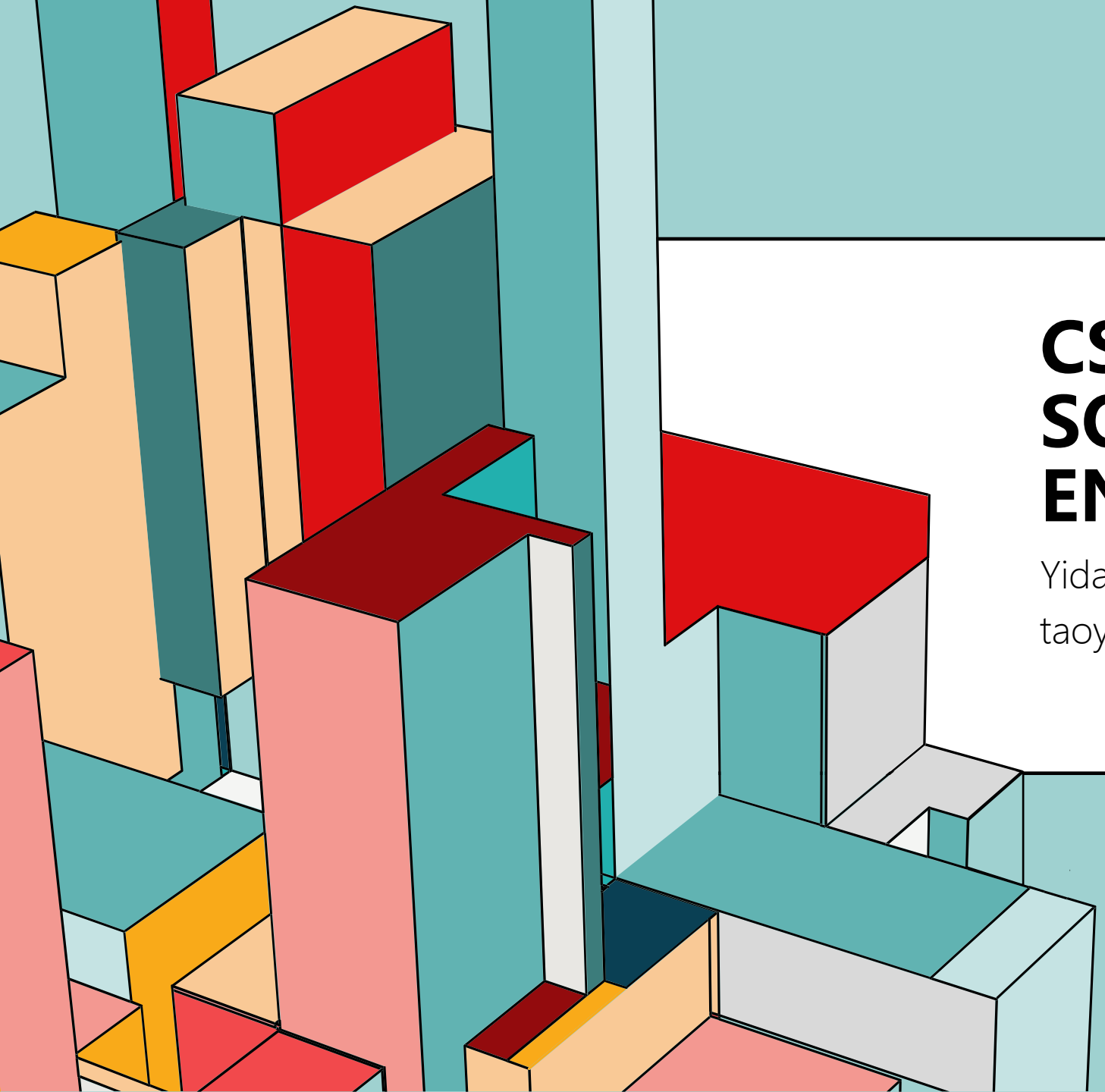


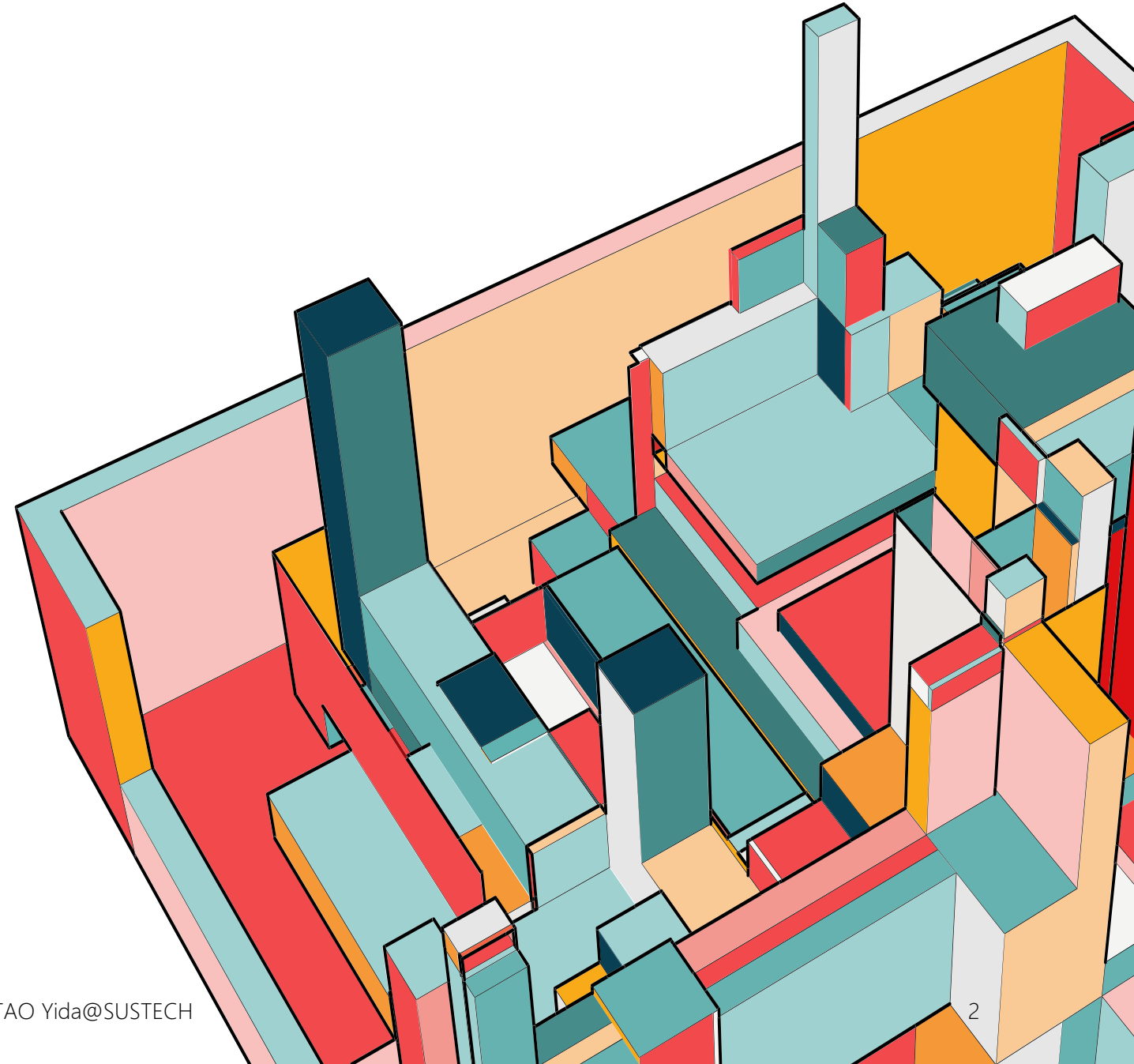
CS304 SOFTWARE ENGINEERING

Yida Tao

taoyd@sustech.edu.cn

