GRASP

- It refers to a collection of design principles and patterns that explain how responsibilities and tasks should be in the software design process.

- GRASP principles and patterns are:
    - **Information Expert:** The class that can fulfill a responsibility should take that responsibility. This principle states that the classes with which the information is directly related or those with the easiest access to this information should bear the responsibility.

      For example, let's consider a customer order processing system. The customer class is responsible for updating its own information because it is the class closest to the customer information.

    - **High Cohesion:** Items (variables, objects, methods) with similar functions should be put together. Methods and data elements within the same class should be closely associated with each other.

    - **Low Coupling:** Dependencies and relationships between classes should be minimized. Ideally, the interdependence of classes should be weak.

    - **Creator:** If a class creates objects of other classes, that class has the responsibility of creating new objects.

      For example, in an order management application, the creation of orders should be the responsibility of the order class. So, the order class must create order objects.

    - **Controller:** The class that processes user interactions and coordinates the system should be designated as the controller**.**

      For example, let's consider a web application. There is a controller class that directs user interactions. This processes the user's requests and calls the relevant business logic.

    - **Polymorphism:** If subclasses can implement a superclass, the task of ensuring the administration of the superclass is assigned to the subclass.

    - **Pure Fabrication:** Classes that do not exist in the real world can be designed to improve system design and increase compliance with other principles.

      For example, when writing a game engine, you can create abstract classes that do not exist in the real world, such as a physics engine class. These classes are designed to suit the needs of the game and are not related to real-world entities.

Cohesion

- It refers to how compatible and related the elements within a software component or module are to each other.

- Different jobs should be separated to be done in different places. If the entire structure, all fields, and methods of a class are for the same common purpose, then the cohesion of that class is high.

- Cohesion levels are as follows, from highest to lowest:
    - **Functional Cohesion:** All methods within a component performs the same function. This is the highest level of cohesion.

    - **Sequential Cohesion:** These are classes that bring together methods that work as pipes, where the output of one feed the other at the class level.

    - **Communicational Cohesion:** Methods within a component manipulate the same data or data structure. Methods are used to access and manipulate this data.

    - **Procedural Cohesion:** Functions within a component are parts of a parent function and often share the same inputs or outputs. Functional separation of tasks related to a subject from top to bottom and bringing them all together in one class.

    - **Temporal Cohesion:** Functions within a component are called at the same time or in a specific order, but there is no other type of cohesion.

    - **Logical:** These are structures brought together that are thought to be related to a single thing, although they are actually of different nature.

    - **Coincident Cohesion:** There is no clear relationship between functions within a component. Functions exist only in the same component, but do not rely on a context or logic. This is the lowest level of cohesion and should generally be avoided.

Cohesion anti-patterns

- It generally results from confusing the roles of objects in architecture and functional structure and not paying attention to the differences between them.

- According to Meilir Page-Jones, there are 3 common types mentioned in his book [Fundamentals Of Object-Oriented Design in UML.]

    - **Mixed-Instance Cohesion:** In this case, some properties of a class are valid for some objects and not for others. This indicates a poorly defined class hierarchy.

      It is common in environments where the concept of object is not well established and object-centered techniques are not well applied, and where data models predominate.

      The attributes after the salary section on the side are not correct for this class. The *type* variable will determine the employee's type, for example manager or director. This causes the creation of methods in the written codes that become longer with if-else structures and will become even longer as new types are introduced in the long run. Therefore, we can parse from this point and create classes of different types.

