“Dive Into Design Patterns” (2019)

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

Basics of OOP

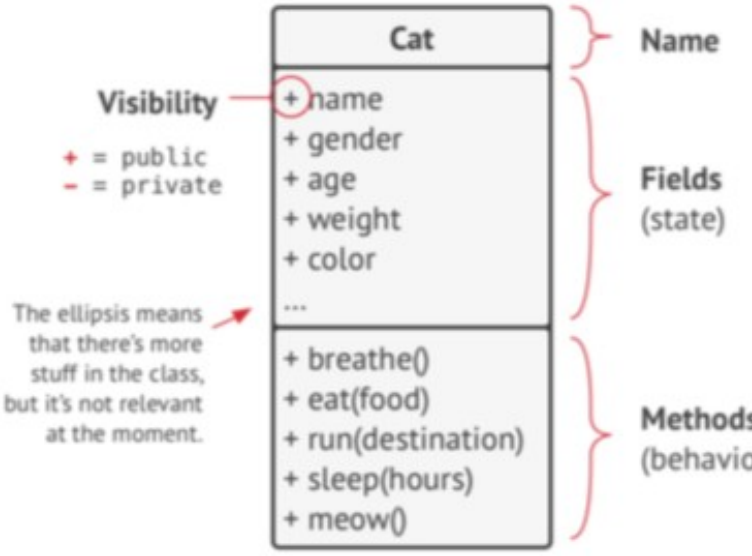
-Notes in this section are simply a brief review of OOP principles. For much greater detail, read my, “The Object Oriented Thought Process,” notes, which are 16 pages in length, instead of the 2.5 pages seen in this section.

-Object-oriented programming – paradigm built on wrapping data and behavior together in classes, which act as blueprints for objects (class instances)

-Behaviors are class methods. State is attributes.

-Encapsulation in part through *public* (viewable to all), *protected* (viewable to containing class and extending classes only), and *private* (viewable to containing class only) members

-UML diagrams commonly used to represent classes and relationships:



-More UML: Abstract classes and methods in italics

-Classes can also inherit from other classes, in which an inheriting class gets the same state and behaviors (as long as not *private*) of the class inherited from. An *is-a* relationship.

-Inheriting classes can also add their own state and behaviors, not seen in inherited class, as well as override methods seen in parent class, in which methods have same signature, but different implementation

-Inheriting class is called *child* class or *subclass*. Class inherited from is called *superclass* or *parent* class.

-Inheritance builds *class hierarchies*

-Establishes *contracts* in which classes must adhere to specific guidelines, by being required to implement all methods in an interface they implement from, all abstract methods in an abstract class they extend, etc.

-By programming to interfaces, instead of implementation, encapsulate implementation from interface, in which implementation can change or be seen in many forms, without breaking existing code, as long as interface remains unchanged.

Pillars of OOP

-Abstraction – a model based on a real-world object or responsibility, but abstracting out only the needed state and behavior of that object, omitting what is not needed. "To perceive an entity in a system or context from a particular perspective."

-Encapsulation – information hiding. Hiding what does not need to be known and presenting only what does. The interface of an object is shown, while the implementation is hidden. Encapsulate through *private* (fully hidden) and *protected* (also available to subclasses) members and state. Interfaces and abstract objects/methods both provide encapsulation

