“The Object-Oriented Thought Process,” 5th

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

**Reading this book was to serve as a refresher to the core concepts of OOP and to get back into an OO mindset after a long time in functional and procedural programming. The notes can provide the same, shortened.

Intro to OO Concepts

-Book more focuses on teaching basic OOP concept than code. How to think OO.

-Object wrappers – OO code that wraps other code (ex. procedural), to make it work like an object. Useful for wrapping functionality from legacy code.

-Basis of OOP – People think in terms of objects, so why not base architecture around them too?

-Object – an entity that has both attributes and behaviors
-Attributes – the data of the object. Should all be *private/protected*, accessed via abstractions.
-Methods – the behavior of the object. Functions inside a class. Access/use attributes.

-In procedural, data is often global, and separated from functions. Functions still access this data, though, and hence access to data is uncontrolled and unpredictable. With objects, data and behavior are grouped together and encapsulated from other objects.
-In proper OO design, there is no such thing as global data

-Objects allow data hiding of details from other objects, resulting in strong control over data and how logic is related among entities. By hiding implementation, use is simplified to a limited interface.

-Encapsulation – the segregation of the behaviors and attributes of an object from other objects. Allows hiding of implementation and only access through interface and select methods/attributes.

-Interfaces – the communication methods between objects. *public* methods. Implementation is *private* or *protected* and accessed through *public* interfaces. Through this, implementation can change without affecting objects that use interface.

-Objects should not manipulate the attributes of other objects. Many small objects that perform specific tasks > huge objects that take on too many responsibilities.

-Getter and setter methods allow controlled access to data and support info hiding. Getter method, aka accessor method. Setter method, aka mutator method.

-Instantiation – creation of an instance of an object. Each instance of an object contains its

own state for attributes and reference to methods (if non-static).

-Class – blueprint for an object that defines attributes, behaviors, and messages of objects built from it. Classes instantiated into objects, which are instances of classes.

-Message – behavior implemented in an object and called by another object (ex. *public* method). “Object A calls Object B which sends it a message.” Opposite to *private* behaviors.

-Primary visual tool in OOD are UML diagrams, with class diagram being most common type. In class diagrams + means *public* and – means *private*.

-Instantiation method uses to create object is considered part of the interface

-Inheritance – allows a class to share the attributes and methods of another class. Child classes can be created as abstractions of parent classes that share like behavior/attr. Inheriting classes can have their own behaviors/attr separate from parent classes. Common for data to be inherited but behavior not to be. Supports DRY.

-Superclass – class from which another class inherits from. Parent class.
-Subclass – extends (inherits from) superclass. Child class. Derived class.
-Superclass can be subclass also and vice versa.

-Abstraction – class design as a mixture of hidden implementation and public interfaces so as only to provide an abstraction of a class, not a full definition, on a, “need to know,” basis. Also seen through inheritance, where child is abstraction of parent class, and composition, where class contains instances of other classes.

-As inheritance tree grows, so does complexity of abstraction, hence leaving many devs wary of using inheritance.

-Single inheritance – classes inheriting from one class only
-Multiple inheritance – classes inheriting from multi classes. Not allowed in some langs (ex. Java). Can result in nameclashes.

-Is-a relationships – child class is-a parent class. Dog is-a Mammal. Dog *extends* Mammal.
-Allows child classes to be referenced as parent classes. Ex. Circle and Square both extend Shape. An array of shapes can also hold Circles and Squares.

-Polymorphism - “many shapes” in Greek. The ability of a class to take on multiple forms through inheriting subclasses, where different forms can have different implementation, but share the same common interface. Abstract methods, etc.

-Method signature – name + param list of method (and in some langs, return type)

-Overriding – when a child class redefines the implementation for an existing method in a parent class with the same signature

-Abstract method – method given definition (name, return type, scope, etc.) but no implementation. Implemented by child classes or classes implementing interface, etc. All child

classes inheriting from class with *abstract* method must implement details for. Allows forced uniformity and shared interface amongst child classes.

-Abstract methods allow flexibility as can create subtype objects as instances of parent classes, then call same method on all, where each subtype has own implementation, but all share same interface.

-Constructor – method called to build object from class. Holds initialization details for attributes, start-up tasks, etc..

-Destructor - Called when object no longer needed, to free memory, as well as perform any needed tasks before deletion. Often time garbage collectors destruct by default when scope ends, etc.

-Composition – when an object contains instances of other objects. *has-a* relationship. Car *has-a* Engine.

Thinking in Objects

-Analysis of business problems and possible solutions should come before choosing a language framework. The requirements determine the tools, not vice versa. Do conceptual analysis and design first.

-Separation of interface and implementation is essential and all not “need to be known” should be hidden. The driver only needs to know how to use what is in the cab, not under the hood. The interface should be minimal.

-Interchangeable objects must have the same interface (else not interchangeable)

-Implementation should be able to be changed without requiring change in interface definition (method name, return type, etc.)

