“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 Class Design Guidelines

-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.

Designing with Objects

-The design process is as much or more important than the design itself.
-Ex.
1. Gathering of business requirements and tech limitations
2. Developing a statement of work that describes the system
3. Gathering the requirements from this statement of work
4. Developing a prototype for the user interface
5. Identifying the classes
6. Determining the responsibilities of each class
7. Determining how the various classes interact with each other
8. Creating a high-level model that describes the system to be built

-High-level module step #8 above would include mix of class diagrams, class interactions, etc. as shown through UML

-Waterfall design method – distinct steps of a design <--> implementation loop (each in own section), then finally deployment. Often leads to not what user wanted and dated, though.

-Other design processes (from most to least most used in 2020) include: Agile, Scrum, rapid prototyping, and extreme prototyping. Iterative design key in most.

-Understanding all business requirements early on is key to successful design and reducing expensive changes later on. It is important to work directly with the users for this reason.

-Poor design early on leads the later problems. Then, in the future, when the code becomes un-manageable and the horrible design is revealed or causes problems, the reputation of the one who made, and the company providing it, it is the one hurt most.

-Statement of Work – using business requirements, details a statement of what uses the system will have, what services it will provide, etc. Written in narrative form, with bullets.

-Once business requirements, and statement of work done, then move on to developing more technical requirements, as far as what user will need to have done, etc.

-With requirements ironed out, a user interface prototype can be done. This can be simply a drawing, app created interface, or fully coded demo.

-Important to determine class responsibilities properly during class design to be single responsibility, etc. Determining class relationships helps with this, as does UML modeling.

-If existing structured code must be used, this can be broken down into methods inside classes (wrappers).

-If have old, vendor, etc. classes, sometimes useful to wrap in new, where wrapping methods can both provide new interface, or modify implementation, via man-in-the-middle style processing.
-Middleware for writing objects to persistent DB state also considered class wrapper.

-Wrappers very commonly used, and provide a way similar to inheritance to create relations between existing classes

Mastering Inheritance & Composition

-Inheritance often debated whether to use or not, including abstract classes, which are a form of inheritance, with many devs now preferring composition only

Inheritance
-Liskov Substitution Principle – Objects of subclasses must behave same way as objects of superclass. To fulfill this, overriden methods needs to have the same signatures and return type. All behaviors and attributes of superclass must also be appropriate for subclasses. Lastly, must properly test to ensure proper LSP.

-LSP is important, as if a subclass differs in this, then it is not properly fulfilling the *is-a* relationship for inheritance. Imagine a B*ird* superclass is created and has a *fly* method, then a *Penguin* class is created, inheriting from B*ird.* A penguin cannot fly, and thus is in violation of LSP. To fix this, we could create a *FlyingBird* and *NonFlying* bird classes in between *Bird* and *Penguin*.

-Classes should be design in a most general to most specialized hierarchy, with shared

behaviors in most general class as possible

-With each breakdown into a more specialized subclass, complexity increases. As a result, sometimes it is better to make models less 100% true to real world models, and more true to the models used by the system, even if not real world accurate, to simplify and make model more manageable.

Composition
-Note that author considers aggregation and association both as parts of composition

-Like inheritance, be careful in design of not making composition overly complex, by having many nested compositions (A contains B, which contains CD, which contains EFG, etc.)

Arguments Against Inheritance
-Inheritance weakens encapsulation within inheriting classes by having classes which share attributes and behavior, cross-class

-Changes in superclass are not apparent to subclasses, so change in superclass can break subclasses, or require changes in many subclasses to account for

-Inheritance should exclusively be used in *is-a* relationships, as to not force relationships, or create code unnecessary relationships exist, where classes would be fine existing without relationships

Polymorphism in Detail
-The main benefit of polymorphism often comes in when building an interface (ex. abstract class) that defines general function signatures, when then are implemented (via *extends*) by a variety of implementing classes.