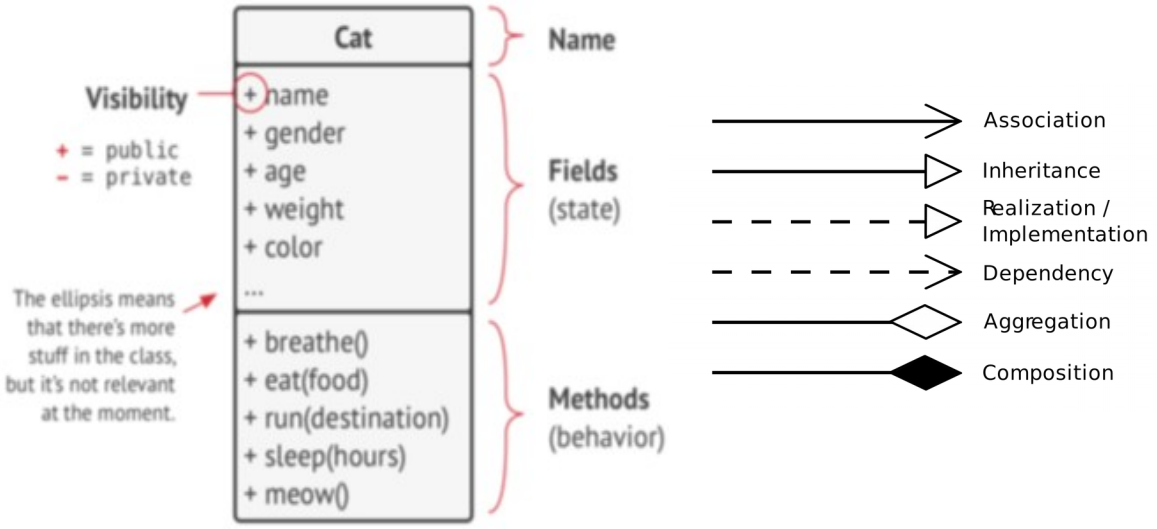
-Inheritance – The ability to build new classes based on existing classes. Code reuse, where simply extend existing class and add onto it, override, etc. in subclass. As subclasses get same (*public/protected*) state & behavior as parent class, also get same interface, allowing child object to be treated same as parent. Typically can only inherit from one superclass, but can implement many interfaces.

-Polymorphism – The ability to take many forms. Allows a child class to be referenced as if it was a parent class (it is, thanks to *is-a* relationship) in code. An child class can "pretend" to be a parent class, since it actually also is one.

-ex.

```
beasts = [new Cat(), new Dog()]          //Cat and Dog both inherit from Animal
foreach(Animal a : beasts)
    a.makeSound;
```

Object Relationships



-Concrete class *implements* interface, inherits from other concrete or abstract class, and *implements* abstract methods

-Association – when an object uses or interacts with another object. Composition and aggregation both types of association. Ex. Professor communicates with student.

-Dependency – when the functionality of one class is dependent on another class, a dependency is present. A dependency exists if changes in one class require changes in another class. Ex. Lesson is dependent on curriculum.

-Composition – when an object contains other objects as "whole-part" members. The contained objects exist only within the containing object, only to serve as part of the whole. *Has-a* relationship.

-Aggregation – object contains a reference to another object, typically passed in, that can thus also exist outside the containing object. *Has-a* relationship.

UML Composition. University consists of departments.

UML Aggregation. Department contains professors.

Basics of Patterns

-Design pattern – typical solutions to commonly occurring problems in software design. Pro-active solutions to solve re-occurring problems.

-Differ from algorithms as algorithms are more, same in, same out, expressions with a clear set of actions to achieve a goal. Design patterns are more conceptual, where they are more like a blueprint, that can have greater difference in implementation, as long as the pattern concept is adhered to.

Parts of Pattern
-Intent – describes the problem and the solution of the pattern
-Motivation – why the pattern is a proper solution to the problem
-Structure – the parts of the pattern and their relationships

-Also often come with code example, as real world example of pattern, to help assist in comprehension

Types of Patterns
-Idioms – very basic and low level patterns, often only applicable to a single language
-Architectural patterns – patterns for how to build and relate classes, applicable to most any language

-Creational – provide object creation mechanisms that increase code flexibility and reuse

-Structural – how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient

-Behavioral – handle effective communication and assignment of responsibilities between objects

Important People in Patterns
-Christopher Alexander in "*A Pattern Language: Towns, Buildings, Construction*," is thought as the first important text, in which he designs a language for construction, in which the units of this language are repeatable patterns

-The Gang of Four in their 1994 book, "*Design Patterns: Elements of Reusable Object-Oriented Software*," which specs out 23 patterns, which largely compile the main OOP patterns still used today. GOF.

Why Patterns?
-Patterns are tried and tested solutions that just work. Knowing patterns lets you know solutions, and gives you quick and efficient methodologies to solve common problems.

-Patterns also define a common language amongst developers. Trying to explain your complex solution to another team can be difficult, but if the solution can be abbreviated with something like, "I built the program as an MVC to…," this makes it much easier to understand, due to shared pre-existing knowledge.

Software Design Principles

Code Reuse
-Results in less time to develop, meaning lower cost also. Why rebuild the wheel each time when much can just be reused?

