“Clean Architecture”

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

I - Intro

-Software architecture - the fundamental structures of a software system and the discipline of creating such structures and systems, where each structure comprises software elements, relations among them, and properties of both elements and relations.

-Structure = components, classes, functions, modules, layers, services, etc. Physical constraints (processor speed, RAM limitations, bandwidth, etc.) also considered.

-Proper architecture design during code base creation (or lack of) makes or breaks the cost of change, scalibility, readability, etc. for future use.

-Architecture is like the foundational logic of code. Though syntax, etc. might change, the logic persists. Architecture does the same with how to organize and connection the structure of the code and system that runs it.

-Anyone can write code and have it work. To develop a beautiful, sustainable, update-able, organized structure is not as easy.

-The opposite of clean architecture is a highly coupled, heavily interconnected, poorly organized architecture. The result is high difficulty to modify, maintain, scale, etc..

-Often new hires are brought in and told to just use existing components, modules, classes, libraries, frameworks, etc., without actually learning the design themselves. Over time, as seniors retire, these people become leads. If they never took the time or had the time to learn proper architecture and software design to create proper hierarchies, dependencies, etc., you end up with team leads who don’t use rudimentary design patterns, and the system falls to dependency web nightmare, zero scalability/adaptability, etc.

II – Design vs Architecture

-Author supposes there is little difference between the two and the terms, and if anything, architecture is just low level design.

-As a company grows, engineer staff also grows, but with poor architecture, productivity will stagnate. Then, payroll has increased, while productivity has stagnated, which results in a higher cost of business, without adjoining productivity. This is what the executives will look at and cause problems for the engineers.

-If the goal is to get code out as fast as possible, and then the code becomes a mess as a result, making it incredibly hard to clean, then cleaning will get put on hold for producing more code ASAP. Eventually, the cleaning never occurs, or it becomes so messy, the code must just be fully re-done.

-Software architecture is defined to be the rules, heuristics, and patterns governing:
  -Partitioning the problem (and hence the system) to be built into discrete pieces
  -Techniques used to create interfaces between these pieces
  -Techniques used to manage overall structure and flow
  -Techniques used to interface the system to its environment

-Appropriate use of development and delivery approaches, techniques and tools.

-Proper architecture controls complexity, enforces best practices, gives consistency and uniformity, increases predictability, and enables re-use.

III – A Tale of Two Values

-Behavior – code that satisfies the requirements for the program

-Architecture – If the behavior is the "where" in software the architecture is the "soft" part, which makes the behavior is easy to change.

-The two often battle in a case of function vs design. Should it just work, or should it work and be easy to change? If designed to just work, the system becomes resistant to change and hard to scale.

-Architecture can be urgent and important or not urgent and important, but never unimportant. Convincing managers and business side of this is hard, but vital to long term sustainability, especially during early stage development of a system.

IV – Programming Paradigms

-Programs are simply data. The for loops, assignments, subroutines, etc. are all just methods to interact with that data.

-Paradigm – a way to program, including what structures to use, how to use them, how state is managed, etc..

I) Structured Programming – imposes discipline direct transfer of control
II) OOP – imposes discipline on indirect transfer of control
III) Functional – imposes discipline on variable assignment

-Paradigms are largely based on rules that what not to do, rather than what to do (ex. *side effects* in functional programming). Paradigms add restrictions, not capabilities, which in turn promote structure and organization.

V – Structured Programming

-Established by mathematician Dijkstra in the late 50's as a way to bring rules and order for how to program

-Designed with the basis of code needing to satisfying proofs to be valid, done through established structures with an expected result acting as proof (algorithms).

-All programs can be composed from three structures: sequence, selection, and iteration (repetition), all of which can be declared valid with proper proof. The use of just these structures is structured programming.

-The result of this, was the large vanishment of the *goto* statement (which performed a one-way transfer of control from one line of code to another)

-In structured programming, modules can be recursively decomposed into smaller proveable units,

meaning a large function, etc. can be broken down into smaller ones, where each decomposition has its own restrictions. This is called ***functional decomposition***.

