cetera aren't objects normally in Java.
 - c. One of the biggest complaints about O-O is that it's memory-inefficient.
 - d. Example
 - i. Consider a database for baseball
 - ii. Stores every ball ever played.
 - iii. Want the count, and runners on base to be stored.
 - iv. Can take $4 * \text{balls} + \text{strikes}$
 - v. Can OR $1=1^{\text{st}}, 2=2^{\text{nd}}, 4=3^{\text{rd}}$
 - vi. That works, but it's not a great solution: it exposes rep!

- vii. Want both pieces of data to be objects, but that's a waste.
 - viii. Say there's only 19 possible counts, and only 8 possible runner scenarios.
 - ix. A 2-and-0 count means the same thing for any ball played, so why do you need a SEPARATE object for each?
 - e. Create one copy of each distinct object and store only a reference to it wherever the copy is needed.
 - f. Use a factory method to create the objects as needed
 - i. If one already exists, return it.
 - ii. If none exists, instantiate it and then return it.
 - g. Used wherever there are many references to a limited number of objects.
 - h. The objects MUST be immutable since a change will affect everybody who's using it.
 - i. Say "return a new count going from where I am now to..." to get around the immutability.
 - j. The upside: The == operator works for equality tests. That's cheap!
- IV. Singleton Class
- a. Have an index of items, and want only one instance of any class in the index, or you won't be able to understand the 'lookup' results.
 - b. A singleton class allows only one (or a few) instances of itself to exist.
 - c. Make the constructor private, and use a factory method to instantiate.
 - d. Should be mutable. The point is to have the same changes affect everybody.
- V. State Pattern
- a. The best representation of an instance may depend on what value it currently has.
 - b. Storage for video would be very different for T2 as compared to *Dinner with Andre*.
 - c. May also want to base storage on some recent or known future use.
 - d. Example
 - i. Consider a list.
 - ii. For short lists, it's cheaper to just have an array.
 - iii. For longer lists, may want an array of arrays so it's easier to get to any particular element.
 - iv. "I'll be doing insertions now," so switch to the linked-list representation. "I'll be doing accesses now," so switch back.
 - e. Can't change the representation at runtime, so what can be done?
 - i. Always use the array of arrays.
 - 1. Demands some extra overhead.
 - 2. Not terrible.
 - ii. Use Object for the representation and always cast it to the appropriate current storage.
 - 1. Downcasting is expensive!
 - 2. It's worse than the savings gained.
 - 3. The code is harder to read too.
 - 4. The compiler can't really help you anymore since it doesn't know what representation you really will be using at any given moment.
 - iii. Could just create a new object.
 - 1. Change the list each time, return a new list after every insert/remove operation
 - 2. If the representation is shared, that falls apart.
 - f. The Better Solution
 - i. Create an abstract ListRep superclass, private to List
 - ii. SmallListRep and LargeListRep would be subclasses.
 - iii. Can then change at will, without having to downcast. (ListRep has all the same behavior so the code doesn't care what's happening behind the scenes)
 - g. The Rub
 - i. This is only worthwhile when there's a significant performance gain.
 - ii. Never implement it on the first try. Stick with the simple list, then see where performance needs to be improved.

- iii. This would *usually* be used only for immutable classes.
- VI. Bridge Pattern
 - a. Consider Image as a superclass of JpegImage and GifImage
 - b. Now you want to add ScalableImage as a subclass of Image.
 - c. Why can't you have a Scalable JpegImage?
 - d. You don't want to create ScalableXXX subclasses for every single image class that already exists.
 - e. You want the behavior hierarchy to be separate from (and parallel to) the implementation hierarchy.
 - f. Separate the choice of behavior from that of representation
 - g. The Rub
 - i. Now you've got two hierarchies, which vastly adds to the complexity.
 - ii. You only need the bridge pattern when the implementation and behavior need to both vary independently.
- VII. Strategy Pattern
 - a. In code, you may want to decide what behavior is appropriate, and then do it.
 - b. Simplest Form
 - i. One function specifically makes the decision, and then do it.
 - ii. There's no way to get the answer back, the "do it" has already been executed.
 - c. Want to think of procedures as "first class objects" (can reference distinctly)
 - d. Possible Implementation
 - i. Map behaviors to integers.
 - ii. Then you're restricted to whatever ideas were already hardcoded.
 - e. Best Approach
 - i. Make classes with one operation (like Runnable)
 - ii. Can pass the class around and eventually someone says "Okay, do your thing."
 - iii. The class also needs a way to store arguments ("WHERE to move?")
 - f. For Assignment 4
 - i. Let chooseStrategy() return a class as described.
 - ii. The caller then calls .perform() to execute the action.
- VIII. Command Pattern
 - a. Once behaviors are first class operations, other opportunities arise.
 - b. Imagine trying to compile a giant program.
 - c. Want to use all available machines.
 - d. Keep a list of what needs to be done, and let machines pull jobs off the list.
 - e. Compilers need to get a complete package of what needs to be done.
 - f. Can develop separate Compile, Link, Test objects and put whatever's needed in the queue. The machines don't care what they get.
- IX. Composite Pattern
 - a. Consider container objects
 - b. Containers may contain other containers.
 - c. Composite pattern describes tree-like structures
 - d. There are many different ways to express trees, but this has some advantages
 - e. Composite is a subclass of Component, so a Composite can be a leaf itself.
- X. Visitor vs. Interpreter Pattern
 - a. Interpreter Pattern
 - i. What will be done with the tree? Sometimes only the top needs to be touched, sometimes every single leaf.
 - ii. Let Composite draw all children, and let each node draw itself, for example.
 - iii. It's easy to add new node types then – just define all the right behaviors for each.
 - iv. It's hard to add new behaviors since every single node type would have to be modified.
 - b. Visitor Pattern