-Using design patterns help keep code properly decoupled, with less dependencies, more reliance on interface instead of implementation, etc., and thus assist in making code more re-usable and flexible.

-Code with high complexity has many components. Complicated code has a higher level of difficulty, often among less components.
-While complexity means more relationships to manage, etc., it also generally results in more loosely coupled and flexible code (why The Zen of Python states, "complex is better than complicated."
-Design patterns help reduce complication, even at the cost of greater complexity

Levels or reuse:
low) reuse of classes through class libraries, containers, iterators, etc.

mid) patterns – smaller and more abstract than frameworks. More reuse than low.

high) frameworks – represent key abstractions for solving problem and define relationships between them. User the hooks into framework, subclasses, etc., and defines custom behavior. More reuse than mid.

-While frameworks require a significant investment and are often all encompassing for a project, patterns can be used as more selectively, where conceptually makes sense.

Extensibility
-Provides the ability to easily add to the existing code without requiring much change (or any change if following open-closed principle)

Universal Design Principles
I) Encapsulate what varies
II) Program to an interface, not implementation
III) Favor composition over inheritance
IV) SOLID

Encapsulate what varies
-Identify the aspects of the application that vary and separate them from what stays the same

-Doing this separates the code into different components. This lesses the effect of change, but reducing coupling, and properly separating responsibilities.

-Methods should also properly encapsulate responsibility. A method should do a single thing. A *get* method should not also be calculating, just getting, etc.. Extract what is not part of the single responsibility into a new method.

-Classes should be further broken down they become too large. A class should represent a single model or have a single responsibility. If it finds itself serving another responsibility, through a bunch of helper methods, calculation methods, etc., break it up into multiple classes.

Program to an Interface
-Program to an interface, not an implementation. Depend on abstractions, not concrete classes.

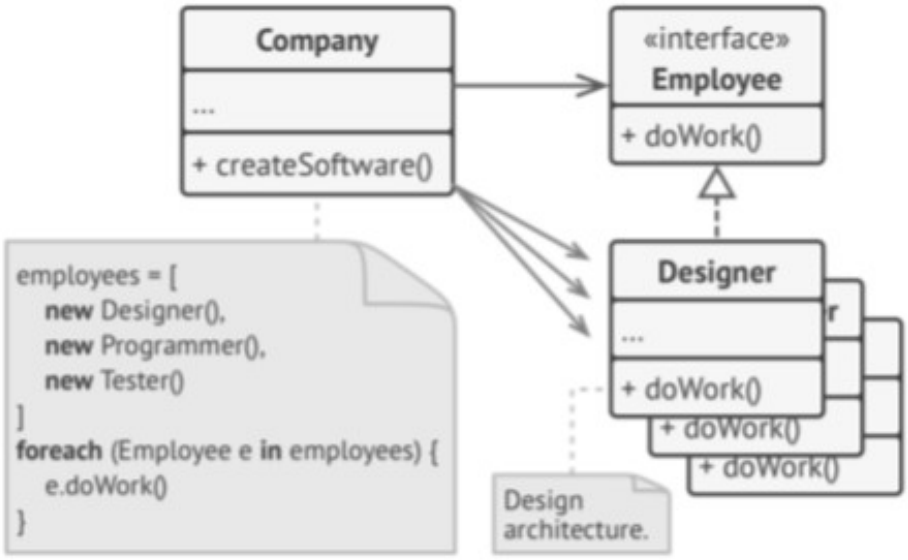
-By programming to an interface, the implementation can change or have as many variations as desired, and as long as the interface stays the same, the classes using it don't really care.

-All methods used by other classes, should be defined in an interface or abstract class, then implemented by a concrete class(es). Classes that use these class should then be dependent on this interface/abstract instead of the concrete.

-When designing, first get the requirements, then build the interface, then implement it, then review and see if more should be moved to the interface, etc.

-Though programming to an interface builds more complexity, it pro-actively adds more flexibility, which will be much appreciated as the system expands and

-Programming to an interface also builds a uniform means to call all classes that implement it, allowing many classes with different responsibilities to be called in the same manner and ensure all will work through their shared contract



-Above example doesn't care about the type of object in the for loop. As long as they are all *Employees,* the interface is the same, so it knows it can call *doWork* on them.