-Developing formal proofs for individual applications never caught on, though, and instead, implementation of the basic three structures, as long as adhering to those structures, are generally just seen as proof themselves

-Testing can show the presence of bugs, but not the absence of. Programming thus is an endeavor of failing to prove incorrectness. In structured, this comes into play by failing to prove the smaller modules as broken as part of the whole.

-In addition to the three basic control structures (sequence, selection, repetition/iteration), structured programming also includes:
I) subroutines – callable units such as functions, procedures, methods, etc.
II) blocks – groups of statements to be treated as a single statement

Procedural Programming

-Structured often used when doing procedural programming, where the focus is to break down tasks into collections of variables, data structures, and subroutines

-OOP vs procedural – while OOP is centered on breaking down components more into blocks, OOP breaks down into objects, where only logic is split into classes, and then accessed via interfaces

-Functional vs procedural – while procedural is based on a sequence of imperative commands that may share state, functional has a free order of code execution through "same input, same output" functions that avoid side effects resulting from execution. Functional is often declarative, while procedural requires imperative logic.

-Many procedural langs also include functional elements, such as first-class functions, high-order functions, etc.

VI – OOP

-Data modeling that reflects real world objects and relates them through encapsulation (segregated scope and details), inheritance (the ability to implement parent methods/data), and polymorphism (a class can also be multiple other classes through implementation)

Encapsulation
-Details of class can be hidden from other classes. Only the interface, not the underlying logic, must be known for use.

-public, private, protected

Inheritance
-The redeclaration of a variables and functions within an enclosing scope
-*SecurityException* inherits from *Exception*

-x is-a X

Polymorphism
-Essentially the application of pointers to functions, seen in OOP without having to manage these pointers individually, through inheritance

-The power of polymorphism lies in the ability to write code as plug-ins of sorts, where if root

necessity changes, where all classes using that root also require changes, only the root class which others inherit/extend from, needs to be changed (as long as the interface stays the same).

Dependency Inversion
-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

-Above forms a tree:

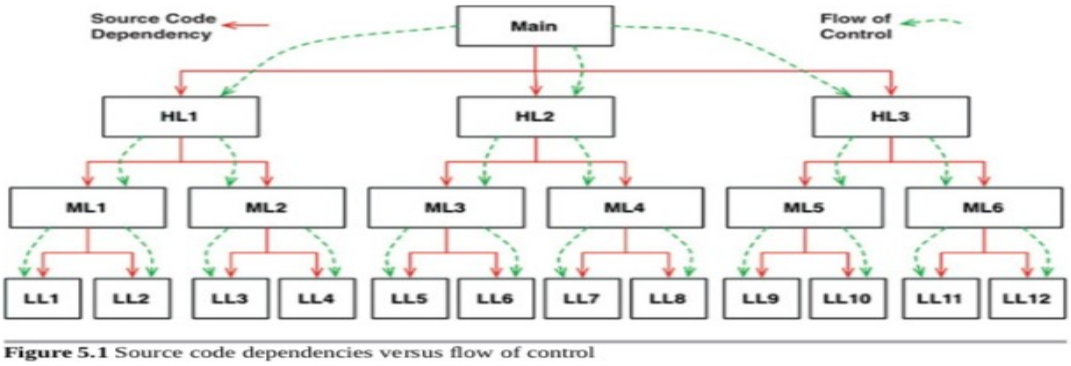


Figure 5.1 Source code dependencies versus flow of control

-In above, in order for higher level function to call lower level function, higher must *include* etc. to reference, creating an equivalent dependency and control flow tree, where low level functions are fully dependent on higher level functions.

-Dependency inversion negated this equivalent dependency/control flow by creating an abstraction between high and low level classes, the abstraction acts as an unchanging interface to separate the logic of both, thus de-coupling and providing *independent developability*.

-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. Low level classes define execution logic and extend and interface which contains methods to call those

Summary
-OOP allows, through polymorphism, to create a plugin based architecture, where low level logic is encapsulated from interface, and thus can be changed without breaking implementing modules or causing dependencies issues.

