SOFTWARE DESIGN

EGCO343 SOFTWARE DESIGN

KANAT POOLSAWASD
DEPARTMENT OF COMPUTER ENGINEERING
MAHIDOL UNIVERSITY

DESIGN MODEL (1)

- Design model is the place where customer requirements, business needs, and technical considerations come together in the formation of a product or system.
- Design creates a representation or model of the software
  - Design model (Unlike analysis model) provides detail about:
    - Software data structures
    - Architecture
    - Interfaces
    - Components that are necessary to implement the system

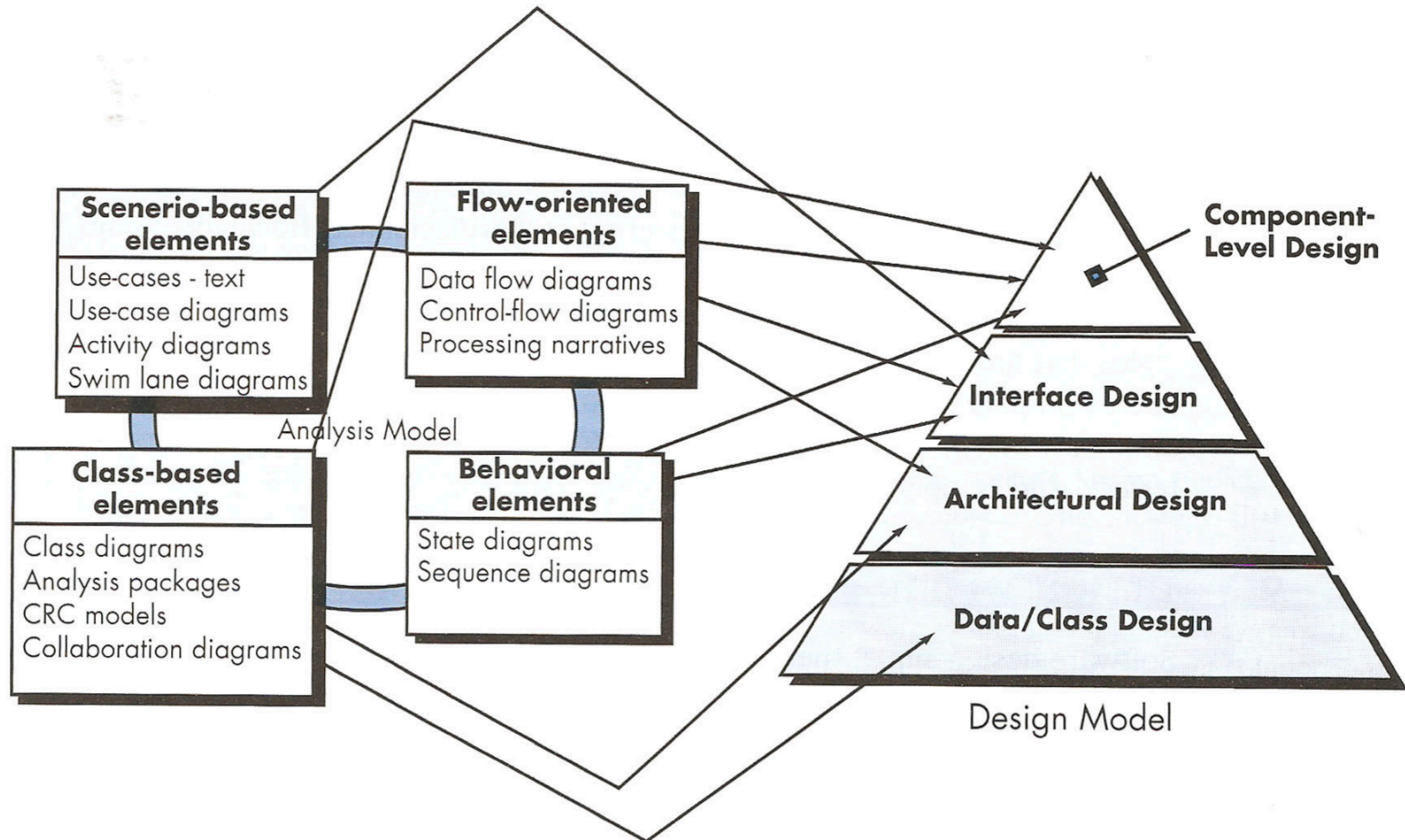
DESIGN MODEL (2)

- Design allows a software engineer to model the system or product
- This model can be assessed for software quality before:
  - Code is generated
  - Tests are conducted
  - End-users become involved in large numbers
- Design is the place where **software quality is established**

WHAT ARE THE STEPS ?

- Design depicts the software in a number of different ways:
  - Architecture of the system or product
  - Interfaces that connect the software to:
    - End-users
    - Other systems and devices
    - Components
  - Software components that are used to construct the system

TRANSLATING THE ANALYSIS MODEL INTO THE DESIGN MODEL



DESIGN CONCEPTS (1)

- **Abstraction**
  - Procedural abstraction – a sequence of instructions that have a specific and limited function.
  - Data abstraction – a named collection of data that describes a data object.

LEVEL OF ABSTRACTION

- Procedural Abstraction refers to a sequence of instructions that have a specific function
  - Name of procedural abstraction implies the functions
  - Example of a procedural abstraction
    - The word Open for "a door"
    - Open implies a long sequence of procedural steps
- Data Abstraction is a named collection of data that describes a data object
  - Context of the procedural abstraction open,
    - Define a data abstraction called door
    - Encompass a set of attributes that describe the door

LEVEL OF ABSTRACTION (PRACTICE)

- Write low level of abstraction for both procedural and data abstraction — *openDoor();*

Procedural Abstraction	Data Abstraction
High Level: *openDoor()*	High Level: *Door ID / Key ID*
Low Level:	Low Level:

DESIGN CONCEPTS (2)

- **Architecture**
  - The overall structure of the software and the ways in which the structure provides conceptual integrity for a system.
  - Consists of components, connectors, and the relationship between them.
- **Patterns**
  - A design structure that solves a particular design problem within a specific context
  - It provides a description that enables a designer to determine whether the pattern is applicable, whether the pattern can be reused, and whether the pattern can serve as a guide for developing similar patterns

DESIGN CONCEPTS (3)

- **Modularity**
  - Separately named and addressable components (i.e., modules) that are integrated to satisfy requirements (divide and conquer principle)
  - Makes software intellectually manageable so as to grasp the control paths, span of reference, number of variables, and overall complexity
- **Information hiding**
  - The designing of modules so that the algorithms and local data contained within them are inaccessible to other modules
  - This enforces access constraints to both procedural (i.e., implementation) detail and local data structures

DESIGN CONCEPTS (4)

- **Functional independence**
  - Modules that have a "single-minded" function and an aversion to excessive interaction with other modules
  - High cohesion – a module performs only a single task
  - Low coupling – a module has the lowest amount of connection needed with other modules
- **Stepwise Refinement**
  - Development of a program by successively refining levels of procedure detail
  - Complements abstraction, which enables a designer to specify procedure and data and yet suppress low-level details

DESIGN CONCEPTS (5)

- **Refactoring**
  - A reorganization technique that simplifies the design (or internal code structure) of a component without changing its function or external behavior
  - Removes redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failures
- **Design Classes**
  - Refines the analysis classes by providing design detail that will enable the classes to be implemented
  - Creates a new set of design classes that implement a software infrastructure to support the business solution

ARCHITECTURAL PATTERNS

ARCHITECTURAL PATTERNS

- Architectural patterns help to specify the fundamental structure of a software system, or important parts of it.
- Architectural patterns have an important impact on the appearance of concrete software architectures
- Define a system's global properties, such as …
  - How distributed components cooperate and exchange data
  - Boundaries for subsystems
- The selection of an architectural pattern is a fundamental design decision; it governs "every" development activity that follows

EXAMPLE OF PATTERNS

- These describe how code and components inside an application are organized. Common types include:
  - Layered (N-Tier) Architecture
  - Pipes and Filters Pattern
  - Ports and Adapters Pattern
  - Broker Pattern
  - Observer
  - Publish-Subscribe
  - Model–View–Controller (MVC)
  - Etc.

ABSTRACT MACHINE (LAYERED)

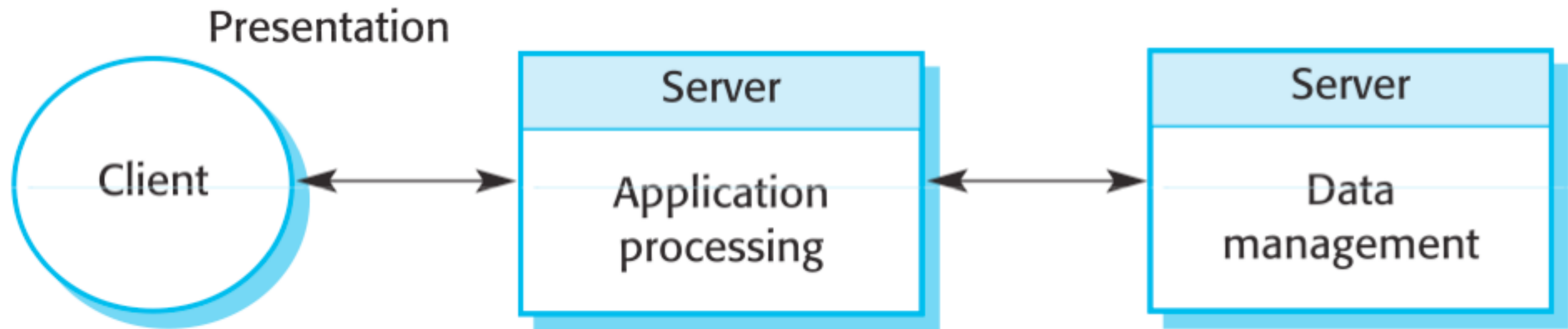
ABSTRACT MACHINE (LAYERED) MODEL

- Used to model the interfacing of sub-systems.
- Organizes the system into a set of layers (or abstract machines) each of which provide a set of services.
- Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.
- However, often artificial to structure systems in this way.

LAYERED APPLICATION ARCHITECTURE

- **Presentation layer**
  Concerned with presenting the results of a computation to system users and with collecting user inputs.
- **Application processing layer**
  Concerned with providing application specific functionality e.g., in a banking system, banking functions such as open account, close account, etc.
- **Data management layer**
  Concerned with managing the system databases.