-As a result of this forced implementation by extending classes, a method which takes in *x* type object can take in any *y* type object, as long as *y extends x*. This allows for wide flexibility in not just design of classes, but functions, etc. that take in objects as args, call methods on objects, etc.

Frameworks & Reuse

Frameworks
-A framework often provides a set of standard functionality to allow you create objects, access objects, update objects, etc., through ready set methods to do so, creating required relationships, etc., based on design patterns used by the framework in these actions (ex. MVC)

Contracts
-Any mechanism that requires a developer to comply with specifications of an API, as implemented by the framework.

-Often done through abstract classes and interfaces, both which force adherence on extension or implementation of

-Contracts are useful as in between points for reducing coupling between classes, by creating coupling to contracts, after which the extending/implementing classes will be dependent on the contract, and then high-level code is also dependent on the contract, instead of the low-level classes bound to that contract. ie dependency inversion.

When to Use Abstract Classes, Interfaces, and Inheritance
-Abstract classes cannot be instantiated, but any class extending must define implementation for declared abstract methods in abstract class. Very polymorphic.
-If abstract class also implements methods, contract would be in abstract methods also in that class

-PHP defines syntax for declaring abstract classes, as well as interfaces

-Abstract classes can have hierarchies. An abstract class can extend an abstract class, which a normal class then extends. Interfaces, however, exist stand alone, as an interface cannot implement another interface. A class can also implement multi interfaces, but only extend one class.
-Interfaces thus provide more flexibility and less potential for undesired class hierarchy coupling, while abstract classes allow for partial implementation through normal, concrete methods

-Note that you can both extend an abstract class and implement an interface. This allows segregation between the two and further specialization and generalization, as well as fully abstract interfaces (ex. more generalized) with a partially concrete interface (ex. more specialized)

-Interfaces, even though a special type of inheritance, are separate from standard inheritance as implementing an interface does not require a strict *is-a* relationship, and is more like a *behaves-like-a* relationship. Interfaces are thus less relationship coupled than inheritance.

-Common interface use is to provide a standard method to access similar methods (ex. *getID*) across multiple classes that can not be related by inheritance, abstract class extension, etc. (ex. *Planet, Dog*), which then creates a single standardized inheritance for users to call, even across classes without much relation, but similar behavior

Code Reuse
-Code written just to work is generally very closely coupled to the project it was written for, and thus hard to re-use quickly and effectively

-Code reuse can be built after existing, separate modules exist, based on common behavior, etc., through interfaces, etc., which then become basis for future code design, and thus end up building a framework through a series of contracts

-One benefit of above type framework based design is that bugs also largely become standardized, and you don’t get many forks, each with bugs occurring at random times, being fixed in some cases, not in others, etc. A framework creates a single codepath, and thus largely avoids this.

-Framework design can also be done at the start of development, by considering future applications, as opposed to just current requirements/parties, and building a framework that would be general enough to accommodate them, even with uncertainties due to hypothesizing