-Through dependency inversion, high level classes can be separated from low level classes and coded as plugin modules, removing dependencies, by having both depend on an abstraction.

VII – Functional Programming

-Largely based on l-calculus invented by Alonzo Church in the 1930s

-In truly functional languages, variables are always immutable
-Immutable variables eliminate race conditions, deadlocks, etc., thus eliminating one of the main problems on concurrent programming

-Immutability also comes with performance restraints from lacking concurrency, and thus compromises

must be made. Often thus segregate stateless pure functional components from non-pure stateful functions, and place the mutable code in a file with transactional memory (similar to how DB commits, etc.), preferring immutable over mutable components.

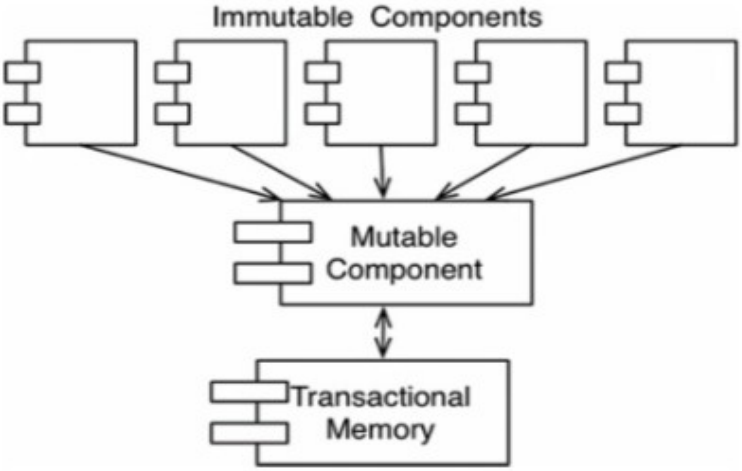


Figure 6.1 Mutating state and transactional memory

-ex. in Clojure you can do this w/ *atom* type variables that can be mutate by the *swap!* Function

Event Sourcing
-Where store all transaction instead of state. When state is required, apply all transactions from start to get current state.

-Often done with state being computed from period transactions and stored on a regular (ex. nightly) schedule, then using those states for the final state computation, while still storing transactions too

-Benefit is all encompassing history of state, where no state is ever stored, modified, lost, etc.

Summary
-By making variables functions, you get same same input, same output data, without needing to store and manage state. Such immutability alleviates problems of concurrent processing and simplifies state.

-Paradigms are largely based on rules that what not to do, rather than what to do (ex. *side effects* in functional programming). Paradigms add restrictions, not capabilities, which in turn promote structure and organization.

-Software is simply sequence, selection, iteration, and direction, nothing more, nothing less. The rest is all design and architecture.

VIII – OOP Review

https://www.codeproject.com/Articles/22769/Introduction-to-Object-Oriented-Programming-Concep

-Objects largely based on models (ex. person, car)
-Class is the root representation of the object, from which individual *instances* of the object are created
-A class has a name, as well as associated methods and attributes
-Manipulation of an non-static instance will not affect other instances

-A well defined class will exist as a meaningful single responsibility representation, and support reusability, which increasing maintainability and expandability of a system

-Divide and conquer

Interface
-Similar to a template, it defines the required methods names, types, args, etc., without providing implementation details. It is a template for what you need to fulfill, not how to fulfill it.

-An interface cannot be instantiated, but is *implemented* by other classes

-Any class that inherits from that interface must then implement those method definitions. Thus, all classes implementing interface X will be able to be treated as X.

-It is a blueprint only in telling what must be done, not how it will be done, which is determined by classes that use the interface.