THREE-TIER ARCHITECTURE

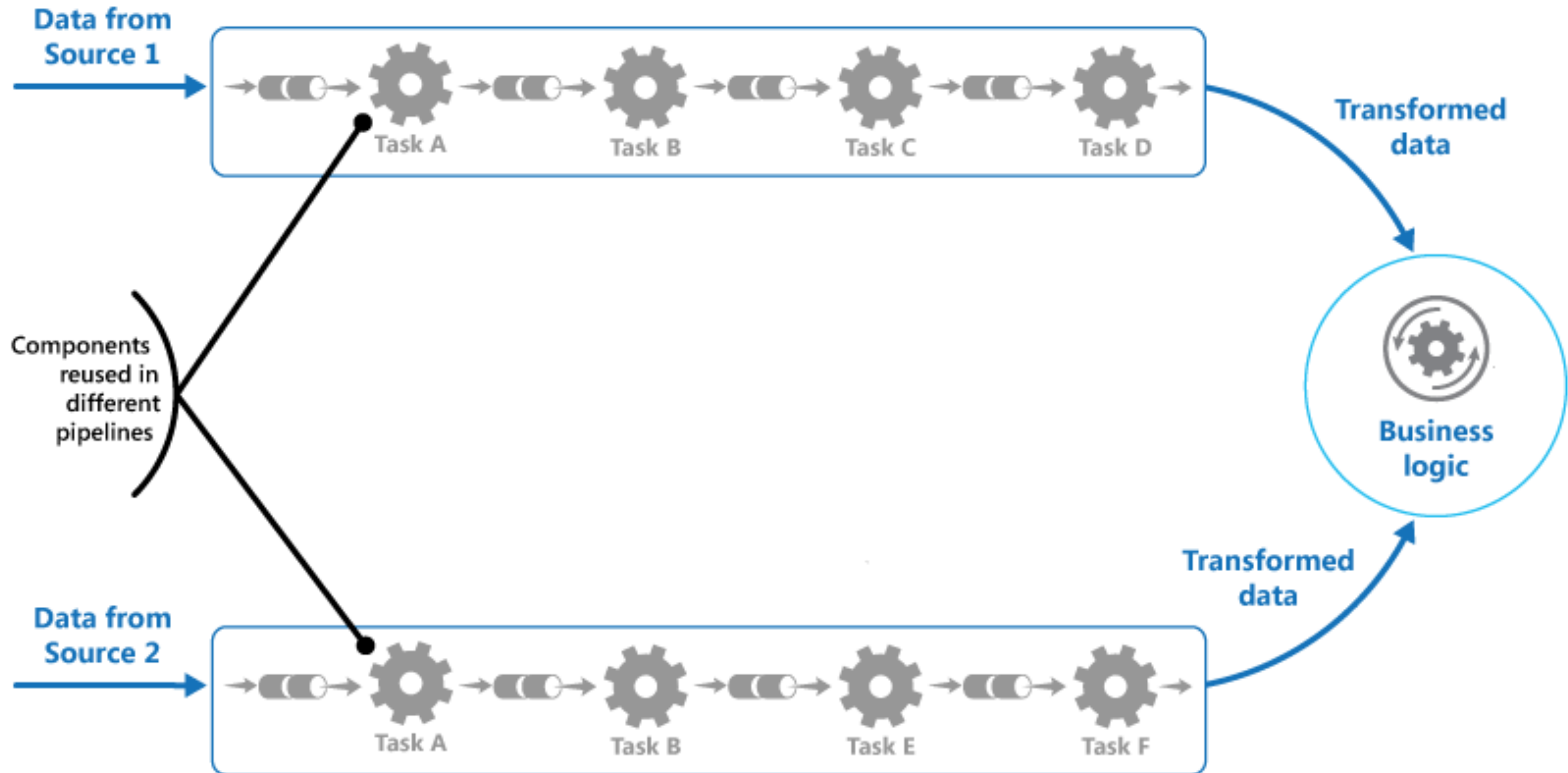


PIPE-AND-FILTER PATTERN

PIPE-AND-FILTER PATTERN (1)

- Components: Filters that read input data stream and transform it into output data stream
- Connectors: Pipes that provide output of filter as input to other filter
- Advantages: Simple, no complex interaction, high reusability, portability
- Disadvantages: Require common data format, no shared state, redundancy in (un)parsing
- Example: unix shell (ls –l | grep key | more ...)

PIPE-AND-FILTER PATTERN (2)

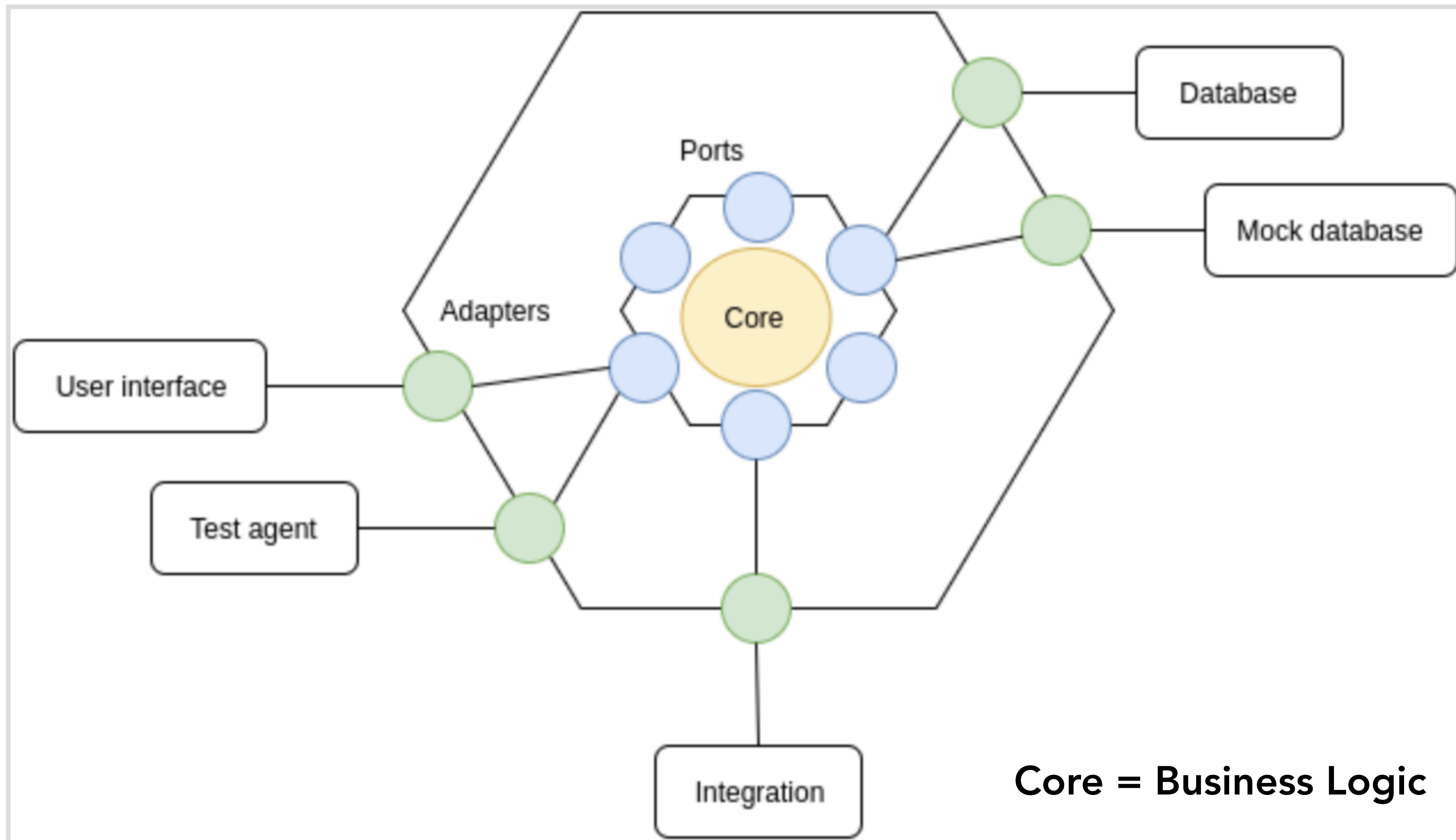


PORTS AND ADAPTERS PATTERN

PORTS AND ADAPTERS PATTERN (1)

- The objective of the Ports and Adapters pattern (Hexagonal Architecture) is to create a flexible and maintainable application architecture by decoupling the core business logic (the domain) from external systems such as databases, user interfaces, and third-party services.
- This pattern achieves that by defining “ports” as abstract interfaces through which the application interacts with the outside world, and “adapters” as concrete implementations of those interfaces for specific technologies or frameworks.

PORTS AND ADAPTERS PATTERN (2)

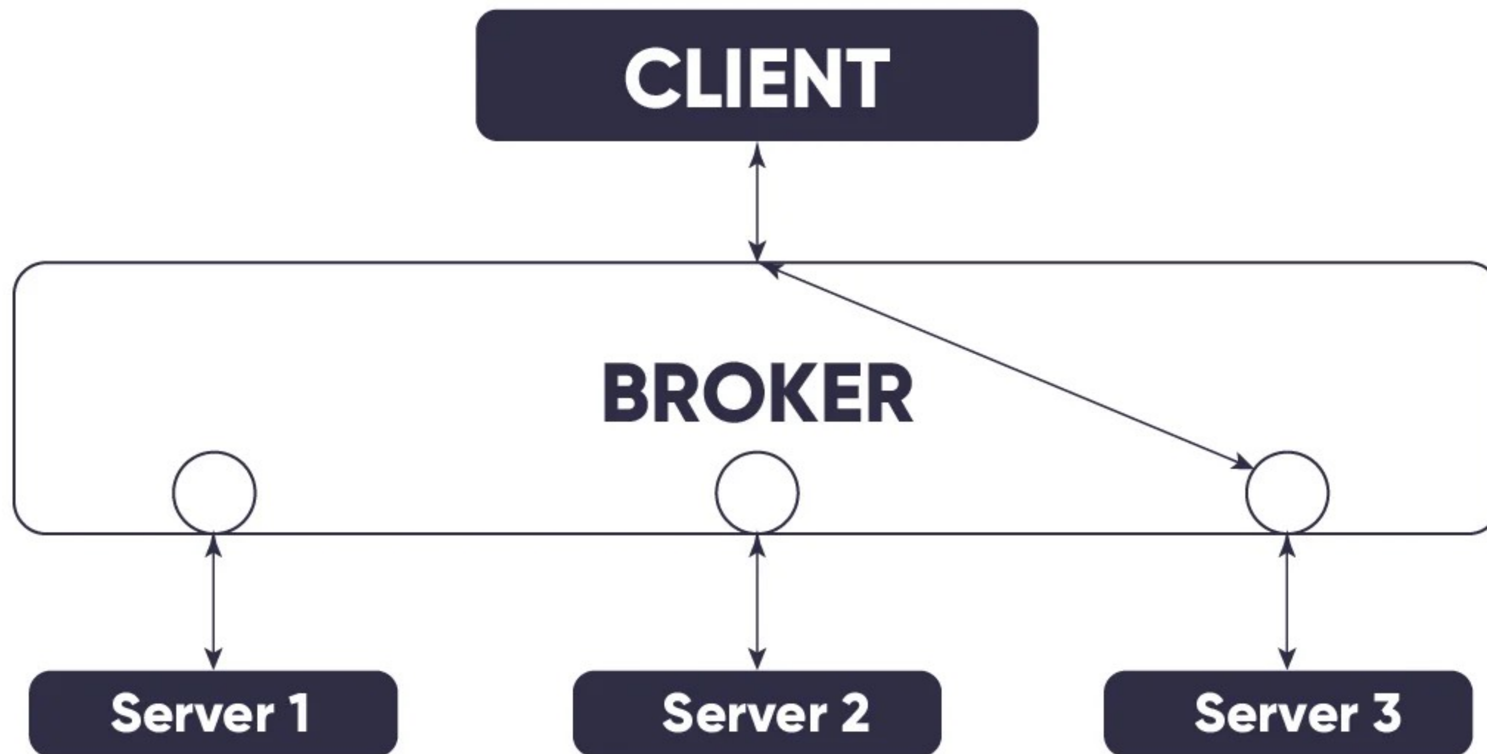


BROKER PATTERN

BROKER PATTERN (1)

- The Broker Pattern enables communication between decoupled components in a distributed system through a central broker that handles service registration, request routing, and result delivery — promoting loose coupling and scalability.