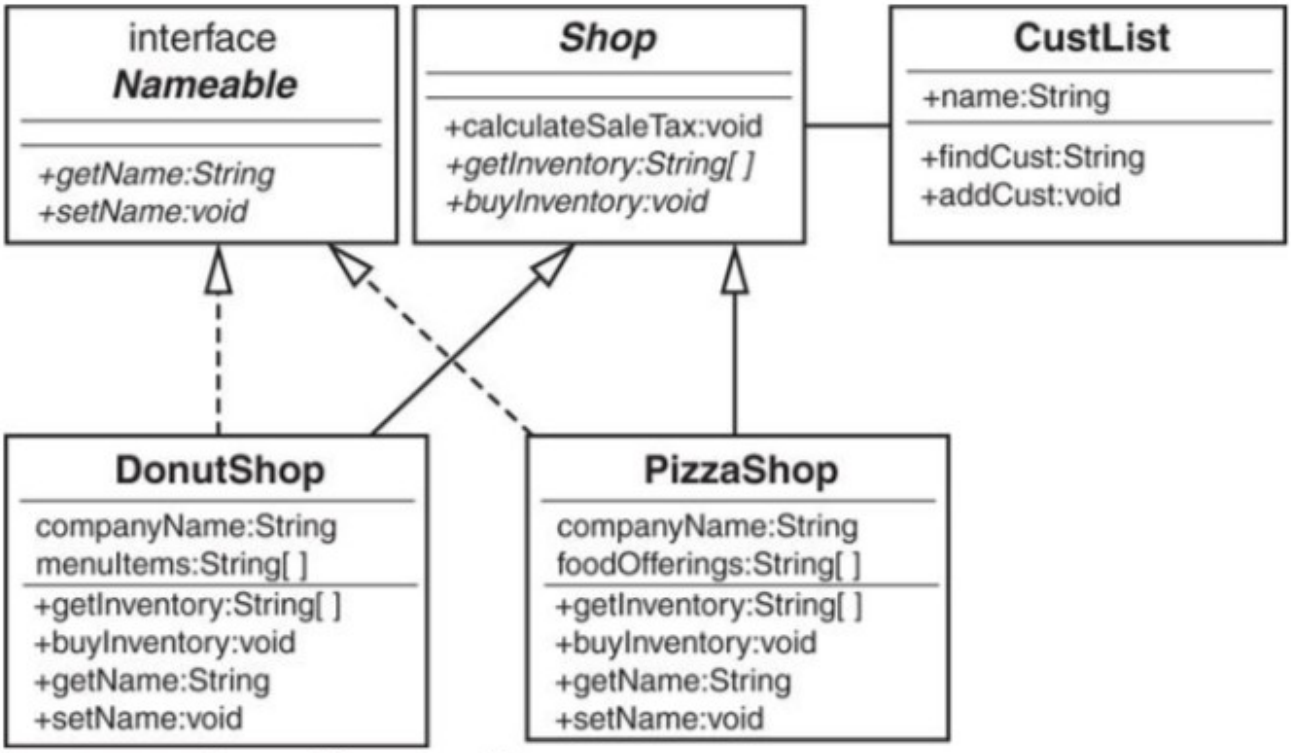
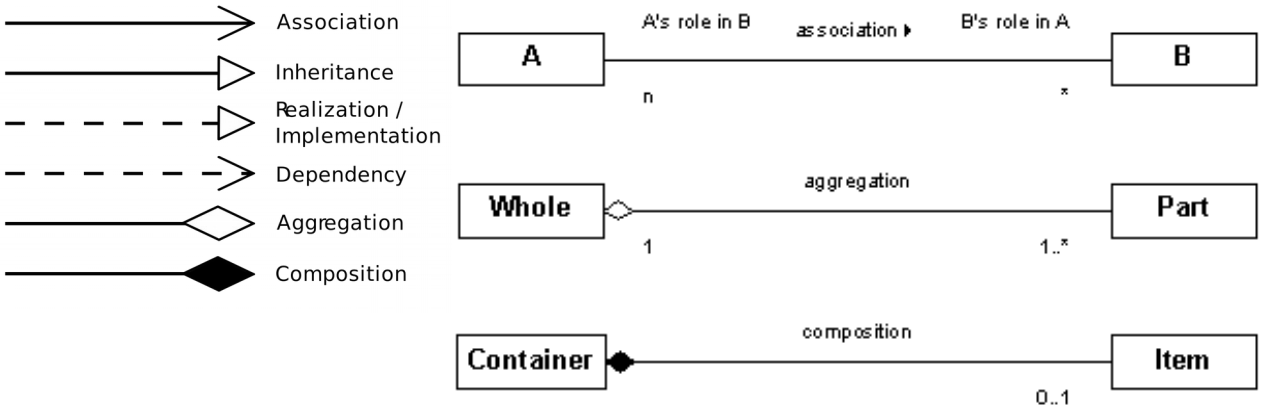


Figure 8.7 A UML diagram of the Shop model.

