“Head First Design Patterns” (2004)

Notes open for creative commons use @ developer blog: https://unfoldkyle.com, github: SmilingStallman, email: kmiskell@protonmail.com

Intro to Design Patterns

-Design patterns are pro-active. They solve problems before they occur with the time honored solutions others have already developed. No need to rebuild the wheel, etc..

-Simple OO design with largely just inheritance creates many problems as very often a parent class can be defined with methods/attributes that should be inherited by most children, while a few outlying, or other future children should not inherit those methods/attributes.

-Concrete methods in abstract classes can cause similar problems. What if future behavior changes and abstract class concrete methods now need to be changed? All extending classes will also feel those changes.

-Design patterns are specific patterns (ex. factory), while principles are more overarching guidelines.

-<u>Design Principle</u>: Identify the aspects of the application that vary and encapsulate them from what stays the same.

-<u>Design Principle</u>: Program to on an interface, not an implementation. If *Dog* inherits from *Animal*, when creating a *Dog* object, create it as *Animal animal = new Dog()*. As this gives more flexibility, as the object can now be passed as an *Animal* arg, etc..

Behavior Separation Ex.
1) Start with *Duck* class that has a couple methods shared by all children. After looking at it, *fly()* and *quack()* might not apply to all ducks.

2) Create *FlyBehavior* and *QuackBehavior* interfaces to separate these responsibilities, where one has *fly()* and the other *quack*.

3) Create classes for all behavior interfaces with different implementation depending on desired behavior. Ex. *FlyWithWings implements FlyBehavior*.

4) Add not instantiated *FlyBehavior* and *QuackBehavior* objects in *Duck* class through composition. Then, create wrapper functions to call their *fly()* and *quack()* methods.

5) Any class that extends *Duck* will then instantiate the *Duck FlyBehavior* and *QuackBehavior* classes with the proper behavior based subclasses. To decouple instantiation of the behavior classes from the *Duck* class definition, *Duck* can instantiate those classes by reading in behavior types during composition.

The result is a *Duck* object that can have many implementations, without tying those implementations to the class, through composition and separation of behaviors. If another animal needs to fly, an implementing *FlyBehavior* class could potentially be used by both *Duck* and the new animal. By composing based on interfaces, any class implementing that interface can be instantiated for a *Duck* object, providing easy addition of new behaviors, through new implementations, etc..