BROKER PATTERN (2)



OBSERVER & PUBLISH-SUBSCRIBE

OBSERVER PATTERN

- The Observer pattern is a behavioral design pattern that defines a one-to-many dependency between objects so that when one object (called the subject) changes its state, all its dependent objects (observers) are automatically notified and updated.

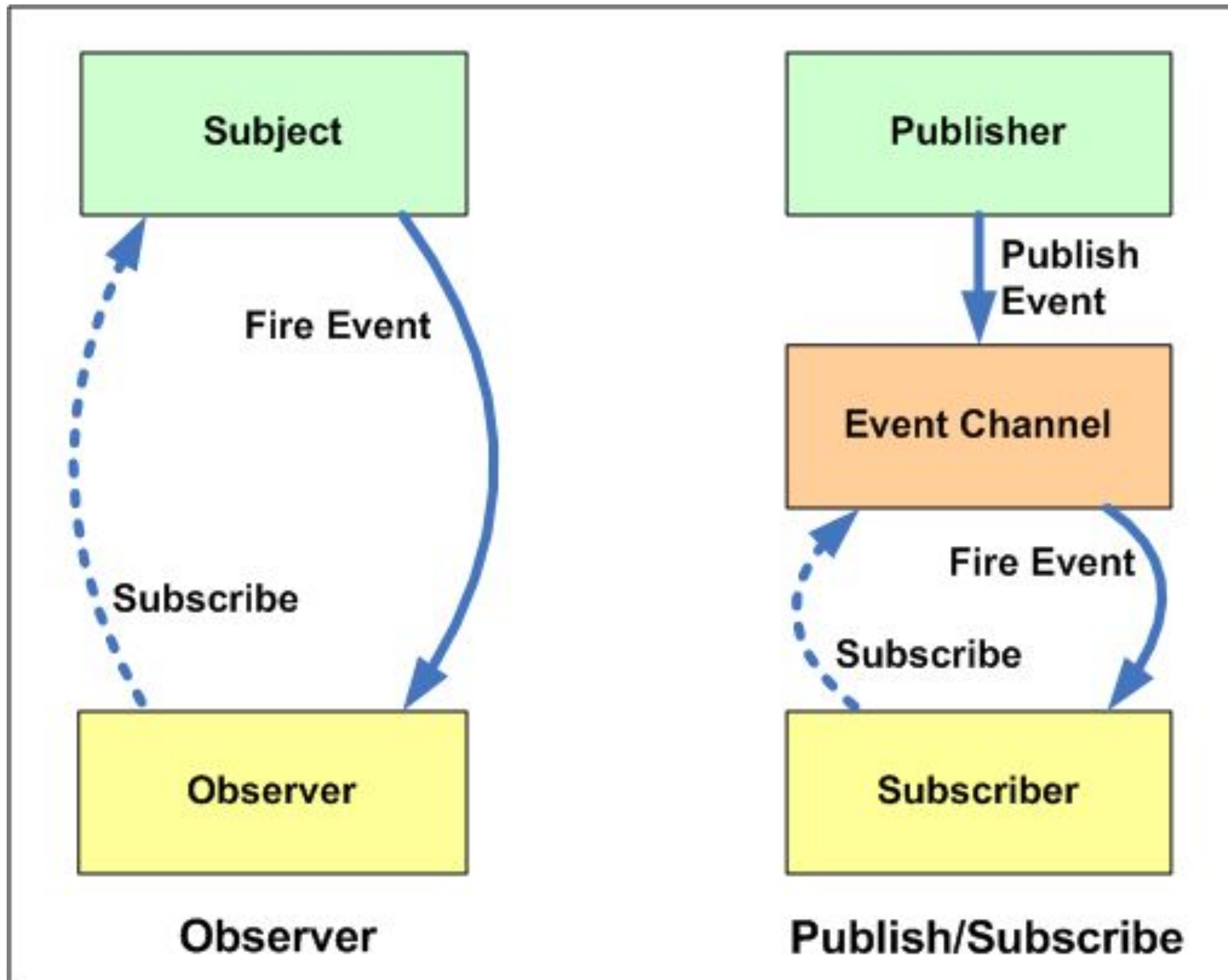
PUBLISH-SUBSCRIBE (1)

- The Publish–Subscribe pattern (Sub/Pub) is a messaging paradigm that decouples the components that produce messages (publishers) from those that consume them (subscribers).
- Instead of sending messages directly to specific receivers, publishers broadcast messages to a message broker or event bus, which then delivers them to all interested subscribers based on topics or event types.

PUBLISH-SUBSCRIBE (2)

- This design enables **loose coupling, scalability, and asynchronous communication** between system components.
- Publishers remain unaware of who receives the messages, and subscribers can join or leave dynamically without affecting the system's core logic.
- Common implementations include message brokers like Kafka, RabbitMQ, and Google Pub/Sub.
- The pattern is widely used in event-driven architectures, real-time notification systems, and microservices communication to ensure responsive and maintainable distributed systems.

OBSERVER VS PUBLISH-SUBSCRIBE



MODEL-VIEW-CONTROLLER

MODEL-VIEW-CONTROLLER (1)

- Model–View–Controller (MVC) is a software architectural pattern that separates an application into three interconnected components:
  - Model: Handles data, business logic, and rules of the application.
  - View: Manages the user interface and how information is presented to the user.
  - Controller: Acts as an intermediary, processing user input and coordinating communication between the Model and View.

MODEL-VIEW-CONTROLLER (2)

- The main goal of MVC is Separation of Concerns (SoC) — each part of the application **focuses on one specific responsibility**, improving maintainability and scalability.

MODEL-VIEW-CONTROLLER (3)

- **Model**
  - Represents the data and business logic of the system.
  - Responsible for data access, validation, and processing.
  - Independent from UI — the same Model can be reused with different interfaces (e.g., web, mobile).
- **Example**
  - A `Product` class that retrieves product information from a database and applies pricing logic.

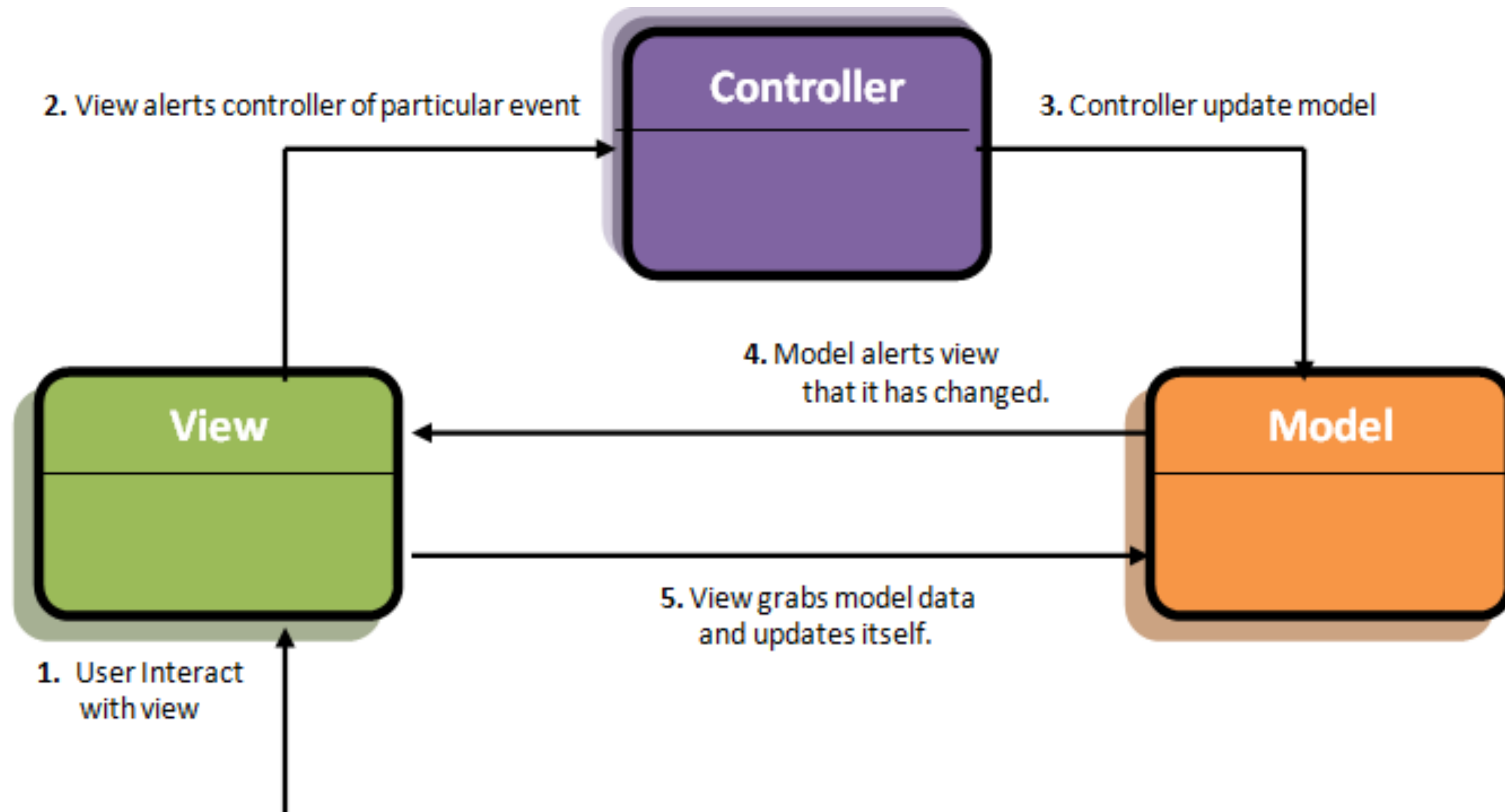
MODEL-VIEW-CONTROLLER (4)

- **View**
  - Displays data to the user and defines how information is presented.
  - Should not contain complex logic; it only renders what it receives from the Controller or Model.
  - Can be implemented using templates, UI components, or API responses (e.g., HTML, JSON, XML).

MODEL-VIEW-CONTROLLER (5)

- **Controller**
  - Serves as the bridge between user actions and system responses.
  - Handles user input, triggers changes in the Model, and updates the View accordingly.
  - Defines the flow of control within the application.

MODEL-VIEW-CONTROLLER (6)



ADVANTAGES OF MVC

Advantage	Description
Separation of Concerns	Each component handles a specific task — easier to manage and debug.
Maintainability	Code is modular, so updates or bug fixes can be done without affecting the whole system.
Reusability	Models and Views can be reused across different parts of the application.
Parallel Development	Multiple developers can work simultaneously — one on UI (View), another on business logic (Model), and another on routing (Controller).
Testability	Components can be tested independently, especially the Model and Controller.

