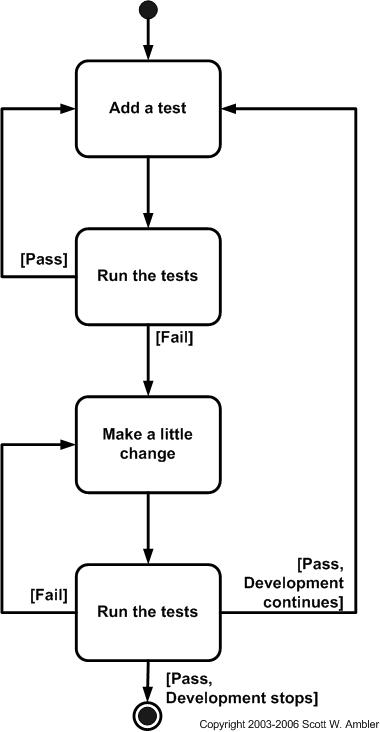
# [Clean Architecture Book Notes]

## [Chapter 1]

* The word “architecture” is often used in the context of something at a high level that is divorced from the lower-level details, whereas “design” more often seems to imply structures and decisions at a lower level. But this usage is nonsensical, and so it is with software design, The low-level details and the high-level structure are all part of the same whole.
* TDD by Jason Gorman: Test-Driven Development is a methodology in software development that focuses on an iterative development cycle where the emphasis is placed on writing test cases before the actual feature or function is written.It combines building and testing.( adapted by Agile S.D)



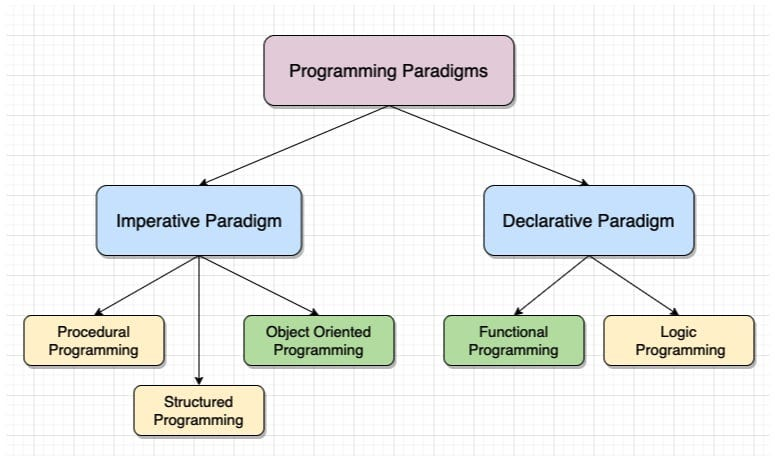
## [Chapter 2]

* Every software system provides two different values to the stakeholders: (behavior and structure).
* BEHAVIOR: machines behave in a specific way.
* ARCHITECTURE (structure) has to do with the word “software”—a compound word composed of “soft” and “ware.” The word “ware” means “product”; the word “soft” that’s where the second value lies, Software was invented to be “soft.” It was intended to be a way to easily change the behavior of machines.
* The first value of software—behavior—is urgent but not always particularly important.
* The second value of software—architecture—is important but never particularly urgent.
* Software architecture revolution was in programming paradigms, Paradigms are ways of programming, relatively unrelated to languages, A paradigm tells you which programming structures to use, and when to use them.

## [Chapter 3]

- Programming Paradigm [strategy, style followed while implementation]

* STRUCTURED PROGRAMMING: replaced the unrestrained jumps with if/then/else and do/while/until constructs. (discipline on direct transfer of control.)
* OBJECT-ORIENTED PROGRAMMING: The function became a constructor for a class, the local variables became instance variables, and the nested functions became methods. This led inevitably to the discovery of polymorphism through the disciplined use of function pointers. (discipline on indirect transfer of control.)
* FUNCTIONAL PROGRAMMING: a paradigm of building computer programs using expressions and functions based on mathematical functions without mutating state and data.(discipline upon assignment.)



## [Chapter 4]

### -STRUCTURED PROGRAMMING:

* Structured programming is a program written with only the structured programming constructions:

