

# “Dive Into Design Patterns” (2019)

Notes open for creative commons use @ developer blog: <https://unfoldkyle.com>, github: SmilingStallman, email: [kmiskell@protonmail.com](mailto:kmiskell@protonmail.com)

## Basics of OOP

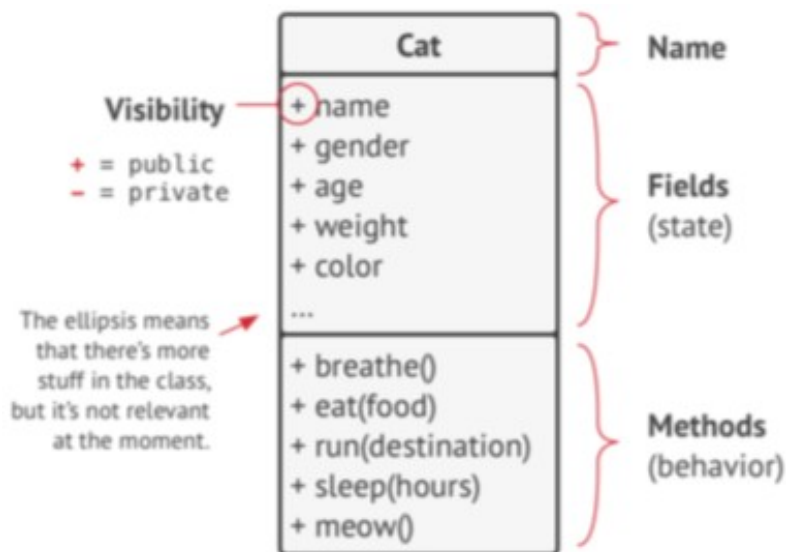
-Notes in this section are simply a brief review of OOP principles. For much greater detail, read my, “The Object Oriented Thought Process,” notes, which are 16 pages in length, instead of the 2.5 pages seen in this section.

-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

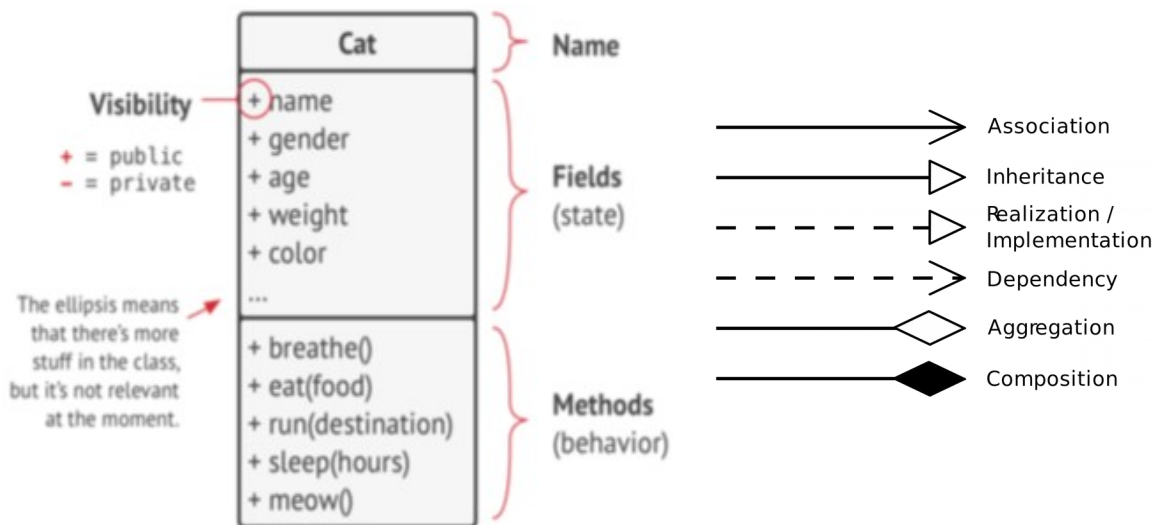
-Inheritance – The ability to build new classes based on existing classes. Code reuse, where simply extend existing class and add onto it, override, etc. in subclass. As subclasses get same (*public/protected*) state & behavior as parent class, also get same interface, allowing child object to be treated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

## Intro to 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 of 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. Use 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 lessens 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.

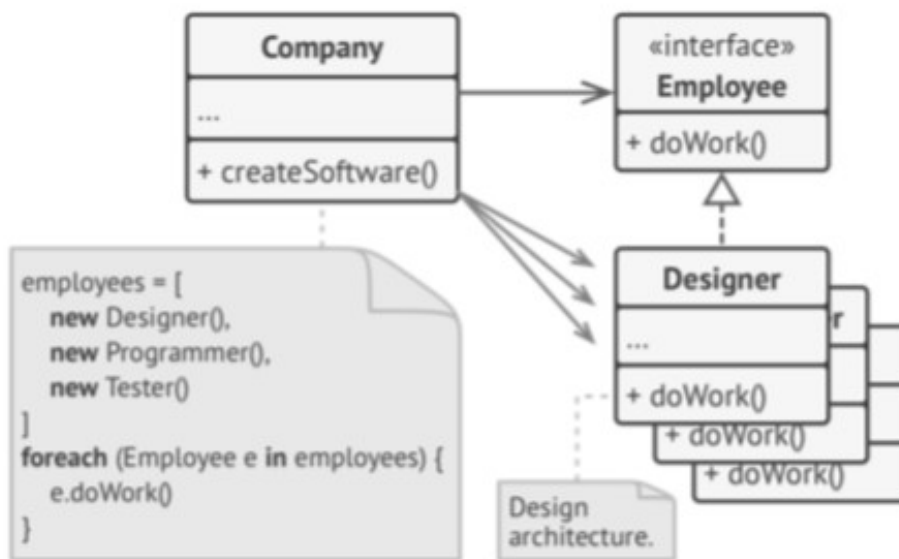
-By programming to an interface, the implementation can change or have as many variations as desired, and as long as the interface stays the same, the classes using it don't really care.

-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