-Abstract classes provide similar benefits, by allowing a parent class to hold the shared behaviors, where the unshared behaviors are defined as abstract methods, then implemented as needed by concrete classes

-Methods which are dependent on other classes can be separated from an abstract class into concrete classes to further add flexibility and reduce dependencies by removing the existence of the classes the abstract class is dependent on into specialized subclasses

Favor Composition Over Inheritance
-Composition (and aggregation) with properly selected items allows more selectively than inheritance which forces implementation

-Inheritance forces same signature methods, which constrains child classes, while composition allows for selective behavior addition

-Inheritance breaks superclass encapsulation by making members and state available outside of class (to subclasses)

-Tight coupling of subclasses to superclasses

-Inheritance might actual result in duplicate code, from shared behavior not existing in a parent class, but existing in a child class of that parent class that does not apply to be a child of another parent classn

-Trying to re-use code through inheritance hierarchies can lead to parallel inheritance hierarchies, where an inheritance tree depends on an inheritance tree depends on another inheritance tree through composition

-Main reason is selective behavior, implemented through inclusion as needed, as a opposed to forced behavior, as seen with inheritance

SOLID

-Five basic guidelines covering much of already covered above in an easy to remember mnemonic set, intending to keep code understandable, flexible, and maintainable. Do not expect to apply all five in all code, but use as reference, and when proper tools for the job.

-Single Responsibility Principle - "A class should have just one reason to change." Keep classes to a single responsibility, responsible to a single business actor (only one actor could request it to change). Keeps code pro-actively small by preventing massive many responsibility classes. Also decouples responsibilities, reducing cost of change, where change in one responsibilities has little no effect on other responsibilities.

-Open Closed Principle - "Classes should be open for extension but closed for modification." Once a class hits prod, it is closed; don't change that class, and instead add or modify behavior via additional subclasses, with diff implementation, overriding, specialized state, etc.. Exists as if no change in code, no change needed to code using it. Questionable principle, as forces inheritance.

-Liskov Substitution Principle - "When extending a class, remember that you should be able to pass objects of a subclass in place of objects of the parent class without breaking the client behavior." Breaking this removes a lot of flexibility that comes from polymorphism. Requires subclasses and overriden methods to:

I) have same signature as parent methods
II) not throw different exceptions from matching parent class methods
III) not have stronger pre-conditions for inherited methods
-ex. of breaking: inherited method requires args passed into method to be int while parent does not
IV) not weaken post-conditions for inherited methods
-ex. of breaking: parent method closes DB on exit while inherited method does not
V) preserve invariants all invariants of superclass, where invariant is a rule that describes properties and relationships of a class that must remain constant during class
VI) If possible in the language, not change the values of *private* fields of parent class(es)

-Interface Segregation Principle - "Clients shouldn't be forced to depend on methods they do not use." Keep interfaces small. Since classes can implement many interfaces, easy to both segregate and use multi responsibilities. Keep a balance to not divide more than needed too, though.

-Dependency Inversion Principle - "High-level classes shouldn't depend on low-level classes. Both should depend on abstractions. Abstractions shouldn't depend on details. Details should depend on abstractions." Where low-level classes implement basic logic, and high-level classes contain complex business logic that use low-level classes to perform actions. Instead of building low level first, change

the direction of the dependency, by first designing interfaces based on required low-level logic. High and low level classes thus depend on abstraction and details of low level class is based on abstraction (interface). Allows high level classes to use many low implementations, where all implementations fulfill same abstraction.

Creational Patterns - Factory Method

-Provides a method for creating objects in a subclass, where subclasses can also alter the types of objects that will be created

Problem
-High level code relies on one class (ex. shipping truck) but another class with similar responsibilities is now required (ex. shipping ship) and other classes may be required in the future.

Solution
-Instead of creating objects in the high-level code via *new* in the high-level constructor, etc., create inside an inherited or abstract factory method instead, which hides the *new* logic, and returns an object of an abstract type.

-Implementations of factory method in subclasses can now return any type of concrete object implemented by abstract object.
-Factory method in parent and child classes can also return default type.