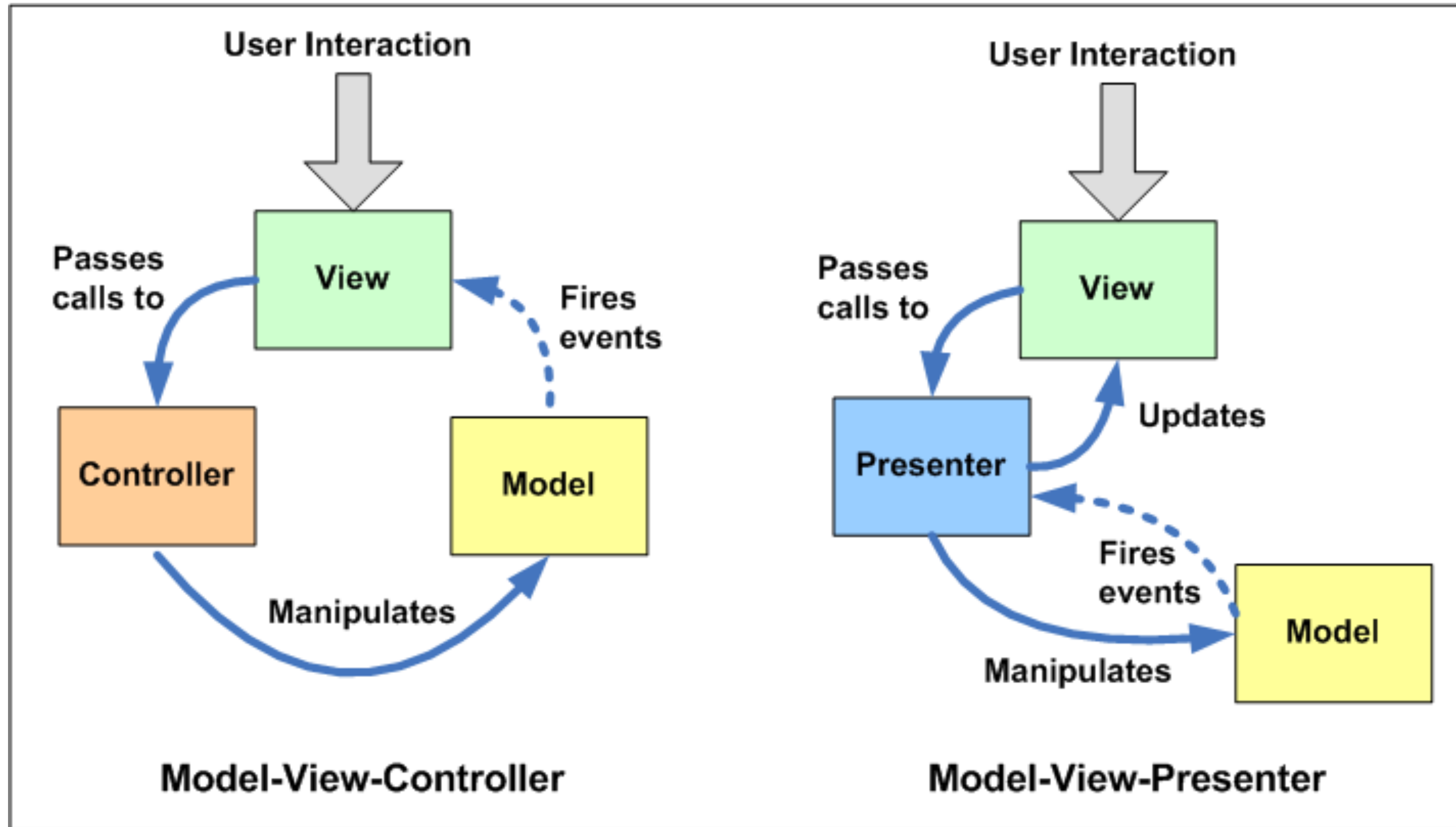
DISADVANTAGES OF MVC

Disadvantage	Description
Increased Complexity	Setting up MVC structure may be overkill for small applications.
Learning Curve	Developers must understand how the components interact properly.
Controller Overload	Controllers can become bloated if too much logic is handled there (known as "Fat Controller").
Frequent Updates Across Layers	A change in one layer might require updates in others to maintain consistency.
Tight Coupling Between Controller & View	In poorly implemented MVCs, these two layers may become too interdependent.

MODEL-VIEW-PRESENTER (MVP)

- The MVP pattern cleanly divides an application into three interconnected components, with the Presenter acting as the central intelligence.
- Presenter
  - Responsibility: Acts as the **"middle-man"** or delegate **between the Model and the View**. It contains the UI presentation logic.
  - Key Principle: The Presenter receives input from the View, interacts with the Model to retrieve or update data, applies any necessary formatting or presentation logic, and then explicitly tells the View what to display. It holds a reference to the View's interface, ensuring true decoupling.

MODEL-VIEW-PRESENTER (MVP)



MVC VS. MVP (1)

Feature	MVC	MVP
Intermediary	Controller	Presenter
View's Role	Active View: Can contain some logic and interacts directly with the Model to read data (often via Observer Pattern).	Passive View: Minimal logic; delegates all user actions to the Presenter and only displays data provided by the Presenter.
View-Model Link	Loose Coupling: The View may directly observe the Model.	Decoupled: The View is completely separated from the Model; the Presenter is the only link.
Data Flow	Controller receives user input → Controller updates Model → Model notifies View (via Observer) → View updates itself.	View receives user input → View calls Presenter → Presenter updates Model → Presenter gets data from Model →Presenter updates View (via an interface).

MVC VS. MVP (2)

Feature	MVC	MVP
Testability	Harder to test the View's display logic as it often has a tight coupling with a specific UI framework.	Highly Testable: The Presenter can be unit-tested independently of the View, as it communicates with the View through an interface.
Relationship	Often one Controller can manage multiple Views (e.g., in web frameworks).	Typically a one-to-one mapping between a View and a Presenter.

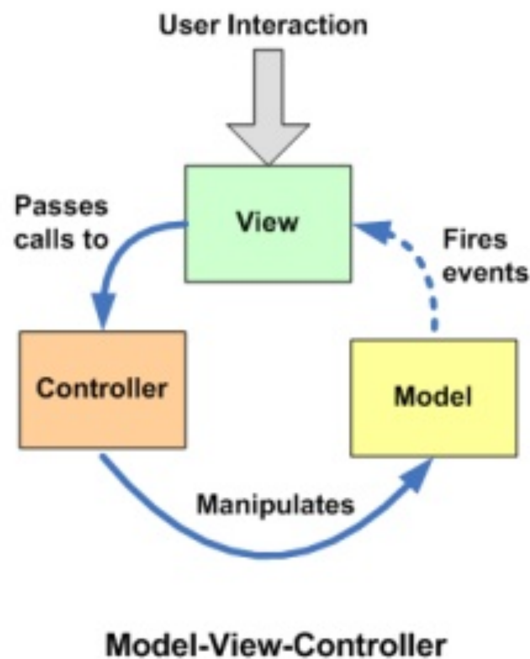
MODEL-VIEW-VIEW-MODEL (MVVM)

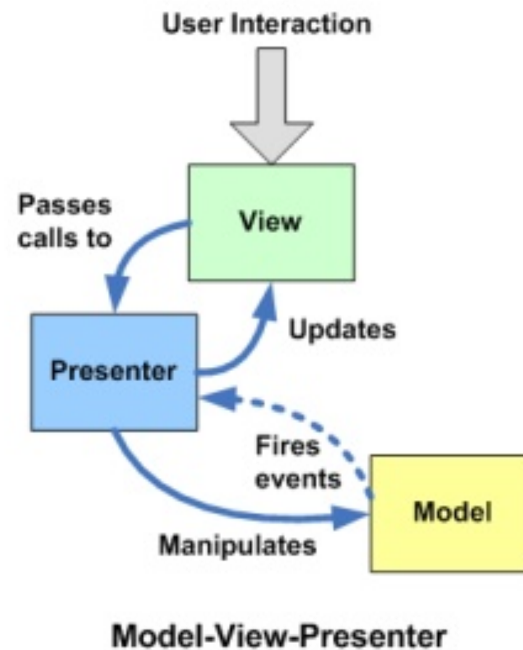
- MVVM (Model–View–ViewModel) is a software architectural pattern that extends the ideas of MVC and MVP to achieve a stronger separation of concerns and better support for modern data binding and reactive UI frameworks.
- It was originally introduced by Microsoft for WPF (Windows Presentation Foundation) but is now widely adopted in frameworks such as Angular, Vue.js, React (with hooks/state), SwiftUI, and Jetpack Compose.

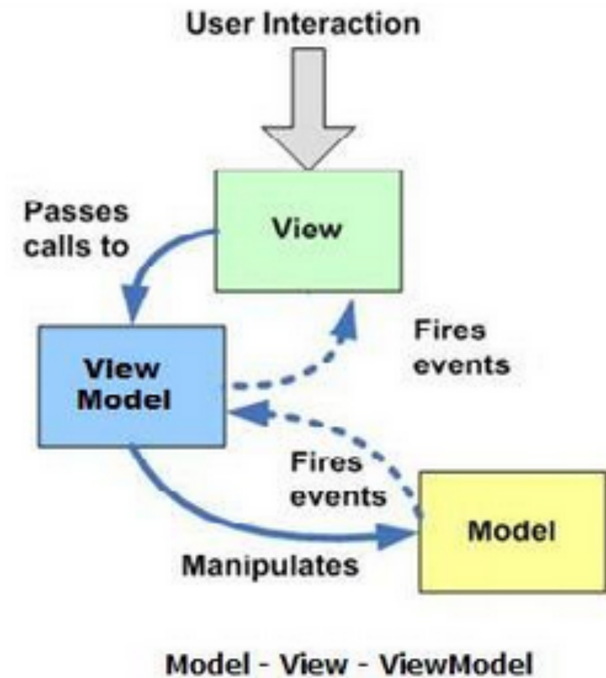
VIEWMODEL

- The ViewModel acts as an intermediary between the View and the Model.
- It exposes data from the Model as observable properties that the View can bind to directly.
- It also handles UI logic, such as formatting data, managing commands, and controlling states (e.g., loading, error, success).
- Unlike a Controller or Presenter, the ViewModel does not reference the View directly — this decoupling enables better testability and reusability.

MODEL-VIEW-VIEW-MODEL (MVVM)







OTHER ARCHITECTURAL PATTERNS

- Shared-Data Pattern
- Center of Competence (CoC) Pattern
- Big Ball of Mud Pattern
- Etc.

CONCLUSION

- Architecture of a software system is its structures comprising of elements, their external properties, and relationships.
- Architecture is a high level design.
- Architecture can be analyzed for various non-functional attributes like performance, reliability, security, etc.