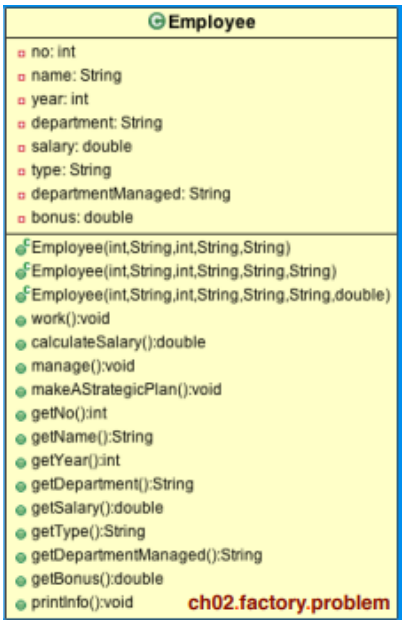


Figure 1.1 lowly cohesive

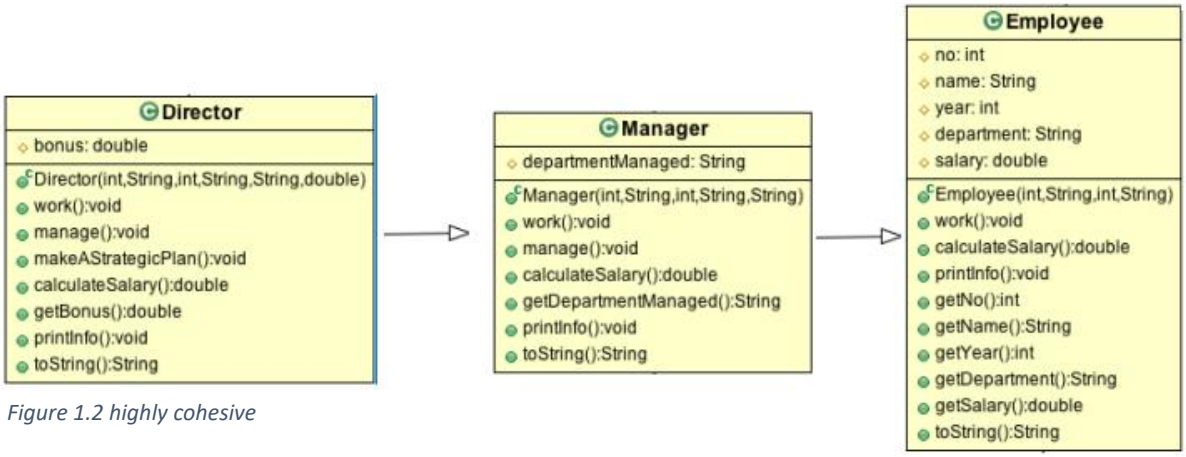


Figure 1.2 highly cohesive

- **Mixed – Domain Cohesion:** It refers to the combination of methods or features belonging to different domains within a component, module or class.

    - **Application Domain:** Contains application-specific objects. Objects responsible for coordinating entities to realize use cases. Events and event handler objects. Workflow etc. objects can be given as examples.

    - **Business Domain:** Objects that represent the business are entity, enum and interface objects. These are **value objects** that will be used as fields of entities but are not entities themselves. Address, PhoneNumber etc. identity with objects, DTO.

    - **Foundation Domain:** Contains the most basic objects. Programming language types, classes such as String and Thread whose code cannot be interfered with. Objects of frequently used libraries, etc.

    Objects should be specific to their fields; more than one field should not be represented by a single object.

    Let's say a software system is being developed for an automobile manufacturing factory, and a component of this system is called "Automobile Manufacturing Component". For automobile production, this component includes classes such as Production Planning, Material Supply, Production Line Control, Product Quality Control, Sales and Distribution Management.

    This component gathers all the functionality of an automobile manufacturing plant in one place. However, these different methods belong to different business areas. While production planning, material sourcing and production line control focus on the production process, sales and distribution management deals with marketing and delivering products to customers.

- **Mixed-Role Cohesion:** The situation where objects with different roles within the same domain have their properties combined into a single object. This is quite common, especially in entity objects.

    Let there be a class called Person and this class should have attributes such as numberOfDogs, numberOfCC. What is expected from the Person class here is that it has the characteristics of a human; different people may not have a dog or a credit card, or they may have one of the two, or they may have one for a while and then lose their ownership. Such different roles should not be collected in the same class.

Object Categorization

- Bringing together the responsibilities and data appropriate to the roles and separating those belonging to different roles is the most basic factor that increases unity.

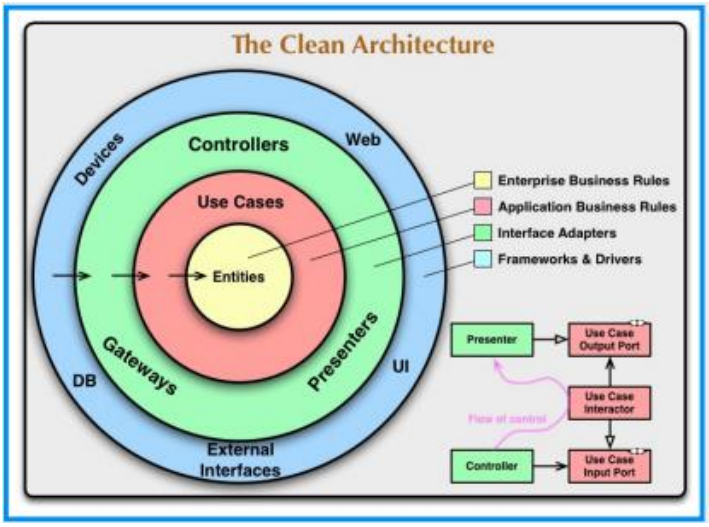
- Ivar Jacobson classifies objects as follows:
  - **Boundary:** These are the objects that manage the communication of the system with its actors, also called interface objects. They are objects that manage communication between a software system and the outside world or users.