-A single class can implement many interfaces and an interface can inherit from other interfaces

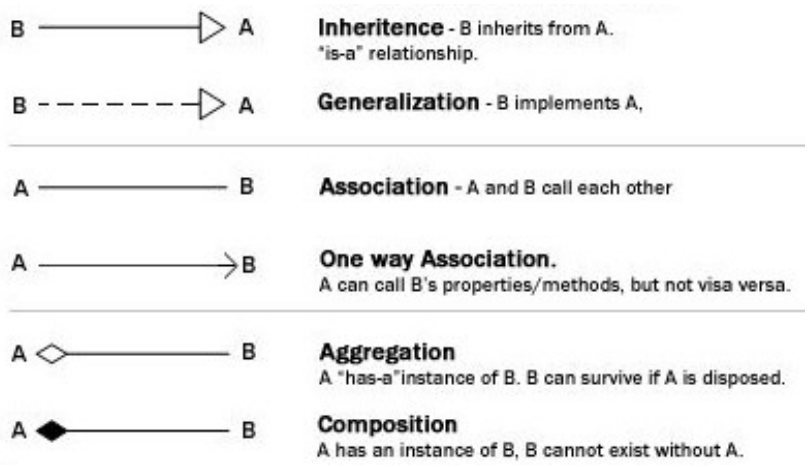
https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-aggregation-vs-composition/

-If class X implements interface Y, then you can create Y as *X myX = new Y()*

-If a class is an idea or representation of a real world object, an interface is an abstraction of such

Association
-A relationship between two classes
-X uses Y by creating and instance of Y inside it, then accessing it's methods

-Relationship: X *has-a* Y
-Person has a hand



Aggregation
-Weak association with partial ownership in which even if object Y is associated to X, it can still exist without X

-Relationship: X *uses* Y
-School uses teachers

Composition
-strong association with full ownership

-Relationship: X *owns* Y, where if X vanishes, so does Y
-school owns courses

Abstraction
-Places emphasis on what an object is or does, not how it works (which is often hidden accessible only through an interface)

-Generalization takes a broad abstraction that can be implemented by multiple times of objects and sets a definition for that abstraction. Polymorphism is an especially important part of this.

Abstract Class
-Defined with *abstract* keyword placed before *class* in many languages

-Cannot be instantiated. Can only be used as a super class on which other classes extend

-Unlike an interface, which only provides method templates, to have implementation detailed by implementing class(es), an abstract class can have implementation for methods

-A class may only inherit from one abstract class (unlike interfaces, which can be multiple inherited)

-An abstract class provides not just a template for implementation, but a base for of implementation details as well. A framework for logic in inheriting classes.

-Abstract class can have an instance of it created, in which *public* entities will be available

-An abstract method can be without implementation if declared with the keyword *abstract*

Method Overloading
-Multiple methods of the same name can exist as long as they have different parameters, return types, etc.

Method Overriding
-An inheriting class can redefine the implementation for the method(s) of a parent class, in which the defined function has the same name and parameter list, but different logic

Use Case
-Something an actor perceives from the system
-Maps actors with functions

-Use cases largely define the system

Class Diagram
-Describe types of objects and their relationships
-Model class structure and contents, providing conceptual, specification, and implementation details.
-Used as primary visual design tool along with system overview diagrams and physical data models

-Part of UML diagram schema

Package Diagram
-Organization of packages and their elements, such as namespaces in which classes exist, which can form modules

Sequence Diagram
-Model flow of logic within a system. Useful for showing not just class relationships, but specific logic flow relationships between those closes.

Two-tier Architecture
-Basic client-server model. UI runs on client. DB stored on server. App logic can be client or server side.

Three-tier Architecture
-UI, application logic, and data separated
-Common architecture that took over two-tier in 90's

VIII – Design Principles

-Good software comes from clean code. Clean code results in well made bricks. Design principles tell give direction on how to make such bricks, but more importantly, how to relate, arrange, etc. them.

-SOLID – design principles no how to arrange functions and data structures into classes, and how those

classes should be interconnected.

-SOLID largely based on the creation of mid-level software structures that tolerate change, and easy to understand, and can be the basis of components used in many software systems.

-SOLID can apply to non OOP as well, applying to groupings of segregated data instead (ex. segregated procedural PHP scripts)

-Mid-level programming – interfaces, etc. built above the level of the code logic, and used to define the underlying logic of those lower modules and components

-Software entity - class, function, module, component, etc.

