Before starting with design patterns, I will cover the systems approach and SOLID principles.

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.

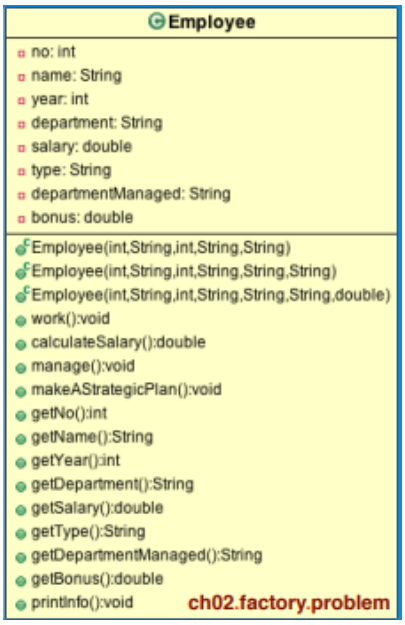


Figure 1.1 lowly cohesive

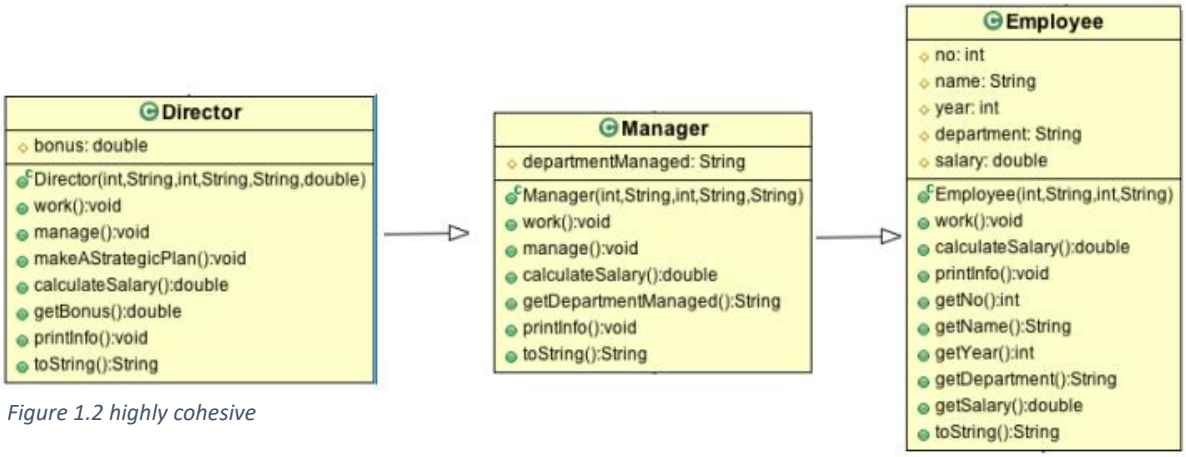


Figure 1.2 highly cohesive

- **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.

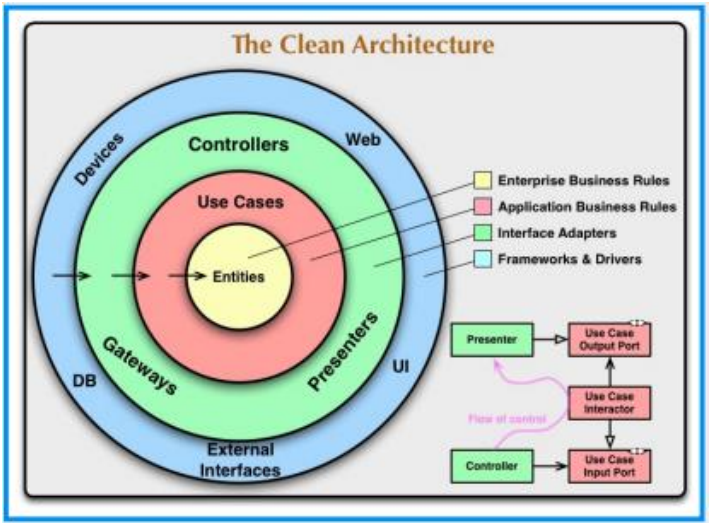
- Ivar Jacobson classifies objects as follows:
  - **Boundary:** These are the objects that manage the communication of the system with its actors, also called interface objects. They are objects that manage communication between a software system and the outside world or users.

  - **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.

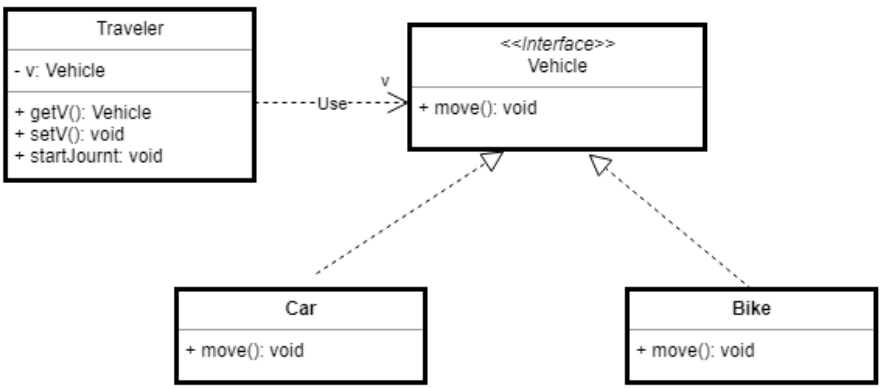
    - **Control Coupling:** It is the dependency in which structures control each other's flows by passing flags. It is a special case of data dependency.

    - **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