“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: The Strategy Pattern

-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.

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

-Design Principle: 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. via Strategy Pattern
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*.

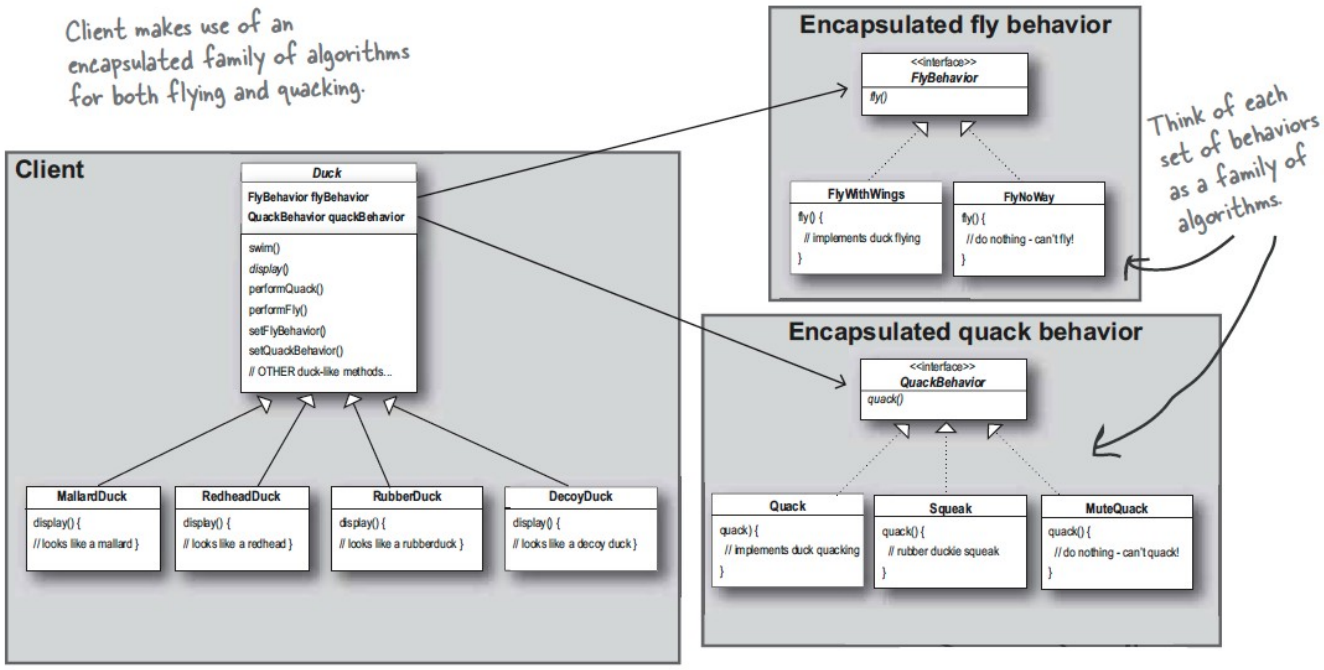
3) Create classes for all behavior interfaces with different implementation depending on desired behavior. Ex. *FlyWithWings implements FlyBehavior*.

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.. The *set* methods for the behaviors also allow behaviors to be set dynamically, including after creation (ex. duck can no longer fly due to broken wings and now must walk).



Composition vs Inheritance
-Inclusion of behavior objects in duck class is an example of composition. Inheritance itself creates dependencies, while dynamically composed objects do not, and allow runtime initialization. Favor composition over inheritance.

Strategy Pattern

-Strategy pattern: Strategies define behaviors used to solve a problem. Context class includes, sets, and uses strategies. Allows class behavior and algorithms to be set dynamically set during runtime by switching out which strategies are used by context class.

-In strategy pattern:
1) Build strategy interface which declares required behavior methods
2) Build concrete classes, each with their own implementation of strategy.
3) Build context class which needs behavior, which has un-initialized strategy object. Context class also has method that reads in strategy type and sets strategy object to that type via *new* object. Lastly has wrapper function(s) which wrap(s) call(s) to strategy method(s).
4) To use, create instance of context class, then use its creation method to initialize strategy with desired behavior, then call behavior via wrapper method.

-In above example, *AnimalNoise* and *AnimalMove* are the strategies (behaviors). Child classes like *MoveWalk* and *Bark* are implementations of the strategies. The context class is *Duck*. Various *Duck* subclasses extend *Duck* and call the *setBehavior* methods during their construction. To set their desired behaviors. To change their behaviors during runtime, they can call the *setBehavior* methods again with new behavior types.

-Summary: A strategy is a general action (interface) used to solve a problem, implemented through a variety of solutions (implementing classes). A context is a class that wishes to solve the problem a strategy (implementing class) handles and may wish to switch solutions during runtime. The strategy pattern thus seperates the strategies from the context, where the context knows it will use a general strategy (uninitialized strategy contained by context), but does not need to determine which implementation of that strategy (via *set* method) to use until composition, and allows the strategy to be changed during runtime (also via *set* method). The context class also wraps calls to the strategies' methods, so the user of the context class never has to directly reference the strategies, creating a uniform context interface.

Shared Vocabularies
-Design patterns give developers a shared voculary, allowing uniform communication in how they code and design.

-Instead of saying, "I made this complex thing," you can just say, I made this thing using *x* pattern, and other devs will already have a good understanding of how it might work.