(1) sequence, (2) repetition, (3) selection

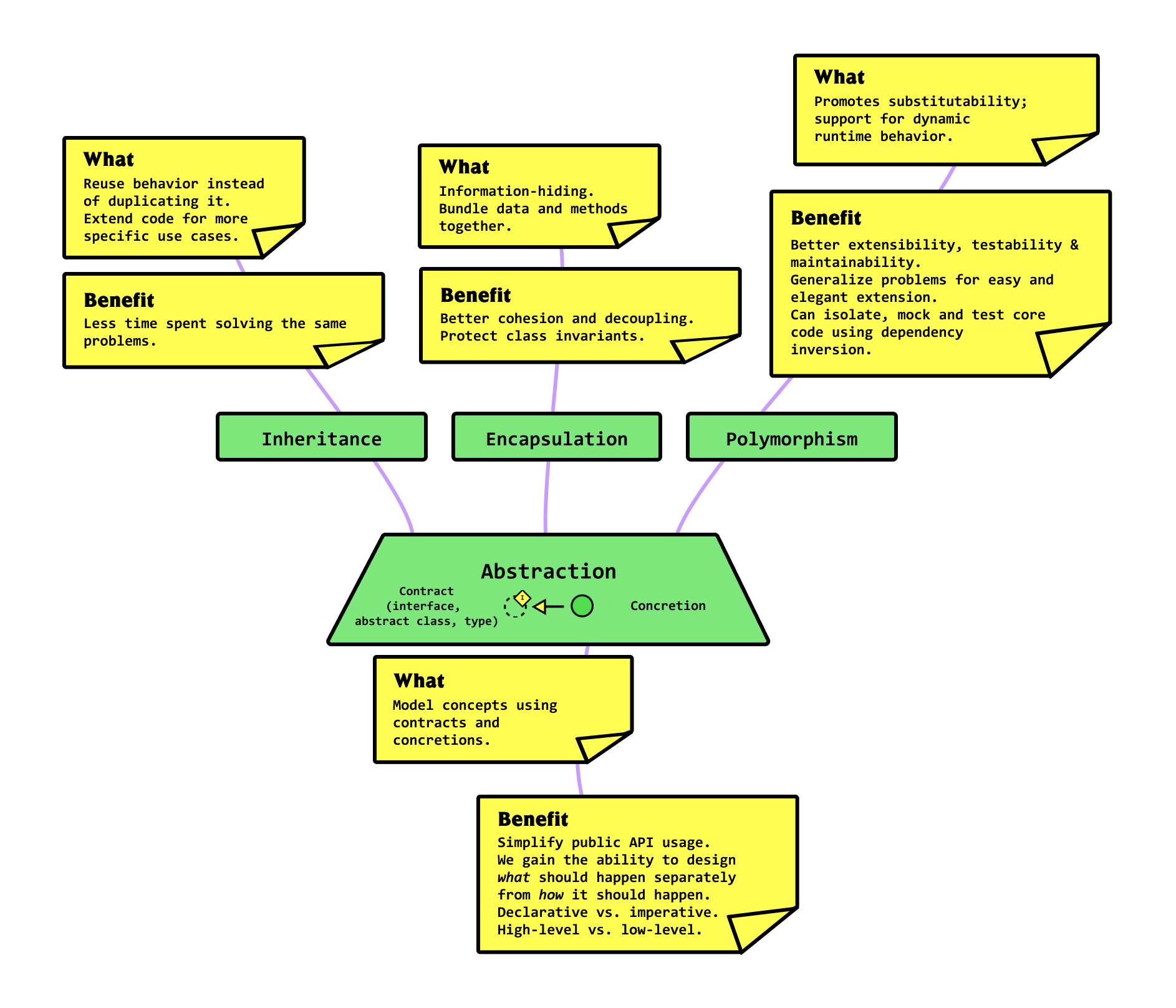
* Structured programming forces us to recursively decompose a program into a set of small provable functions. We can then use tests to try to prove those small provable functions incorrect. If such tests fail to prove incorrectness, then we deem the functions to be correct enough for our purposes.
* Old language used go to () whish cause unstructured code, then structured code replaces it with (if, else, then, switch cases, while, do ,until) aims to organized code using the control of transfer statement.
* Main concepts: 1) top-down analysis.

2)modular programming.