-**S**ingle Responsibility Principle – every software entity should only have a single responsibility, which should be entirely encapsulated by that class/function/module. An entity should only have one reason to change (the responsibility changes).

-**O**pen-Closed Principle – software entities should be open for extension, but closed to modification

-**L**iskov Substitution Principle – Objects should be replaceable with instances of their subtypes by adhering to a contract that allows them to be substituted for one another

-**I**nterface Segregation Principle – Many client-specific interfaces are better than one general-purpose interface. Avoid depending on things not used.

-Dependency Inversion Principle – Depend on abstractions, no concretions. High-level policy should not depend on code that implements low level details. Rather, details should depend on policy.

XI – Single Responsibility Principle

-A software entity should be responsible to one and only one actor, where an actor is the group or people seeking that responsibility. Separate code that different actors depend on.

-Ex. of violation:
module has *calculatePay()*, *reportHours()*, and *save()*. This violates SRP as HR is using *calculatePay()*, business side is using *reportHours()*, and CTO is using *save()*. These three actors are now coupled together and cross-dependencies can arise, where the change in the requested functionality from one actor influences the functionality of another action.

-Issue from responsibility overlap in the same module results when multiple responsibilities overlap in dependencies. Then, a dev goes into change part of that overlap, with the intent of changing one responsibility, but ends up (sometimes unknowingly) changes multiple due to coupling.

-SRP also prevents merge conflicts. What happens if a module has multiple responsibilities for multiple actors and multiple actors request changes at the same time? Multiple coders might start working separately on the module, resulting in broken dependencies across responsibilities. Not with SRP, as the separate actors get their own modules.

SRP in action
1) Separate the data from the functions. If multiple actors need to access the same data, they should be doing so from separate modules.

2) Facade Pattern
-Encapsulates a complex subsystem within a single interface object. Allows multiple responsibilities to

be accessed from a single class, without coupling them.

-Implementation: One class per responsibility. Facade class includes instances of those low-level

Figure 7.4 The Facade pattern

classes (set as private, created as *new* in constructor, etc.). Facade class then provides and interface to those methods, sometimes calling multiple low-level functions/classes from a single interface function, if needed. Client should only interact with facade, not low-level modules referenced in it.

-Ex.

```
class ComputerFacade {
  constructor() {
    this.processor = new CPU();
    this.ram = new Memory();
    this.hd = new HardDrive();
  }

  start() {
    this.processor.freeze();
    this.ram.load(this.BOOT_ADDRESS, this.hd.read(this.BOOT_SECTOR, this.SECTOR_SIZE));
    this.processor.jump(this.BOOT_ADDRESS);
    this.processor.execute();
  }
}
```

-Can also implement by having the data in the class most relevant to it, then having that class act as the facade, where it has methods that call low-level methods via a *private* member of that object. As long as interface in low-level modules stays the same, underlying logic can change w/o affecting facade.

-Downside: Facade itself becomes coupled to underlying classes. If facade becomes too complex, changes to underlying classes can cause a lot of change needed in the facade.

-Use facade when you want a simple single interface to hide a complex system of multiple components with multiple responsibilities/actors.

XII – Open-Closed Principle

-'A software artifact should be open for extension but closed for modification'

-Software should be built in a manner that makes it extensible by from the get-go. If extensions require changes to existing software, the design has failed.

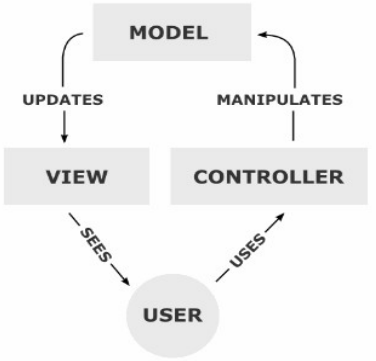
-This is enabled by separating responsibilities (single responsibility principle) and having a proper "hands off the back end code" interface via proper use of the dependency inversion principles

-Since software components are often just modules to display or format data, easy to separate based on one display as a model (ex. web formatted report module, printer formatted report module that both receive data from same financial analyzer).