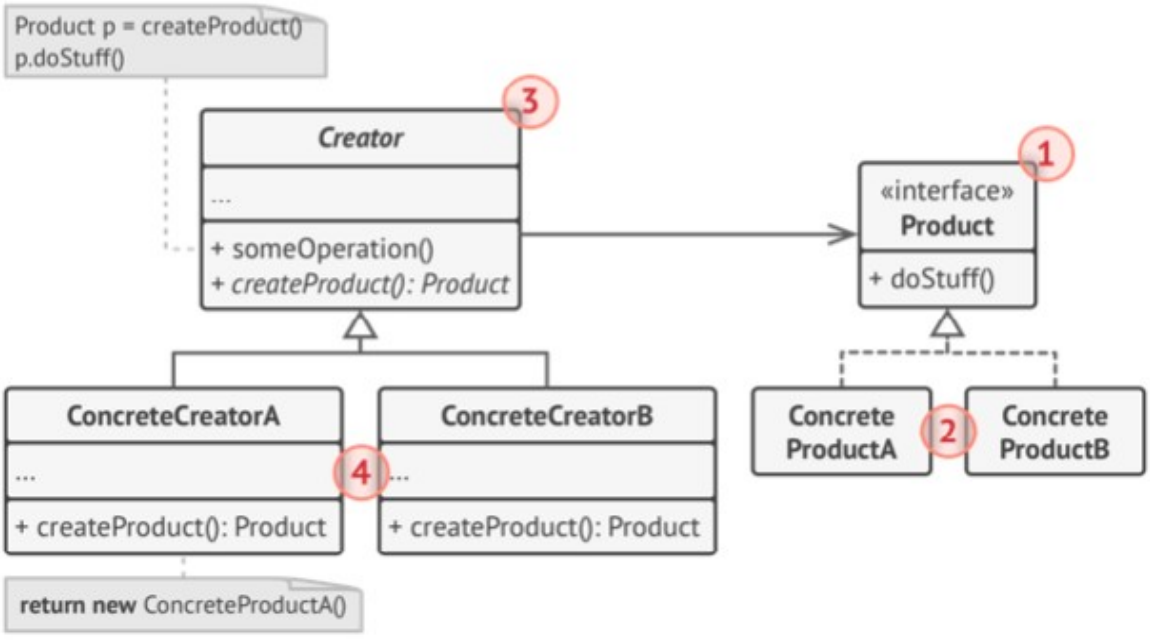
-To make sure all implementations of factory method return same base type, parent factory method should return type of root abstract type (interface, abstract class, etc.).

Structure
1) *Abstract product* declares interface (either via abstract class or interface), implemented by items to be returned by the factory

2) *Concrete Product* implement A*bstract Product*

3) *Creator* contains factory method that returns *Abstract Product*. Can be declared abstract to force implementation in child classes. Can also not be abstract, and return default C*oncrete Product* (though signature should still declare to return A*bstract Product*. Often an abstract class.
-Despite name *Creator* main responsibility is not to create, and usually has other core business logic besides factory method. Since it holds factory method, called *Creator* in pattern specs, though.

4) *Concrete Creator* – optional implementations of the *Creator*, which can each return different



C*oncrete Product* types
-factory method can return any object, not just *new* objects, such as object stored in DB, etc.

Application
-Use when don't know beforehand what exact types of objects code might want to work with

-Use to provide users of your library or framework to extend its internal components. This is done by reducing component creation across framework to single factory method. Users would then create by creating and extension of the factory method return type, and returning it from a class that implements the factory method.
s
-Use to save sys resources by reusing existing objects instead of rebuilding each time, with factory method that can return *new* or existing objects

Implementation
1) Build *Abstract Product* interface for all potential *Concrete Product* types

2) Add factory method to class needing to use *Product*s (the Creator) that returns type *Abstract Product* interface

3) Replace all existing new object creation *new* calls with calls to factory method.

4) Create one concrete class for each *Concrete Product* type, implementing factory method. If too many types, can combine into "genres" and implement class that can return multiple types

5) If no default return needed by base factory method, make *abstract* metho

Pros and Cons
-Pro – removes coupling between *Creator* and *Products*

-Pro – can move production creation code to *Concrete Creators*, thus fulfilling single responsibility principle

-Pro - Can introduce new *Products*, while also fulfilling open-closed principle, by creating new *Concrete Creators* and new *Concrete Products*

-Con – pattern requires lots of subclasses, so adds complexity