DESIGN PATTERNS

GRASP PATTERN

- GRASP (General Responsibility Assignment Software Patterns) is an acronym created by Craig Larman to encompass nine object-oriented design principles related to creating responsibilities for classes
- These principles can also be viewed as design patterns and offer benefits similar to the classic "Gang of Four" patterns
- GRASP is an attempt to document what expert designers probably know intuitively
- All nine GRASP patterns will be presented and briefly discussed

GRASP PATTERN

- Nine GRASP patterns:
  - Creator
  - Information Expert
  - Low Coupling
  - Controller
  - High Cohesion
  - Indirection
  - Polymorphism
  - Protected Variations
  - Pure Fabrication

RESPONSIBILITY-DRIVEN DESIGN

- GRASP patterns are used to assign responsibility to objects
- As such, their use results in a Responsibility-Driven Design (RDD) for Object Orientation (OO) – Contrast to (the more traditional) Data-Driven Design
- With this point of view, assigning responsibilities to objects is a large part of basic object design

DESIGN PATTERN

- Software design patterns were launched as a concept in 1987 by Kent Beck and Ward Cunningham, based upon Christopher Alexander's application in (building) architecture
- Core definition: a named description of a problem and a corresponding reusable solution
- Ideally, the pattern advises on when it should be used and the typical trade-offs
- The most famous design patterns are the 23 described by the "Gang of Four" (GoF) book in 1993

DESIGN PATTERN ADVANTAGES

- Both the GoF patterns and GRASP patterns have notable benefits:
  - Simplifying: provides a named, generally understood building block
    - Facilitates communication
    - Aids thinking about the design
  - Accelerates learning to not have to develop concepts from scratch

GRASP VS. GOF

- GRASP patterns are in a way even more fundamental than the GoF patterns
  - GRASP patterns are equally well referred to as principles, while the GoF patterns are rarely referred to as such
- While the naming of both types of patterns is important, it's less important for the GRASP patterns
  - The concepts are truly what are important

CREATOR

- Who creates an Object? Or who should create a new instance of some class?
- "Container" object creates "contained" objects.
- Decide who can be creator based on the objects association and their interaction.

EXAMPLE OF CREATOR

- Consider VideoStore and Video in that store.
- VideoStore has an **aggregation** association with Video. i.e, VideoStore is the container and the Video is the contained object.
- So, we can instantiate video object in VideoStore class

INFORMATION EXPERT

- Given an object O, which responsibilities can be assigned to O?
- Expert principle says – assign those responsibilities to O for which O has the **information to fulfill that responsibility**.
- They have all the information needed to perform operations, or in some cases they collaborate with others to fulfill their responsibilities.
  - This generally is key to loose coupling and high cohesion.

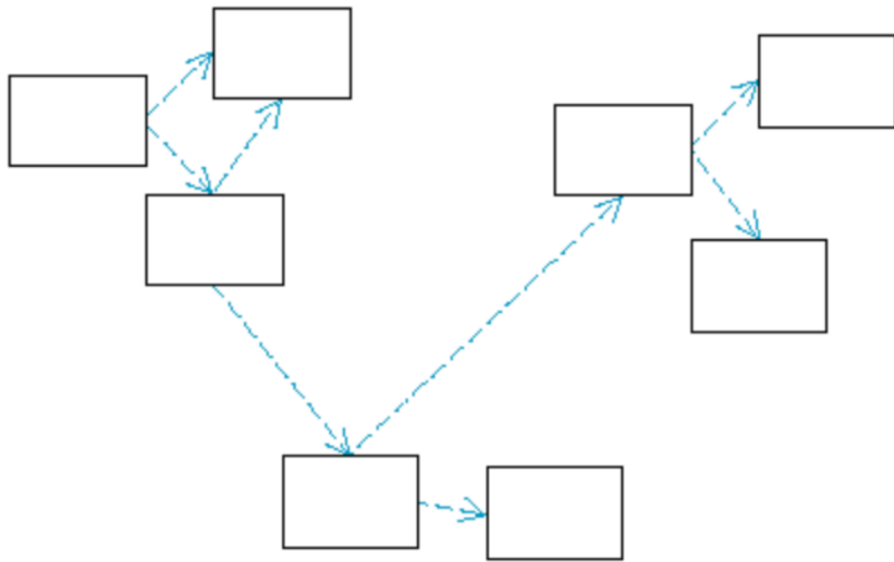
EXAMPLE OF INFORMATION EXPERT

- Assume we need to get all the videos of a VideoStore.
- Since VideoStore knows about all the videos, we can assign this responsibility of giving all the videos can be assigned to VideoStore class.
- VideoStore is the information expert.

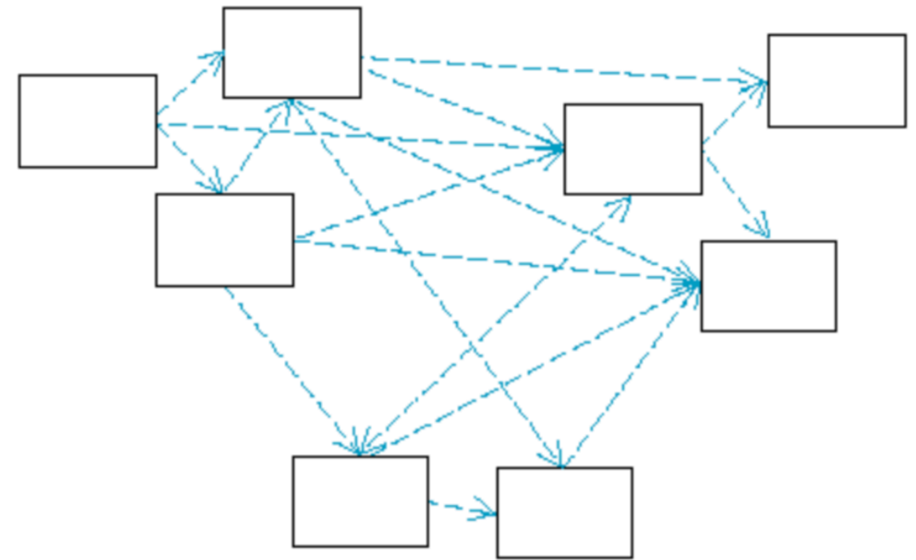
LOW COUPLING (1)

- How strongly the objects are connected to each other? Coupling – object depending on other object.
- When depended upon element changes, it affects the dependent also.
- Low Coupling – How can we reduce the impact of change in depended upon elements on dependent elements.
- Prefer low coupling – assign responsibilities so that coupling remain low.
- Minimizes the dependency hence making system maintainable, efficient and code reusable

LOW COUPLING (2)



Low Coupling



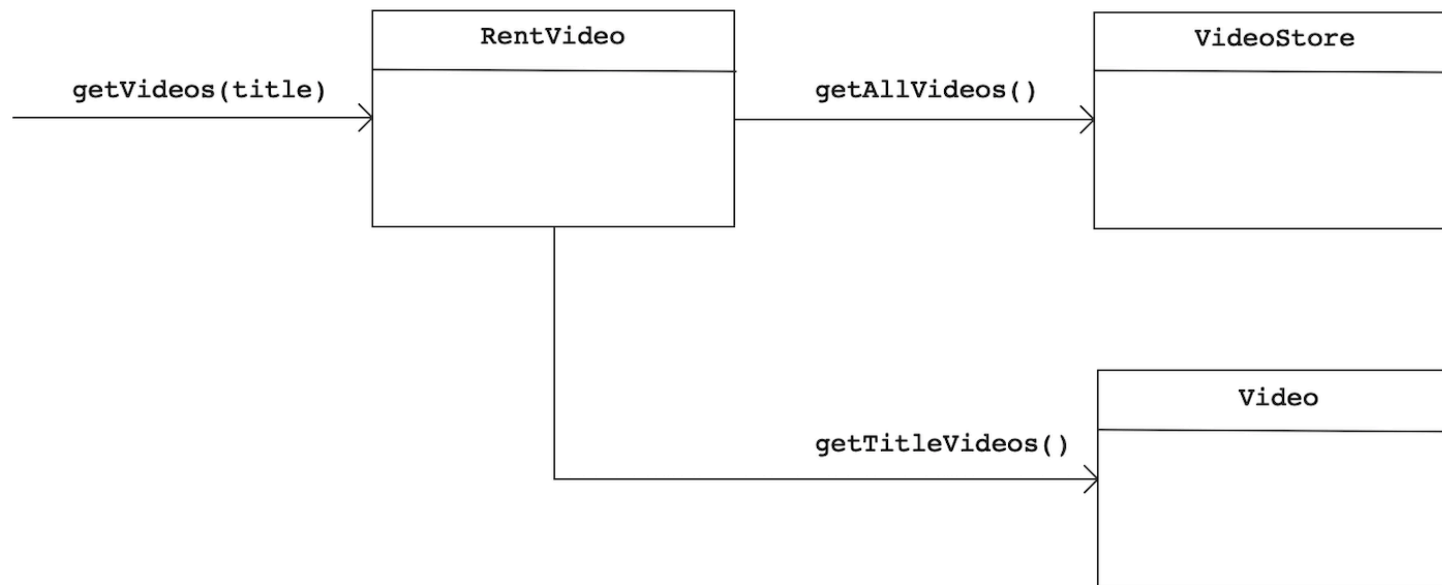
High Coupling

LOW COUPLING (3)

- Two elements are coupled, if
  - One element has aggregation/composition association with another element.
  - One element implements/extends other element.

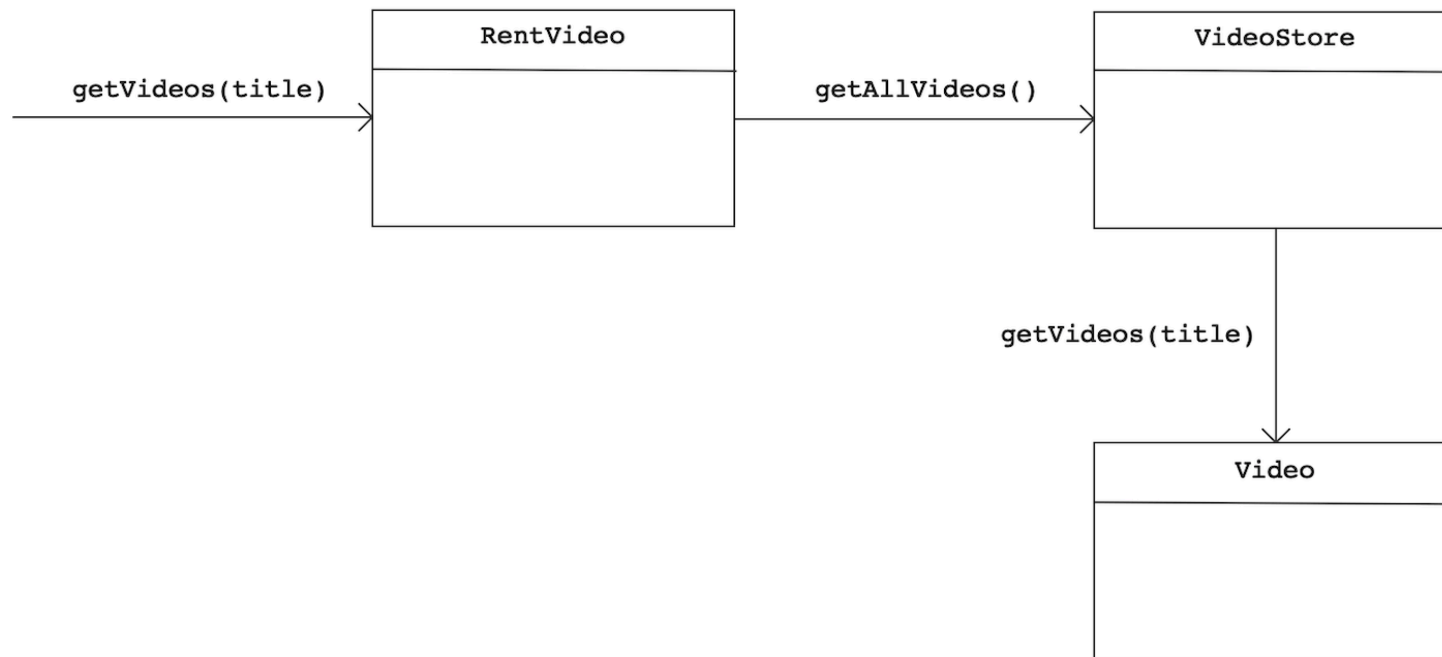
EXAMPLE OF POOR COUPLING

- Here class RentVideo knows about both VideoStore and Video objects. RentVideo is depending on both the classes.