-Changes in one responsibility should not affect another. Modules should also be built to be easily extended on.

-Example of split: controller interacts with pulls data from model and formats into rudimentary format, able to be read by many classes. Web reporter and print reporter models then both uses this pulled, organized raw data to display in different manners.

Model View Controller
https://www.sitepoint.com/the-mvc-pattern-and-php-1/



-Model – the data, including 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 sees user use then sends interaction to controller, which update model. Model then updates view.

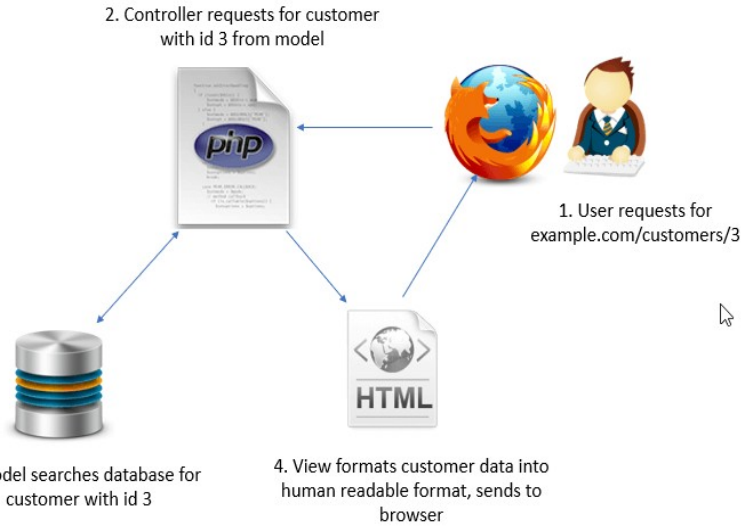
-If component A should be protected from changes in component B, then component B must not depend on component A

MVC Example
-Class Model – has *public* string initialized during construction

-Class View – holds *private $view* and *$controller* objects it initializes during construction. Has *output()* method that reads string from Model and returns it. Also has button that can be clicked.

-Controller – has *private $model* object that it initializes during construction. Has handling function for when button click that updates model
string when *view* button clicked.

-Use:
$model = new Model();
$controller = new Controller($model);
$view = new View($controller, $model);
echo $view→output();