  - **Control:** Control objects are objects that manage business processes and know the relevant business rules. They are often used as components that implement business logic or business processes. These objects are often called "**services**" and are where business rules are applied. For example, in a banking application, a "MoneyTransferService" object that manages money transfer transactions can be given as an example of a control object. This service enforces the rules of money transfer transactions.

  - **Entity:** Entity objects represent the business domain or concepts within the business domain. They usually contain data and data processing logic. They can be associated with database tables or data classes. For example, for a customer database, a data class or object named "Customer" can be given as an example of an entity object. This object represents a structure that stores and processes customer data.

- Another object categorization is **Hexagonal Architecture** (onion, Ports & Adapters).
  - Proposed by Alistair Cockburn.
    https://alistair.cockburn.us/hexagonal-architecture/

  - It was developed later.



http://wiki.c2.com/?HexagonalArchitecture

https://blog.octo.com/en/hexagonal-architecture-three-principles-and-an-implementation-example/

https://madewithlove.com/

Coupling

- It is a measure of how expressive a task is on its own or how it relates to others. As the interrelatedness between objects increases, the complexity between them also increases. For example, a change you make in one class may affect a completely different class. This can cause many problems.

- Coupling types from worst to best:
    - **Content Coupling:** These are situations in which structures depend on each other's implementations. The main reason is wrong abstraction.

    - **Common Coupling:** It is the dependency between structures that use global data variables.

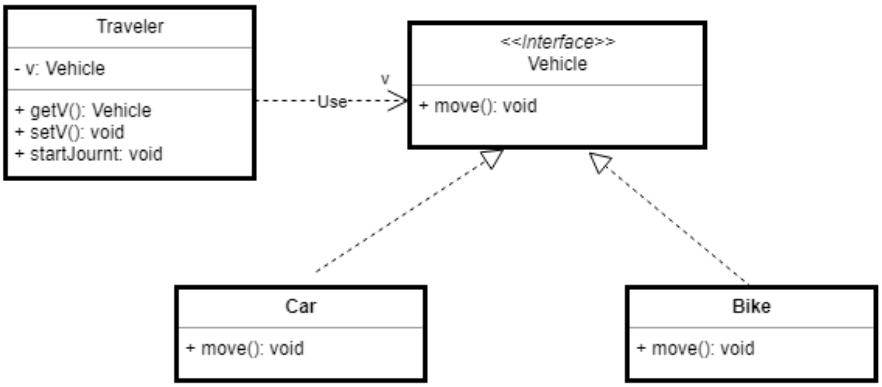
    - **Control Coupling:** It is the dependency in which structures control each other's flows by passing flags. It is a special case of data dependency.

    - **Data Coupling:** It is the coupling formed by components passing simple/primitive/atomic data to each other. Components communicate only through data exchange without interfering with each other's internal structure.

    - **Implementation Inheritance:** It is the situation of inheriting a class. It is problematic because classes inherit each other directly and a content dependency is established between classes. It is recommended not to use it other than structural type similarity.

    - **Message Coupling:** It is a form of dependency that does not require any information other than the interface information of the object. The object is communicated with using the API.

    - **Abstract Coupling:** It is a dependency on types that are abstract rather than concrete. In other words, we can say that it is a dependency on interfaces. Objects only determine the supertype that determines each other's interfaces, they do not know their real type.



Here, the ***Traveler*** class does not need to know what the class that implements the ***Vehicle*** interface given to it is. It is enough for it to be a ***Vehicle*** and here an abstract coupling is created.

- https://thoughtbot.com/blog/types-of-coupling

SOLID Principles

- ***S*ingle Resposibility Principle**: “There should never be more than one reason for a class to change.”

  A class should abstract one and only one thing and focus only on it, have data about it and fulfill responsibilities. Therefore, a class should change only for reasons related to an abstraction.