-Above is example of contract using a mix of contract types. The *Nameable* interface provides contracts for names, which all shops, as well as other object subclasses, could have. The abstract class *Shop* provides a basis for specialized shop classes, providing a contract for inventory methods, as well as a concrete *calculateSalesTax* method, which all shops will inherit. Lastly, *Shop* contains a *CustList* instance, which can be used by *Shop* and all subclasses for customer interaction. Note also that *Shop* subclasses also contain their own specialized behavior and attributes.

-Not that in the above example, the only coupling is to *Shop*. Utilization code can then be designed largely to work with any *Shop* or subclasses, where objects can be referenced as *Shop* instances, even if they are subclass instances, due to contracts

Composition, Association, and Aggregation



-One could say that inheritance is largely designing one class, through main class and sub-classes, through dependencies of children on parents, while composition is creating truly separate classes, where dependency still exists through inclusion, but not explicitly simply through definition

-Composition builds a system by combining less complex parts, while inheritance results in building more complex components

-Composition also builds a large collection of unrelated parts that can be combined in a variety of ways, as needed, which is more flexible that inheritance, which while allows further specialization and generalized references of a main and many subclasses, also creates a set relationship between these parent/child classes.

https://www.geeksforgeeks.org/association-composition-aggregation-java/

-Association – Relationship where object uses another object. Composition and aggregation are types of associations.

-Composition – *has-a* relationship. Objects held are created by the class using then, and thus also die when composing class dies. Thus, closer to *part*-of relationship. Classes highly coupled.

-Aggregation – *has-a* relationship, but objects used are passed into aggregating object, thus less coupling. Since object passed in, can also existing outside of aggregating object.

-Composition best used when composed objects serve no purpose outside of composition, where aggregation used when can use aggregated objects outside of also.

-Again, by adding an interface in between the using object, and object being used, you can de-couple and reduce dependency of use by *x* dependent on implementation of used *y*

Cardinality
-Describes number of potential objects in relationship, including if mandatory or not

-One – 1
-Many - n

-1..n – 1 to many
-0..1 – 0 to one
-0..n – 0 to many

-ex. college 1...n-----1 dorm //college has one-to-many dorms, dorm has one college

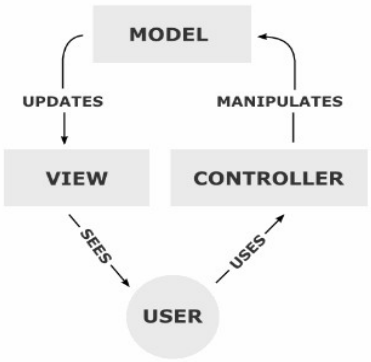
Optional Associations
-Be careful to check all optional associations for *null* (! exists) before trying to use

Design Patterns

-Design patterns are very closely tied to best practices, as the patterns themselves establish contracts for best practices

-Gang of four describes pattern with four essential elements: the pattern name which gives reference point. The problem. The solution (the pattern). The consequences (the results and trade-offs).

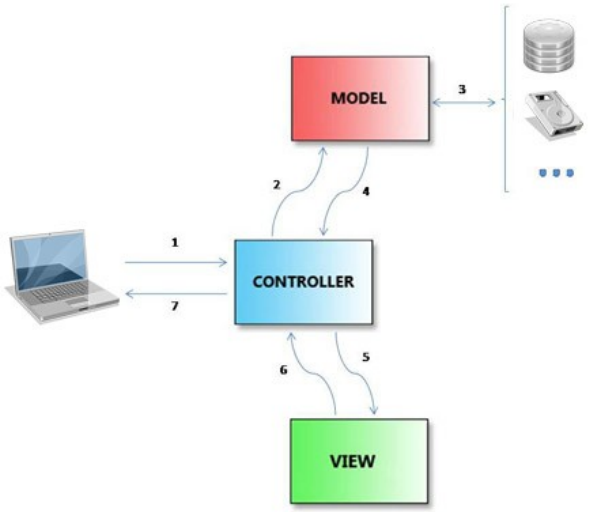
MVC
https://www.sitepoint.com/the-mvc-pattern-and-php-1/
https://code.tutsplus.com/tutorials/mvc-for-noobs--net-10488



-Model – the data and access to it. The model is "blind." It has no connection or knowledge to what is being passed to the view or controller. It does not seek a response, just takes requests. The model is more than the DB, but the gateway to it as well.