MVC Example II

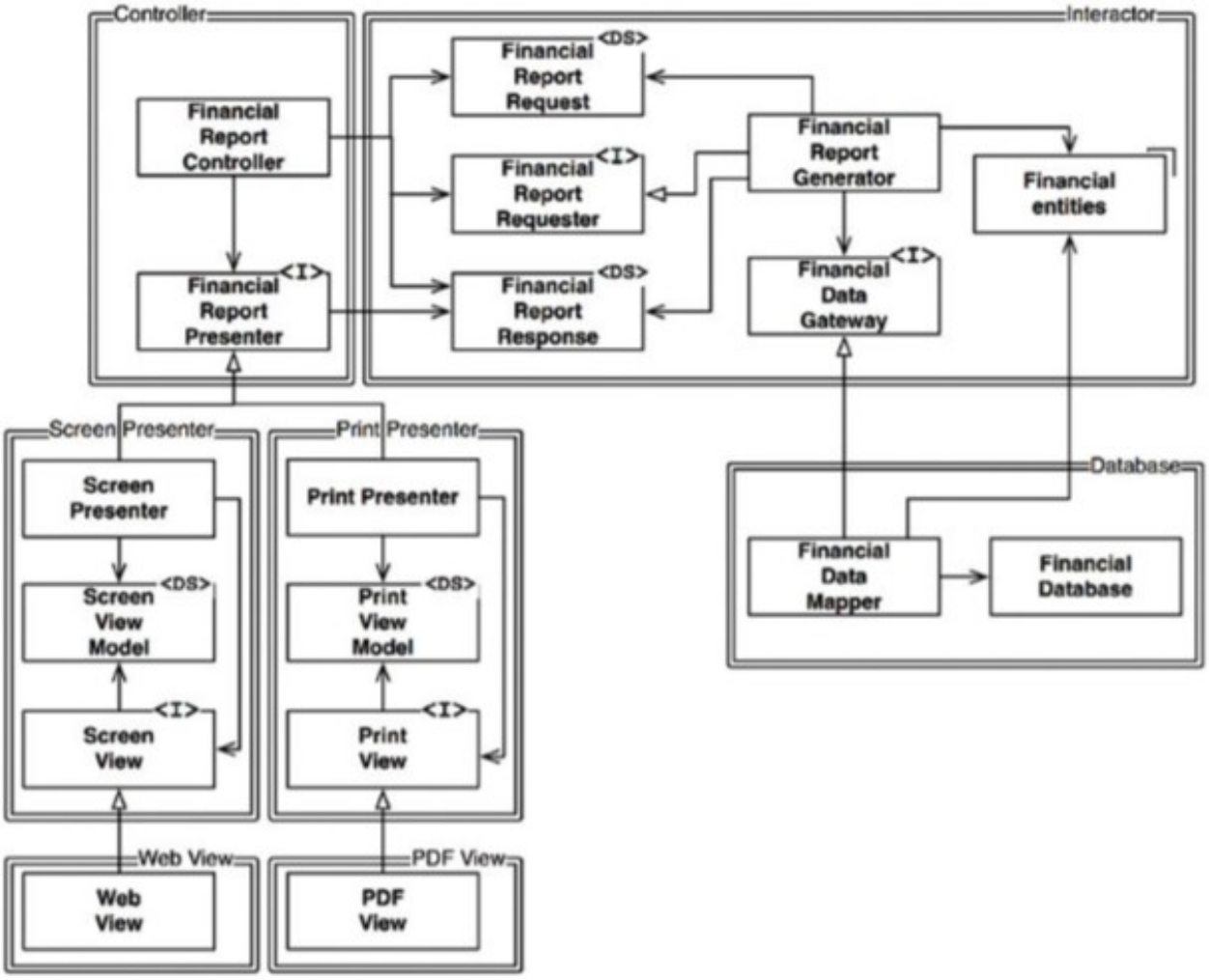


Figure 8.2 Partitioning the processes into classes and separating the classes into components

Classes marked with <I> are interfaces; those marked with <DS> are data structures. Open arrowheads are *using* relationships. Closed arrowheads are *implements* or *inheritance* relationships.

-Model - Database has financial data and methods to access via MySQL

-Interactor – acts as part of controller that serves as gateway to DB. Pulls data from DB and formats into different data structures. Also has method to access these data structors.

-Controller – Can access all data structors and access methods of interactor.

-View – presenter classes, which implement from view and can access interface method to get data from interactor. Once builds a screen view, the other builds a PDF view. These are returned from an interface. Final PDF and web views are classes that implement this interface.

-Note that the interactor is implemented by the Financial Data mapper portion of the DB class. The controller also uses the data structors and interface of the interactor. This is thus the highest level class, which contains both data structors and business logic, and what we want to protect from change. It builds the base rules. The controller simply has access to the output of those rules, and passes them to

the presenters, which prepare the view.

Directional Control
-By having uni-directional control, you keep the components you want most protected as classes that are inherited from, that have data pulled from, etc., but not the other way around.

-By having classes dependent on the highest-level classes only, uni-directionally, even if a lower level class that is dependent on the high-level class changes, the high level class stays unaffected.

Information Hiding
-Hide what information is not needed by lower-level classes in higher-level classes and simply provide and interface to what is needed.

-If this is not done, and a lower level class has direct access to data of a higher level class, then transitive dependencies exist.

XIII – Liskov Substitution Principle

-Objects should be replaceable with instances of their subtypes by adhering to a contract that allows them to be substituted for one another

-In order for this to occur, an inheriting class must have definitions for all methods defined in the parent class

-Failure example. *Rectangle* has *getWidth* and *getSide* methods. Inheriting class *Square* only has *getSide* method. It thus cannot be substituted for *Shape*.