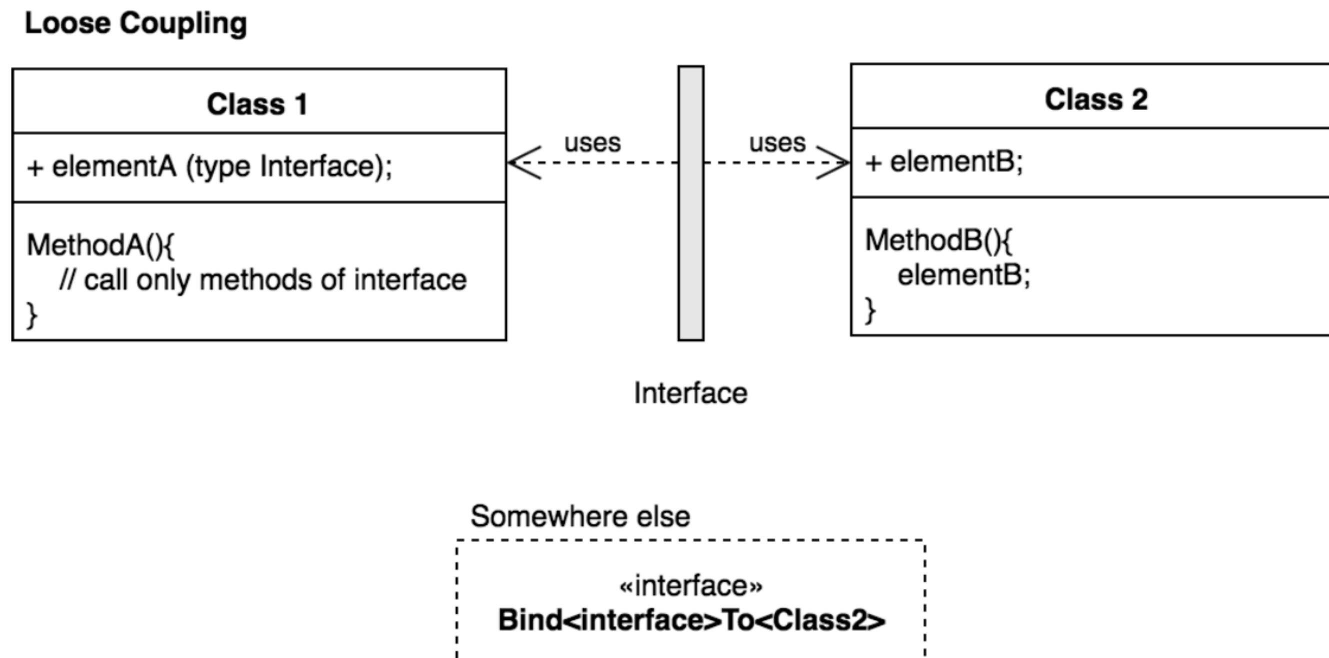
EXAMPLE OF LOW COUPLING

- VideoStore and Video class are coupled, and Rent is coupled with VideoStore. Thus providing low coupling.



LOOSE COUPLING

- Loose coupling is a method of interconnecting the components in a system or network so that those components, depend on each other to the least extent practically possible.



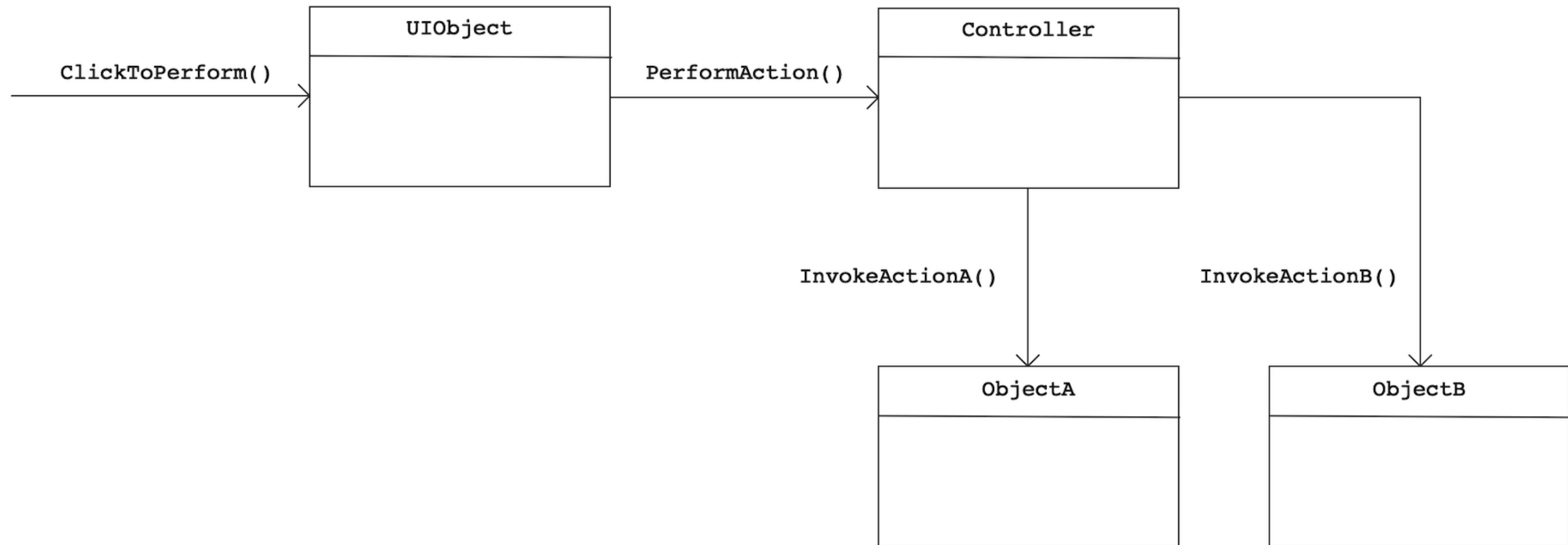
CONTROLLER (1)

- Deals with how to delegate the request from the UI layer objects to domain layer objects.
- When a request comes from UI layer object, Controller pattern helps us in determining what is that first object that receive the message from the UI layer objects.
- This object is called controller object which receives request from UI layer object and then controls coordinates with other object of the domain layer to fulfill the request.
- It delegates the work to other class and coordinates the overall activity.

CONTROLLER (2)

- We can make an object as Controller, if
  - Object represents the overall system (facade controller)
  - Object represent a use case, handling a sequence of operations (session controller).
- Benefit
  - Can reuse this controller class.
  - Can use to maintain the state of the use case.
  - Can control the sequence of the activities

EXAMPLE OF CONTROLLER



BOATED CONTROLLERS

- Controller class is called bloated, if …
  - The class is overloaded with too many responsibilities.
    - Solution – Add more controllers
  - Controller class also performing many tasks instead of delegating to other class.
    - Solution – controller class has to delegate things to others.

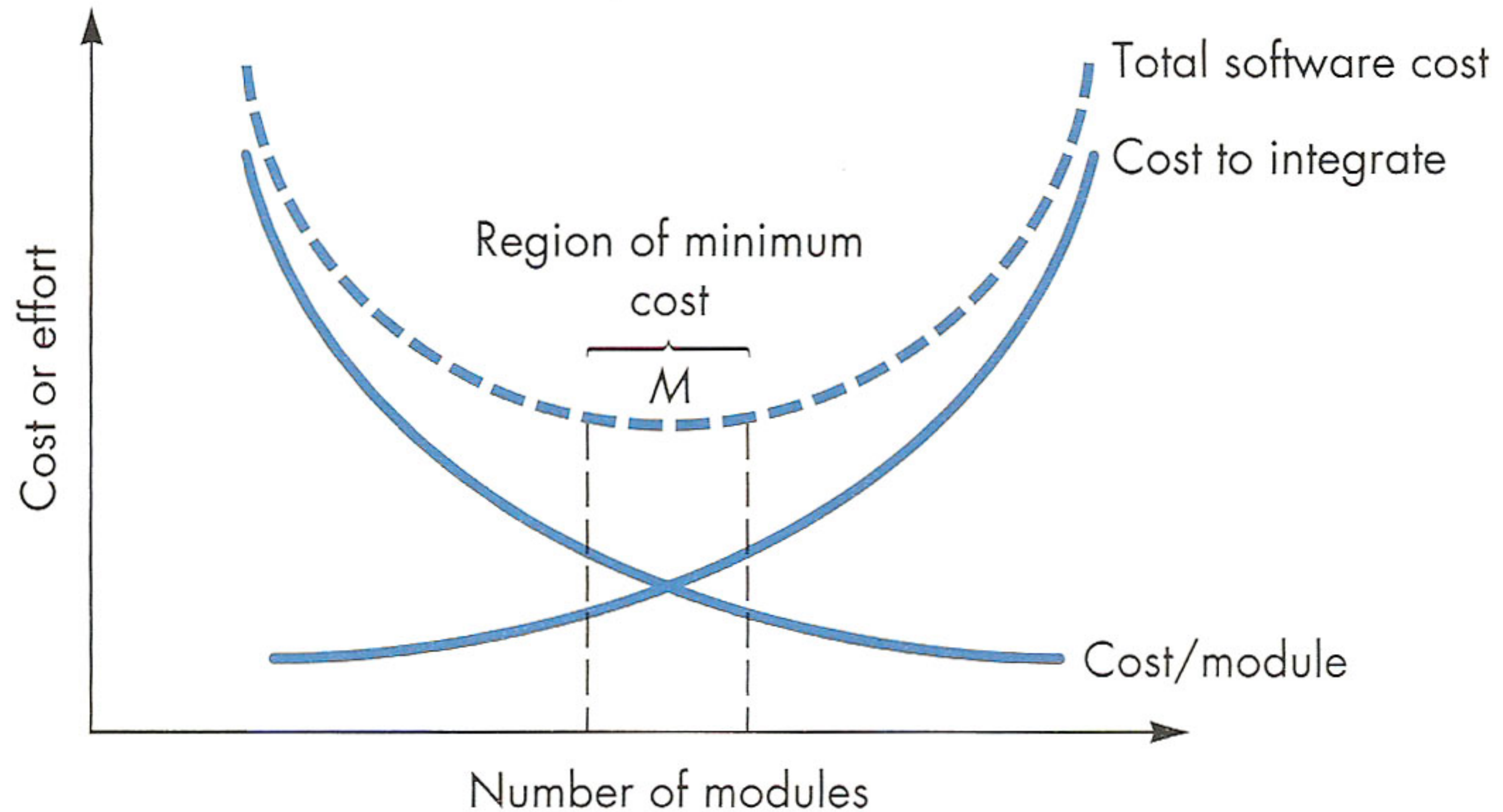
HIGH COHESION

- How are the operations of any element are functionally related?
- Related responsibilities in to one manageable unit. Prefer high cohesion
- Clearly defines the purpose of the element
- Benefits
  - Easily understandable and maintainable.
  - Code reuse
  - Low coupling