- i. Write a Visitor class for each possible behavior.
 - ii. Just send the Visitor to each node and let it do its job.
 - iii. That removes the ability to make different types!
 - 1. Use double dispatching.
 - 2. In Visitor, write a method that's specific to each type.
 - iv. The visit() method on each node calls the appropriate method on Visitor.
 - v. Now it's really easy to add actions since each is local to a single class.
 - vi. Now it's really hard to add new node types since their code is spread among all types of Visitors.
 - c. Which to Use?
 - i. If it's not obvious, use interpreter. It's probably the right choice.
 - ii. More often you'll want to add node types, so Interpreter is the best option.
 - iii. If you plan to add actions more often, use the Visitor pattern.
- XI. Adaptor Pattern (wrapper object)
- a. Code is rarely written in isolation. You want to integrate it with existing systems.
 - b. You want an intervening object to talk from the new system to the old.
 - i. Maybe it just converts datatypes.
 - ii. Maybe it changes / translates the names of things.
 - iii. Maybe it translates levels of data.
 - iv. Maybe it filters unneeded data.
 - c. This is probably the most obvious pattern. It's easy to see when you're faced with a problem that this is what you need.
 - d. The Rub
 - i. The two interfaces need to have enough similarities that it's possible to talk back and forth.
 - ii. Cannot get more precision in data.
 - iii. Cannot increase the granularity of control.
- XII. Decorator Pattern
- a. Subclassing can be used to add behavior.
 - b. What if you want to add behavior to many superclasses?
 - c. Consider scrolling. Have many User Interface objects that all want scrolling, but don't want all the extra code.
 - d. Wrap the class, rather than subclassing it.
 - e. Instead of making it a subclass, just make a class that USES the original component.
 - f. The Rub
 - i. Now you can't use the specific operations of the original object since it's hidden inside a scrolling gadget.
 - ii. The scroller is just a scroller and doesn't know what text boxes do.
 - iii. Need to also keep a reference to the original component to get ITS functionality.
 - g. Usage
 - i. Use whenever you want to add behavior to a group of existing classes, including unknown classes.
 - ii. The wrapper becomes the top-level view. If that's not necessary, just put an object in the representation that can handle the functionality you want.
- XIII. Observer Pattern
- a. Decorator and Adaptor are both types of indirection
 - b. Consider a computer with a wired and wireless connection. Applications don't care which is being used at any particular moment.
 - c. Applications also don't want to keep asking which to use for each network access.
 - d. Instead, apps can go through a proxy
 - i. ONLY the proxy knows which NIC should get the traffic.
 - ii. Proxy  "Does the Work For You"
 - e. The Rub
 - i. There's a small performance hit.
 - ii. There's a small increase in design complexity

- f. Usage
 - i. One thing on one side, many on the other.
 - ii. You might want to pick one thing from the many, like in the example.
 - iii. You could also have many objects that want to receive information about the application.
 - iv. Each would then register interest in observing and the proxy would send information to all those so registered.
 - g. How to get Information to the Observer?
 - i. Could have generic Subject, Observer interfaces.
 - ii. Could build the functionality into specific classes.
 - iii. Pull Model
 - 1. The observer grabs data specifically whenever it's needed.
 - 2. The conscious action is by the observer, the subject just responds.
 - iv. Push Model
 - 1. Send an object storing all the new information.
 - 2. The subject is then responsible for taking the action.
 - h. Figuring out that you need the observer pattern is easy.
 - i. Push vs. Pull is harder.
 - i. Use push when everybody wants the same data, so you only need to send what's useful.
 - ii. Use pull when each observer may want to grab different data.
 - j. It's better to be general and use generic Subject, Observer interfaces.
- XIV. Mediator Pattern
- a. The observer pattern has some messy points
 - i. The subject adopts some of the work involved in doing the observation, which takes away from its real responsibility.
 - ii. The subject also learns who the observers are, which may not be a good thing!
 - b. What if there's more than one subject?
 - c. Consider SETI@home by example.
 - d. The observers may not even care which subject produced the data.
 - e. It might be nice to tag the results so they can be identified later, but the reaction to generation of new data doesn't differ based on which subject generated it.
 - f. The observer pattern could almost be used, but it's too messy to cross-register many subjects with many observers.
 - g. Instead, add a Mediator.
 - h. Subjects register with the mediator as data producers.
 - i. Observers register that they're interested in some particular type of data.
 - j. When subjects send new data, it's dispatched to all waiting observers.
 - k. This is also called publisher/subscriber (pub/sub)
- XV. Summary
- a. Design patterns are "not the panacea"
 - b. They have strengths and weaknesses, like any design ideas do.
 - c. You need to understand why you want a particular pattern, what the details are (all those little choices that come after picking a pattern), and what the drawbacks are.
 - d. These are just ideas of ways to solve the problem.
 - e. Don't limit yourself to this particular subset of ideas.