-API is one example of an interface

-Middleware – connect high and low level. Software layer between apps and OS, API allowing access to DB data, etc. Can be used to connect new and legacy code by acting as in-between for data model/type conversion, etc.

-Object persistence – the concept of saving the state of an object to be used at a later time, even when falls out of scope, by saving state in DB, etc.

-Standalone application – can exist entirely from code defined in app, without needing to pull data from DB, etc.

-By creating classes to be abstract classes, with abstract methods and public interface, code becomes more re-usable. Ex. Going to airport requires various turn methods, but user doesn’t care about turning, they just want to go to airport. Thus, *goToAiport()* public method, thought

through abstraction, handles calling required concrete *turn()* methods.

-Vital to design classes from user perspective, not dev perspective, as far as interface goes. Design requirements should come from users, not just devs.

-Make sure when defining users you consider possible sub-divisions (ex. truck driver, taxi driver, etc.), as each might require special behavior, etc.. UML use cases good for documenting behavior requirements.

-Defining interface first gives a good idea of what logic will be needed and thus how to build implementation

-*public* methods will often call one or multiple *private/protected* methods (interface calls implementation)

More OO Concepts

Constructors
-One constructor built, typically call with *new* keyword, which creates instance of class. Constructor call allocates memory for object, so constructor should handle/call all initialization tasks

-Even if no constructor called, langs provide default constructors, but as general rule, always provide constructor, for easy documentation and maintenance purposes.

-If inheriting from class, in subclass, often call *super()* as first part of subclass constructor logic, which calls parent constructor, thus performing same initialization for subclass as superclass

-Overloading – allows you to have methods with same method name, but different param lists, in same class. Useful for creating multiple constructors.

-Good design to initial all attributes of class to stable state inside construction, as opposed to leaving some variable un-initialized, even if initialized value is just *NULL*

Error handling
-As far as handling errors goes, on error occurrence:
throwing exception > checking for problem and fixing if occurs > aborting app > ignoring

-Exception, thrown when unexpected event occurs in programming. Languages have some core exceptions, thrown by default, and devs can define their own.
try { something that might not work } catch (exception) { handle e } finally { do regardless }

-Not properly catching exceptions will typically cause prog to exit/crash

Scope
-Each object created, has its own memory, identity, and state

-Scope for attributes can be three types: local (inside function, loop block, etc.), object (declared in the root of the class, similar to a global class var), class (*static*...state is persistence across all instances of class).

-Each instance of object and local attributes get their own space in memory, while *static* attributes share same memory across all objects

-object attributes often must referenced from within the class via *this*.*attrName*, where *this* specifies which scope to use (the scope *this* is declared in)

Copying Objects
-How languages copy objects varies widely. Do they only copy by value or do they create a full copy of the object? If they do a full copy, does it also copy objects included through composition (deep copy)?

-Take special care of noting copy details for lang, as a result

Class Anatomy
-Basic anatomy: name, comments, attributes, constructor(s), accessors (getter/setter) methods, public interface methods, private implementation methods,

-Include comment a top of class describing responsibility

Random
-Constructor injection – instead of composing classes inside class via *new*, inject other classes into class during construction, passing them into constructor. This is a type of dependency injection.

-Memory leak – when an object no longer used is not destructed, and memory remains in use during program execution. Can lead to no sys mem available.

More General Design Considerations

-Documentation shows not just the purpose, but the process during development

-Since objects are used by other objects, objects should be defined together, with relationships and uses, and hence needed interfaces, etc. well defined

-Design classes to be extensible. Abstraction allows this.

-Static methods promote strong coupling of classes, as they cannot be abstracted, mocked, etc.. Be very careful when deciding to use.

-Abstract out system specific code that will vary from system to system, allowing remaining code to be used on any system. A system specific wrapper class can provide a unique interface to cross-systems implementation code.

-As copying and comparing objects varies heavily by lang (deep, shallow, etc. only), provide your own methods for copying and comparing objects

-Keep scope for attributes as small as possible

-Maintainability comes from small, loosely coupled classes. The less interdependent classes are to each other, the better. If a change in one class requires a change in another class, they are highly coupled. By having proper interfaces, coupling is next to eliminated.

-Iterative design is the process of writing code in small increments, testing each step. Often time each iteration involves the same steps each iteration, such as: ideation, prototyping, building, analyzing.

-Interfaces can be built as *stubs,* where the interface meets all the required signatures, etc., but the implementation is very simple, not the end desired behavior, etc.. Allows you to test interface without having implementation done.

-If an object requires persistence (persists past execution of program), three options:
  a) serialize to XML or JSON then write to DB or disk – very dated
  b) middleware to convert object to relational model, then DB save
  c) NoSQL DB (ex. MongoDB). Very efficient.

-Serializing – deconstructing (flattening) object, sending it over wire (*marshaling)*, then reconstructing back into object. Data often separated from behavior, to avoid needing to do this.