  We can define SRP for lower and upper levels of class level as follows:
  - **Package:** Structures released together must be in the same package.
  - **Class:** A class should abstract only one thing and have only data and behavior related to that thing.
  - **Method:** It should do a single indivisible job that can be reused, related to what the class abstracts. Different types such as constructor, getter/setter should not go outside the type context. That is, a constructor should not perform any work other than producing objects.

    They must perform an atomic algorithm, a calculation, or a single step of a complex process.
  - **Block:** It should be a group of sentences that have not reached the method level and therefore cannot be reused, but work in an all-or-nothing manner.
  - **Statement:** It should carry out a single step of a task in a comfortable, understandable way as part of a method or block. For example, complex ternary expressions or single-line loops such as "while(move())" whose contents cannot be fully understood should not be used.

- ***O*pen-Closed Principle: “**Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.”

  Software systems are constantly changes because of new feature, new user, new business rules, etc. In these cases, we need to expand existing structures rather than replace them. Existing interfaces, classes and methods must continue to live without any changes.

  When designing the structure from the beginning, it should be designed to take change into consideration. The parts that can change should be designed in isolation from the parts that will not change, and dependencies to be avoided should be provided through abstractions as much as possible. This makes it easier to make innovations without changing existing structures.

- ***L*iskov Substitution Principle: "**Functions that use pointers or referenes to base classes must be able to use objects of derived classes without knowing it."

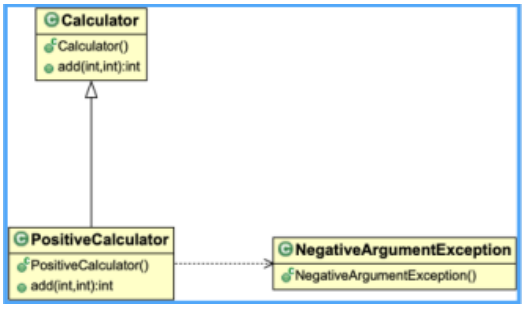
  In summary, subclasses should be usable instead of superclasses.

  LSP also covers **Design by Contract (DbC**). The code must be written accordingly.

  DbC: Clients that know the superclass should not be restricted when trying to use subclasses. However, the service they receive may be more special than they expected.

  For example, an exception not thrown to a method of the base class should not be thrown after it has been overridden by the subclass.

  So, the following situation does not comply with **DbC**.



- ***I*nterface Segregation Principle: "**Clients should not be forced to depend upon interfaces that they do not use."

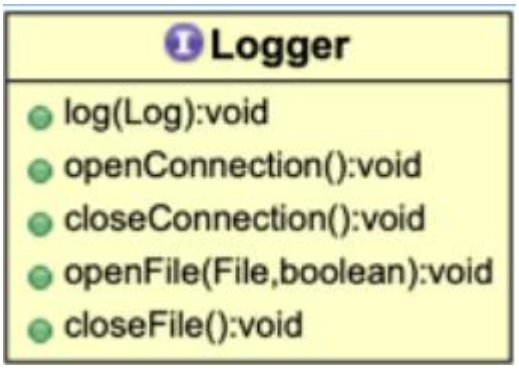
  If an interface has service groups consisting of methods that serve different clients, this interface should be divided into thin interfaces specific to its clients.

  There are two typical indicators of **anti-ISP** status:
  - Different clients call different methods on the same interface. This leads to high complexity.

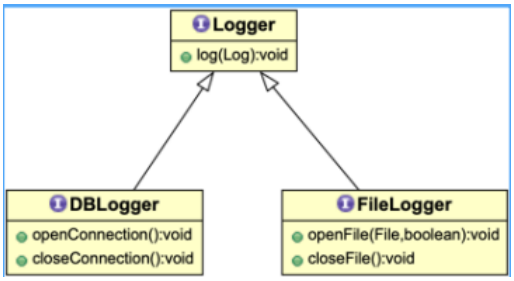
  - Subclasses of an interface have difficulty implementing or adapting some methods. A subclass may not properly implement or override the methods it inherits. In this case, the subclass may want to mark or handle these methods by throwing exceptions or using different methods. However, this is against LSP and DbC principles, which are the design principles of the program, and may force clients (users) to use runtime type information (RTTI).

  When these situations occur, the interface should be divided and clients should see only the methods they are interested in, and subclasses should only inherit and implement the methods they need.