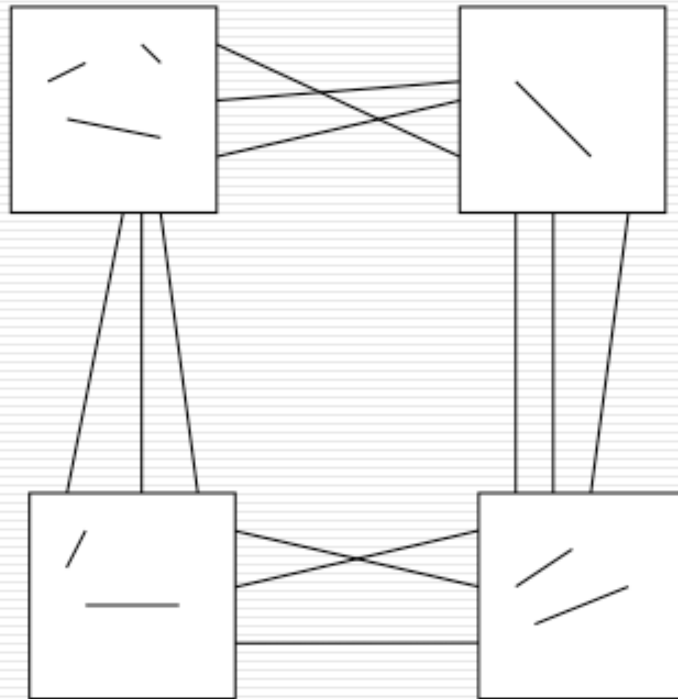
MODULARITY

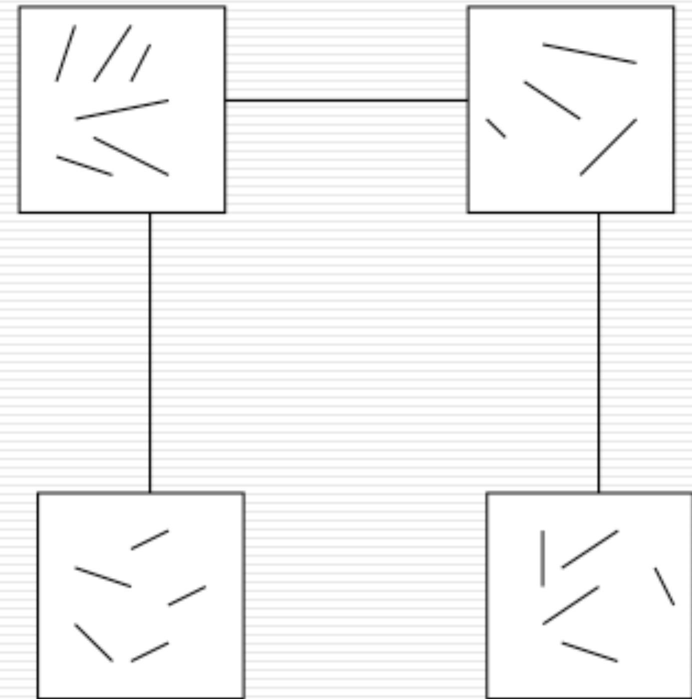
- Software architecture and design patterns embody modularity, that is software is divided into separately named and addressable components, sometimes called modules, that are integrated to satisfy problem requirements.

MODULARITY AND SOFTWARE COST

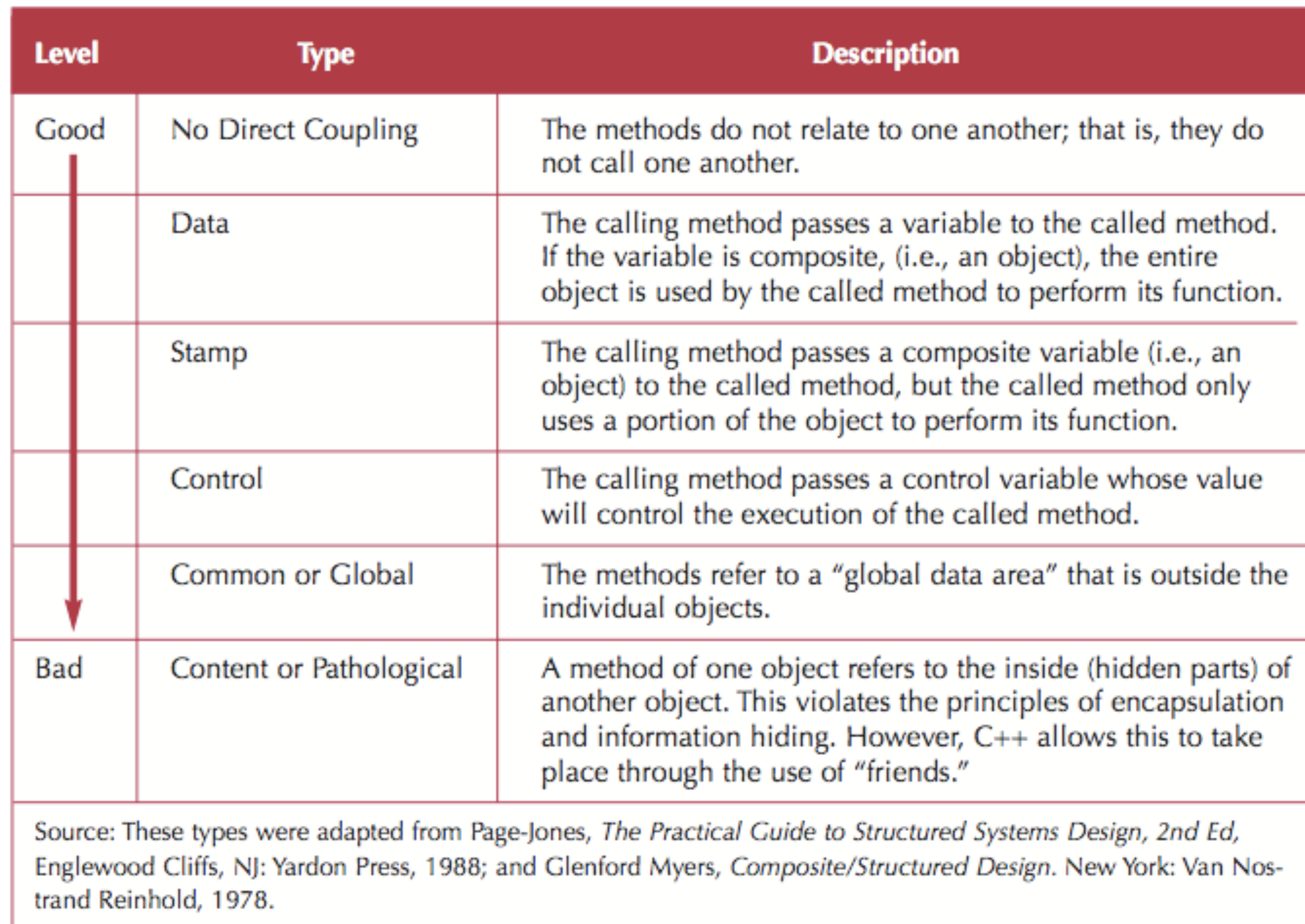


COUPLING AND COHESION





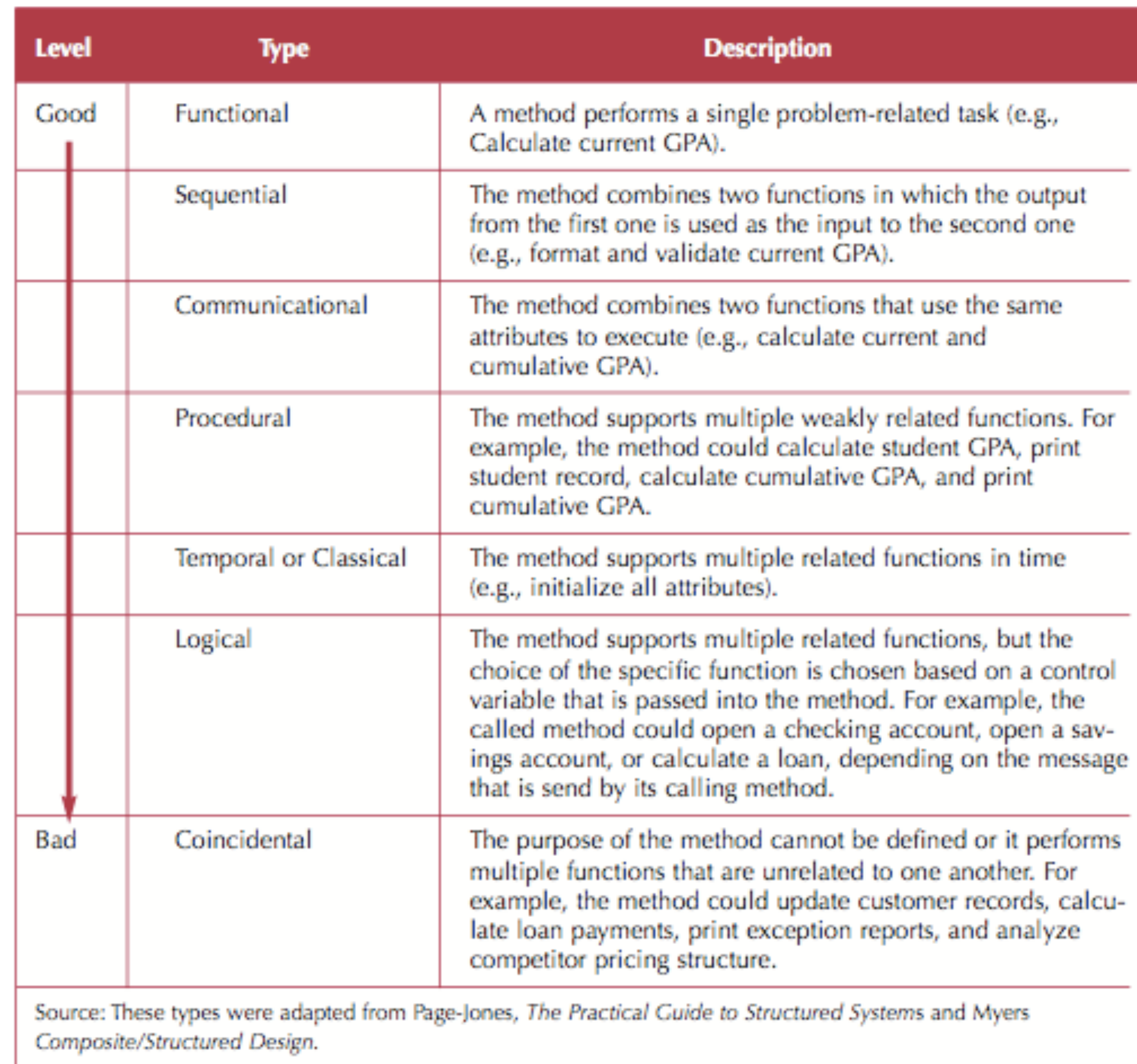
TYPE OF INTERACTION COUPLING



Level	Type	Description
Good	No Direct Coupling	The methods do not relate to one another; that is, they do not call one another.
	Data	The calling method passes a variable to the called method. If the variable is composite, (i.e., an object), the entire object is used by the called method to perform its function.
	Stamp	The calling method passes a composite variable (i.e., an object) to the called method, but the called method only uses a portion of the object to perform its function.
	Control	The calling method passes a control variable whose value will control the execution of the called method.
	Common or Global	The methods refer to a "global data area" that is outside the individual objects.
Bad	Content or Pathological	A method of one object refers to the inside (hidden parts) of another object. This violates the principles of encapsulation and information hiding. However, C++ allows this to take place through the use of "friends."