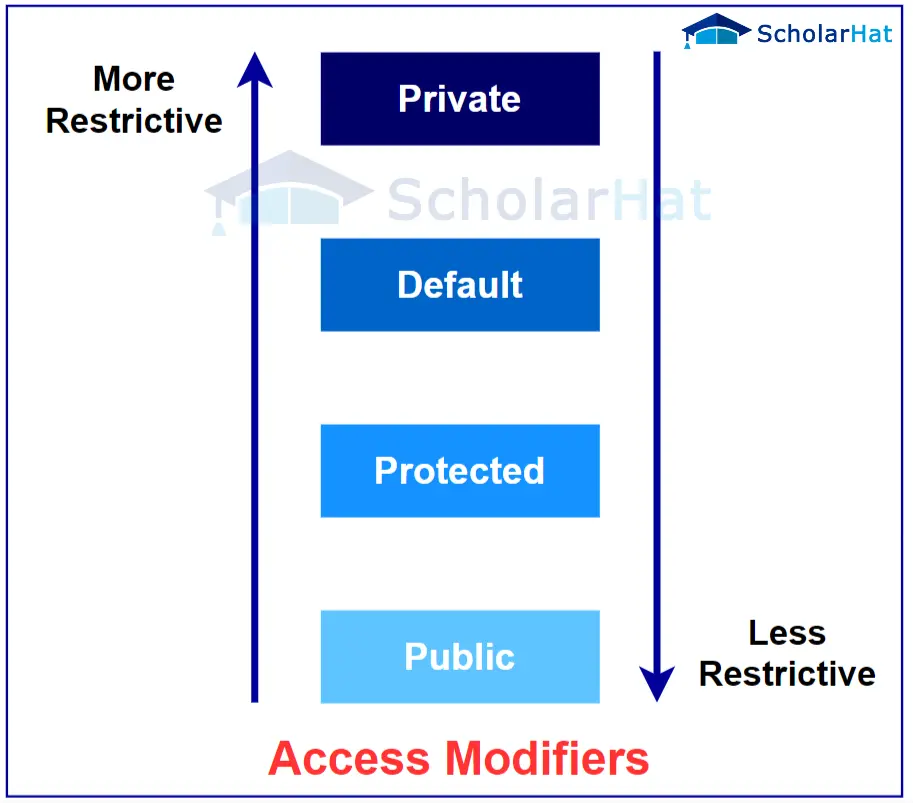
3)structured code

## [Chapter 5]

### -OBJECT-ORIENTED PROGRAMMING (modeling the real world)



* Encapsulation : restrict the direct access to some components of an object ,can be data members and data functions.
* By access modifiers(public , private, protected, default)



* Inheritance: allows programmers to create classes that are built upon existing classes, to specify a new implementation while maintaining the same behaviors (realizing an interface), to reuse code.
* POLYMORPHISM: gives a program the ability to redefine methods for derived classes.
* Leveraging abstraction and polymorphism are key techniques for implementing the Dependency Inversion Principle (DIP) in your software design
* Dependency inversion: high-level modules depend on abstractions rather than concrete implementations (high-level& low-level both depend on the abstraction).

|  |  |  |
| --- | --- | --- |
| **S.NO** | **Abstraction** | **Encapsulation** |
| 1. | Abstraction is the process or method of gaining the information. | While encapsulation is the process or method to contain the information. |
| 2. | In abstraction, problems are solved at the design or interface level. | While in encapsulation, problems are solved at the implementation level. |
| 3. | Abstraction is the method of hiding the unwanted information. | Whereas encapsulation is a method to hide the data in a single entity or unit along with a method to protect information from outside. |
| 4. | We can implement abstraction using abstract class and interfaces. | Whereas encapsulation can be implemented using by access modifier i.e. private, protected and public. |
| 5. | In abstraction, implementation complexities are hidden using abstract classes and interfaces. | While in encapsulation, the data is hidden using methods of getters and setters. |
| 6. | The objects that help to perform abstraction are encapsulated. | Whereas the objects that result in encapsulation need not be abstracted. |

## [Chapter 6]

### - FUNCTIONAL PROGRAMMING:

* a paradigm of building computer programs using expressions and functions without mutating state and data(variables doesn’t vary-immutable)

-----------------------------------------------------------------

• Structured programming is discipline imposed upon direct transfer of control.

• Object-oriented programming is discipline imposed upon indirect transfer of control. (restricted by polymorphism)

• Functional programming is discipline imposed upon variable assignment.

## [Chapter 7]

### - DESIGN PRINCIPLES (SOLID)

* how to arrange our functions and data structures into classes, and how those classes should be interconnected.
* 5 design principles that are used in object-oriented software development.

### SRP: SINGLE RESPONSIBILITY PRINCIPLE

* A module should have one, and only one, reason to change.
* A module should have one, and only one, actor.

-Symptoms of violating it:

1. ACCIDENTAL DUPLICATION: separate the code that different actors depend on.
2. MERGES

* The Single Responsibility Principle is about functions and classes—but it reappears in a different form at two more levels. At the level of components, it becomes the Common Closure Principle. At the architectural level, it becomes the Axis of Change responsible for the creation of Architectural Boundaries.

## [Chapter 8]

### OCP: OPEN-CLOSED PRINCIPLE

* A software artifact should be open for extension but closed for modification.
* behavior can be extended without undo modification.
* Architects separate functionality based on how, why, and when it changes, then organize that separated functionality into a hierarchy of components.
* Higher-level components in that hierarchy are protected from the changes made to lower-level components.
* partitioning the system into components, and arranging those components into a dependency hierarchy

## [Chapter 9]

LSP: THE LISKOV SUBSTITUTION

* if we substitute a superclass object reference with an object of *any* of its subclasses, the program should not break.
* <https://www.youtube.com/watch?v=uzWgmME74W4>

## [Chapter 10]

ISP: THE INTERFACE SEGREGATION

* The lesson here is that depending on something that carries baggage that you don’t need can cause you troubles that you didn’t expect.
* No client should depend on a method it doesn't use.
* Separate the interface.

## [Chapter 11]

DIP: THE DEPENDENCY INVERSION

* High-level modules should not depend on low-level modules, both should depend on abstraction.
* Abstraction should not depend on details,

details should depend on abstraction.