-View – the data displayed to the user, formatted for output. The HTML/CSS/JS/etc.. Can read interactions from the user which it sends to controller. The view *does not* receive data from the controller. Often also does things like data validation before sending to controller or checking with view that user exists, etc.

-Controller – Handles data user input and other user interactions (ex. clicks) and update model accordingly. Controller is one way data flow.



-View shows display. Controller detects interaction (ex. form submit, *GET, POST*, etc.), the calls proper model code. Model handles access/update/etc. of data, then sends updated data to controller which selects proper view.

-Pattern that separates responsibilities of back-end and front end into defined parts and allows back-end logic, UI, and data changes, all without affecting each other

-Example: *book_controller.php* receives user GET request for page, looks at request, then passes off to *book_model.php* for list of requested books. *book_model.php* gets requested info from DB and formats

book titles. Controller then selects appropriate view and presents view to user.

-Many frameworks use MVC

Creational Patterns
-Handle creation of objects

-Abstract factory, builder, factory method, prototype, singleton

-Factory method – Start an interface/abstract class implemented by multiple more specialized subclasses. Factory is then a class which has a creational method to read in desired subclass/implementation, and return instance of requested type. As a result of *is-a* relationship of children, return type of method can be parent, while return logic is child. Note that individual subclass constructors still handle subclass instantiation logic. Factory just returns requested type.

-Factory ex.

```
public class ShapeFactory {
   public Shape getShape(String shapeType){
      switch(shapeType) {
         case "circle":
            return new Circle();
         case "square":
            return new Square();
         case NULL:
            //throw exception
            return NULL;
      }
 }
```

-Factory pattern separates creation interface from implementation method(s) and constructors and allows single method for creation of all subclasses. Objects are then generated from the factory, which provides a uniform creation interface, where instantiation logic can change as long as interface methods stay same. Also creates code standardization, with objects always being created using same factory method, etc.

Structural Patterns
-Create larger structures from groups of objects

-Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy

-Adapter – Allows to create different interface from what already exists. Wraps existing code and provides a new interface. Typically wrapper creates instance of original class, then calls existing methods in class, but through new interface methods. Mainly used to provide a new interface for a class with desired functionality, but non-matching interface for needs.

Behavioral Patterns
-Concerned with interaction and responsibility of objects, in which interaction between objects is built in such a way to reduce coupling

-Chain of Responsibility, Command, Interpreter, Mediator, Memento, Observer, State, Strategy, Template, Visitor, Null Object

-Iterator – provide a standard method for traversing a collection. Hides the internal structure of the collection, while still allowing traversal through elements. Iterators require iteration across multiple iterators without the iteration of one affection the other.

Antipatterns
-Design patterns proactively solve a specific type of problem. Antipatterns react to a problem that has already occured and are learned from past bad experiences. They emerge from failure of past solutions.

-Downside of anti-patterns, is while reactive, are hard to standardize, and thus code lacks comprehensive design, and is more a random mix of one off solutions

Highly Decoupled Code

-Inheritance provides reusability, extensibility, and polymorphism, but it also couples classes by creating *is-a* dependencies. For this reason, many support *composition over inheritance*, using inheritance only if truly needed.

-Composition, which creates classes used inside the using class are also highly coupled. This is only scope/lifetime coupling, though.

-Inheritance can cause complications by being unsure where to delegate common behaviors to, as not all inheriting models might apply for those behaviors

-Dependency injection, unlike composition which creates objects inside an object via *new*, injects objects into a class via a *setter*, etc. method, which the object then assigns to private vars to be used to be used by class.

-Dependency injection often done by passing args into constructor, to set args contained by class during creation time. Also can provide setter, to switch out dependency instances.