Source: These types were adapted from Page-Jones, *The Practical Guide to Structured Systems Design, 2nd Ed,* Englewood Cliffs, NJ: Yardon Press, 1988; and Glenford Myers, *Composite/Structured Design*. New York: Van Nostrand Reinhold, 1978.

TYPE OF METHOD COHESION

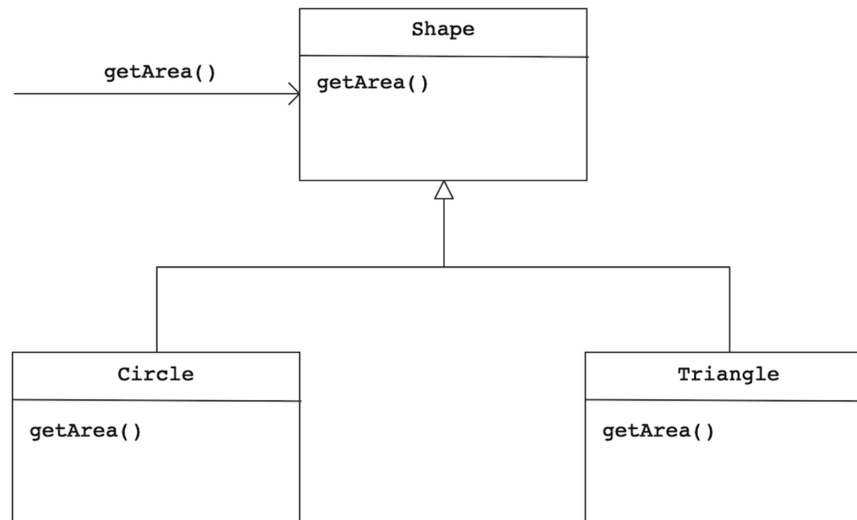


Level	Type	Description
Good	Functional	A method performs a single problem-related task (e.g., Calculate current GPA).
	Sequential	The method combines two functions in which the output from the first one is used as the input to the second one (e.g., format and validate current GPA).
	Communicational	The method combines two functions that use the same attributes to execute (e.g., calculate current and cumulative GPA).
	Procedural	The method supports multiple weakly related functions. For example, the method could calculate student GPA, print student record, calculate cumulative GPA, and print cumulative GPA.
	Temporal or Classical	The method supports multiple related functions in time (e.g., initialize all attributes).
	Logical	The method supports multiple related functions, but the choice of the specific function is chosen based on a control variable that is passed into the method. For example, the called method could open a checking account, open a savings account, or calculate a loan, depending on the message that is send by its calling method.
Bad	Coincidental	The purpose of the method cannot be defined or it performs multiple functions that are unrelated to one another. For example, the method could update customer records, calculate loan payments, print exception reports, and analyze competitor pricing structure.

Source: These types were adapted from Page-Jones, *The Practical Guide to Structured Systems* and Myers *Composite/Structured Design*.

POLYMORPHISM

- How to handle related but varying elements based on element type?
- Polymorphism guides us in deciding which object is responsible for handling those varying elements.
- Benefits: handling new variations will become easy.



PURE FABRICATION

- Fabricated class/artificial class – assign set of related responsibilities that doesn't represent any domain object.
- Provides a highly cohesive set of activities.
- Behavioral decomposed – implements some algorithm.
- Examples: Adapter, Strategy (GoF)
- Benefits: High cohesion, low coupling and can reuse this class.

EXAMPLE OF PURE FABRICATION

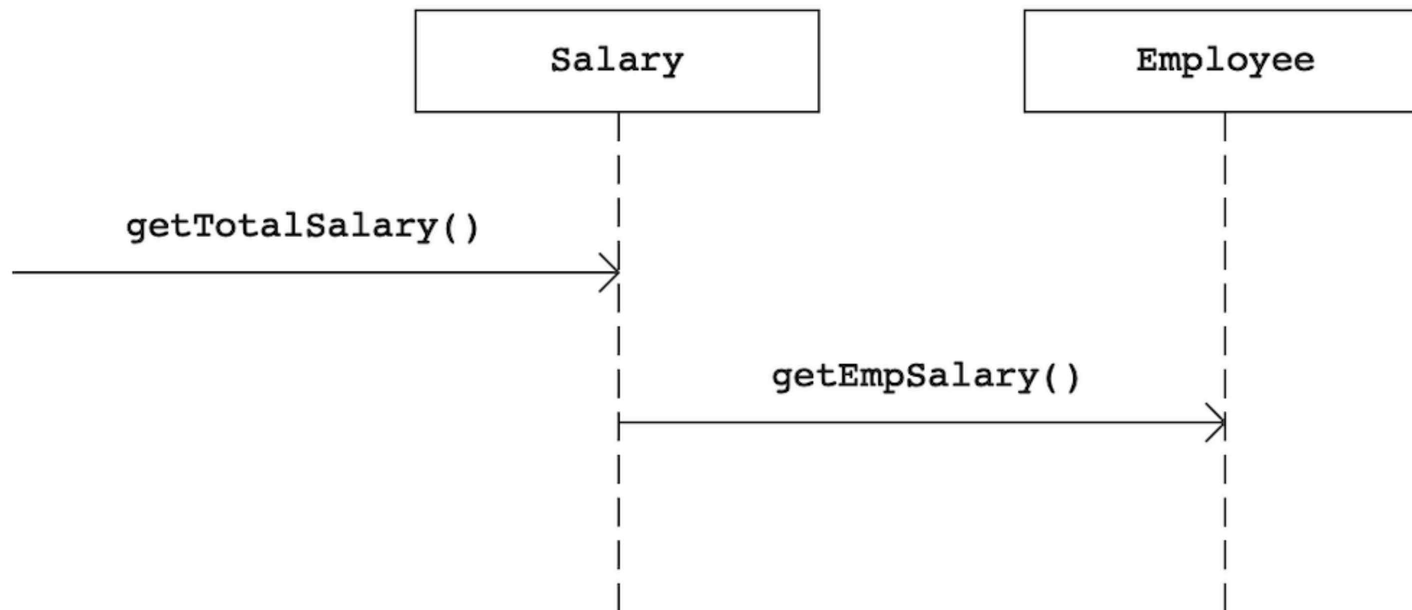
- Suppose we Shape class, if we must store the shape data in a database.
- If we put this responsibility in Shape class, there will be many database related operations thus making Shape in-cohesive.
- So, create a fabricated class DBStore which is responsible to perform all database operations.
- Similarly logInterface which is responsible for logging information is also a good example for Pure Fabrication.

INDIRECTION

- How can we avoid a direct coupling between two or more elements.
- Indirection introduces an intermediate unit to communicate between the other units, so that the other units are not directly coupled.
- Benefits: low coupling
- Example: Adapter, Facade, Observer (GoF)

EXAMPLE OF INDIRECTION

- Employee acts as level of indirection

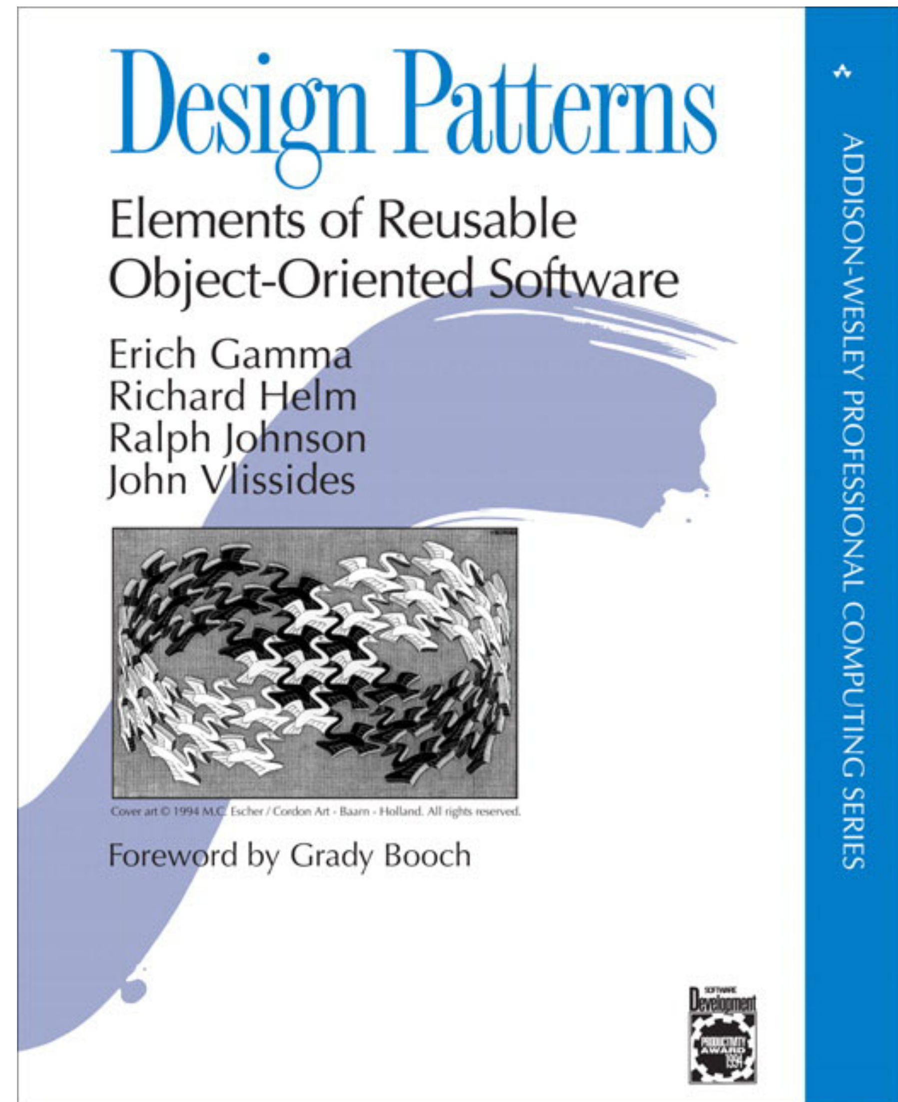


PROTECTED VARIATION

- How to avoid impact of variations of some elements on the other elements.
- It provides a well defined interface so that the there will be no affect on other units.
- Provides flexibility and protection from variations.
- Provides more structured design.
- Example: polymorphism, data encapsulation, interfaces

GOF DESIGN PATTERNS

- Over 20 years ago the iconic computer science book "Design Patterns: Elements of Reusable Object-Oriented Software" was first published.
- The four authors of the book: Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, have since been dubbed "The Gang of Four".
- In technology circles, you'll often see this nicknamed shorted to GoF. Even though the GoF Design Patterns book was published over 20 years ago, it still continues to be an Amazon best seller.



GOF DESIGN PATTERN CATALOGUE