-Design patterns help you think on a system level instead of a one-off level

-Patterns are built on pro-active solutions to problems. By using design patterns, people will know what you are trying to solve in part by the patterns used.

Using Design Patterns
-Design patterns are just patterns used to give a specific type of problem with a desired problem. The more you use design patterns, the quicker you will be able to recognize which patterns apply well as solutions for the problem at hand.

-Frameworks and libraries are often built based on frameworks (ex. MVC). By learning the patterns, you can learn frameworks/libraries quicker too, and have a proper understanding of how and why they work the way they do.

Design Toolbox I
-OO Basics: abstraction, encapsulation, polymorphism, inheritance

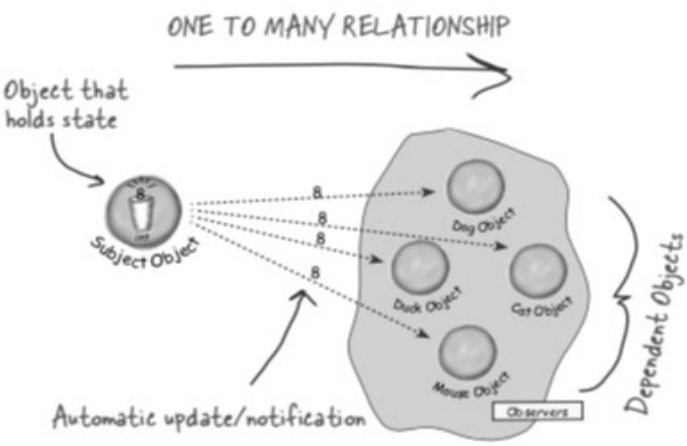
-OO Principles: encapsulate what varies, favor composition over inheritance, program to interfaces not implementations

-OO Patterns: Strategy – seperate the behaviors from the class using them and allow the class to switch the behaviors used as it needs to.

Keeping Your Objects in the Know: The Observer Pattern

-Behavioral pattern

-Consists of subject object, observer object(s), and client object. Subject manages changing data. Observer is registered with the subject to receive updates when the data changes in/by subject. Single subject can have many observers (one-to-many dependency), all observers updated on change, and observers can be dymanically removed. Client creates subject, then creates observers, passing in subject to observers during definition to register subject to observer.

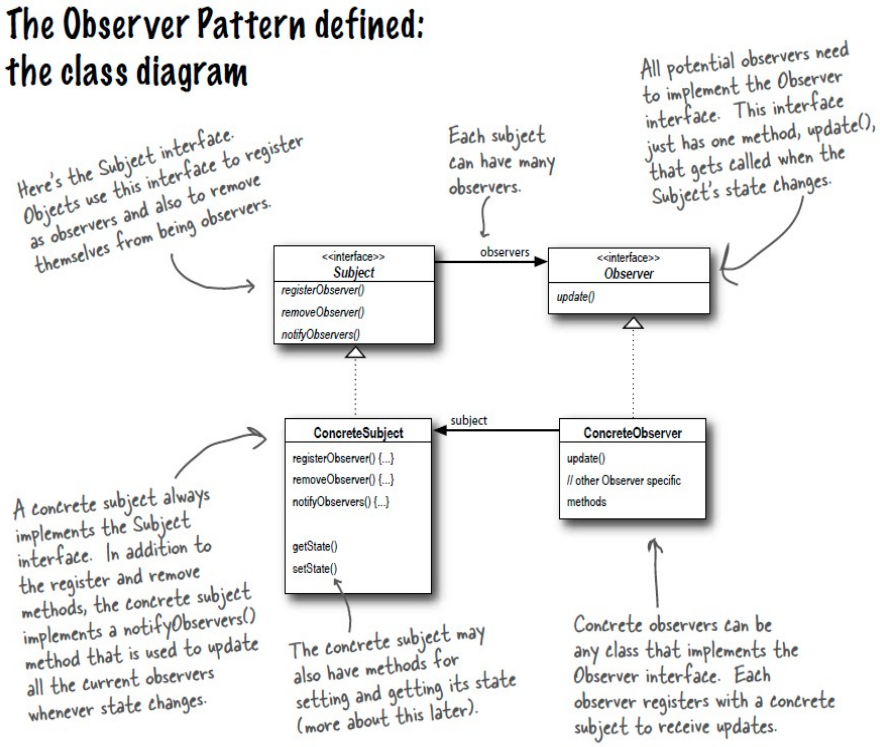


-Observers dependent on subject, as subject provides updates to observers

-Subject handles *registration*, *removal*, and *notify* methods, as well as *setState* and *getState* methods (often delicated to implementation). Also has list of observers to notify.

-Observer object has attributes for data sent from subject and *update* method, which is called by subject's *notify* method. Observer also has subject object, which initializes (registers) during construction.

-Typically interface for observer and interface for subject (program to interface), with concrete subject and observers implementing



Loose Coupling

-Loose coupling means interaction, without the objects needing much knowledge on each other

-In observer pattern, subject only knows the observer interface and does not care about concrete implementations of. New implementations can easily be added.

-Subject also only depends on list of observers to update, all in the same way, meaning new observers can be added, or existing observers can be replaced during runtime

-Observers and subjects exist independtly and can be re-used (subjectA notifies observerB, subjectB

also notifies SubjectB as well as Subject C, etc.)

Design Principle: Strive for loosely coupled design between objects that interact