-Dependency injection often preferred to composition via *new* as reduces dependencies and gives better flexibility

SOLID

https://stackify.com/solid-design-principles/

-Bad code comes largely from rigidity (changing one part of the system breaks another), fragility (when things break in unrelated places), and immobility (when code can only be used in the original context). SOLID attempts to solve this.

-**S**ingle Responsibility Principle, **O**pen/Close Principle, **L**iskov Substitution Principle, **I**nterface Segregation Principle, **D**ependency Inversion Principle

Single Responsibility Principle
-A class should only have a single reason to change (from if the responsibility changes).

-Ex. a *CalculateArea* class calculates area for a shape via an *area()* method displays via a *displayArea()* method. This method is now calculating and displaying. This fails SRP. What happens if calculation requirements change? Changes to *CalculateArea* could now also affect output methods.

-SRP unties coupling. If a requirement changes, only class responsible for method requires changes.

-Tie a class to only one actor's requirements, thus you only have one actor who could require changes, and avoid conflicting change/addition/etc. requests from multiple actors

-If describing responsibility of class requires use of “and,” likely is not SRP valid

Open/Closed Principle
-You should be able to extend a classes behavior without modifying it. ie, you should be able to add new functionality without changing existing code.

-This can be enabled through inheritance, including abstract class extension, so that the original class stays unchanged, and inheriting classes extend and provide required changes.

-Traditionally, OCP was done through standard inheritance of *child* class from *parent* class. Since this produces tight coupling, though, it is now typically done with interfaces, where the interface is closed for modification, but the implementing classes are open to implement however they please.

Liskov Substitution Principle
-Any instance of a parent class must be able to be replaced with an instance of its child class

-Done through contracts, where overridden methods must take in same args and have same return type.

-Validation rules on input params should not be more specific in child class than parent class. At least all same rules for method returns in child classes as parent classes.

Interface Segregation Principle
-Clients should not be forced to depend upon interfaces that they do not use. Many small, more specific interfaces are better than few large ones.

-In order to achieve, interfaces must be designed to only fit a specific client or task. If you find the interface teetering towards becoming ambiguous in client or task responsibility, break down into multi interfaces.

-If needed, multiple interfaces can be implemented by a class (ex. *GrindingCoffeeMaker* might implement both *CoffeeMaker* and *CoffeeGrinder* interfaces)

Dependency Inversion Principle
-High-level modules, which provide complex logic and use lower level modules, should be easily reusable and unaffected by changes in low-level modules, which provide the utility features. This is done by creating an abstraction between the high and low. The abstraction should not depend on details. Details should depend on abstraction.

-High level objects should not depend on low level implementations. If a high level object calls a low level method, but then the method return type, etc. changes, now the high level object’s functionality is broken and requires changes to account for low-level changes.

-Main function – root function which calls functions, but never is called by functions
-High-level functions – functions called by main function, which call other functions
-Mid-level functions – functions called by high-level functions
-Low-level functions – functions called by mid-level functions, which call no other functions

-In proper dependency inversion, both front and back-end, etc., will revolve around business rules, which define interface. Abstractions should not depend on details. Details should depend on abstractions.

-DIP in practice: An interfaces is created, which low-level then implements. Low-level can then change implementation without affecting high-level. Both are now dependent on interface (abstraction).

-Abstraction, high level, and low level modules should all be in own packages so packages reflect same dependency relationships as classes (both dependent on interface package)

-DIP reduces impacts of changes, because low-level implementation can change as long as interface stays the same

Dependency Injection
-Classes should be sent into class instead of composed into class via *new*. This can be done either by injecting via constructor params (*constructor injection)* or another initialization method, like a setter method (*parameter injection)*

-Decouples classes construction from construction of classes

-Dependency injection fulfills dependency inversion principle by decoupling creation of lower level modules from the higher level modules that use them.

-Dependency injection also lets you create methods that take in a more generalized parent class, then pass in child class instances, take in any object implementing an interface, allowing different implementations to be passed in, etc.