Here, the **Logger** interface has been created assuming that logs will be made to both the file and the database. In this case, subclasses that only want to log to the database or file will not be able to implement some methods. Similarly, with this interface, clients who only want to log to the database or file will have to see irrelevant methods.



It is correct to divide the **Logger** interface into two interfaces with high cohesion.



- ***D*ependency Inversion Principle: "**High level modules should not depend upon low level modules. Both should depend upon abstractions. Abstractions should not depend upon details, details should depend upon abstractions."

  - **High Level Modules and Low-Level Modules:** The basis of **DIP** is to divide software components into high level modules and low-level modules.

    High-level modules: they do business logic or application-level work.

    Low-level modules: Responsible for the details required to implement this functionality.

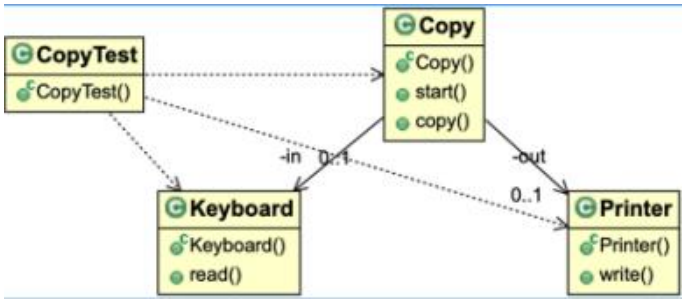
    For example, the functionality to send an email. The high-level module determines how to handle sending emails, while the low-level module is responsible for the details of communicating with the email server.

  - **Dependency Inversion:** DIP recommends that high-level modules do not have direct dependencies on low-level modules and that these dependencies are reversed.

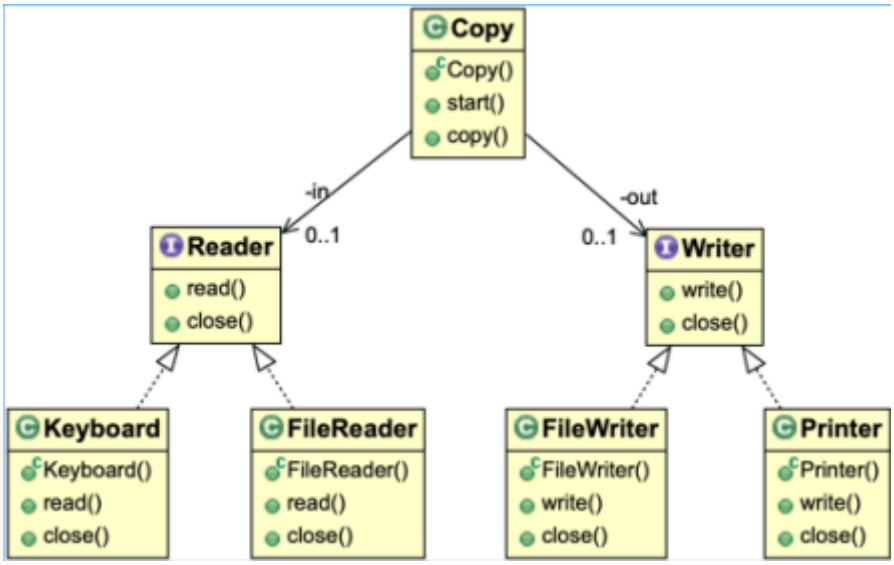
    That is, the high-level module does not need to know how the low-level module works or is implemented. Instead, both modules communicate with each other via abstracted interfaces. This provides greater flexibility and resistance to changes.

- **Abstractions and Interfaces:** DIP recommends programmers use abstractions and interfaces. When interacting with the low-level module, the high-level module should depend on the low-level module's abstractions and interfaces, not its implementations. In this way, even if the internal structure of the low-level module changes, the high-level module is not affected.

The Copy class here does not work without the Keyboard and Printer objects, and if there is a change in these objects, it is very likely to be affected.



If we transform this structure according to DIP, we get a structure like this.



Law Of Demeter

- It argues that in order to do its job, an object must know a small number of objects from which it will inevitably receive service.

- A class should only interact with the following objects:
  - Himself.
  - Objects that come as parameters of the function.
  - Objects created by itself.
  - Component objects within their own class.

- A class should not directly access “distant objects” such as:
  - Global variables.
  - Program-wide objects (such as Singleton patterns).
  - Objects inside objects that come as parameters.
  - Objects inside objects returned by methods.