LECTURE 5

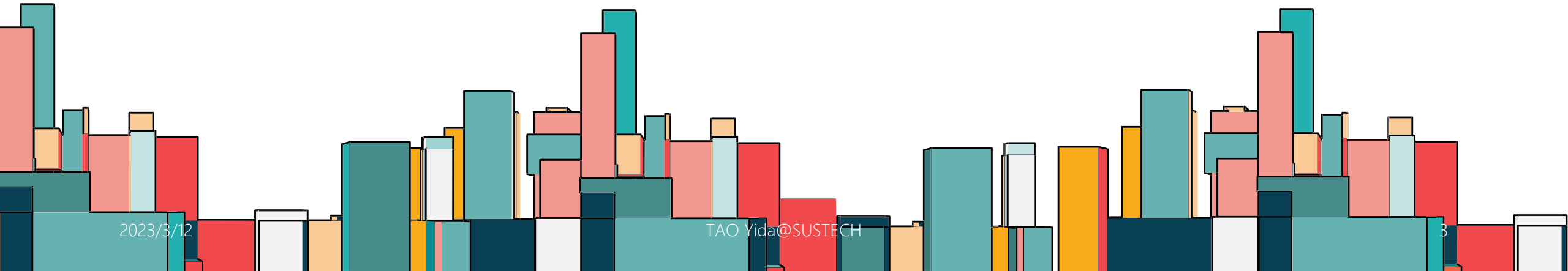
- Software Design Overview
- Design model
- Design concepts



WHAT IS DESIGN?

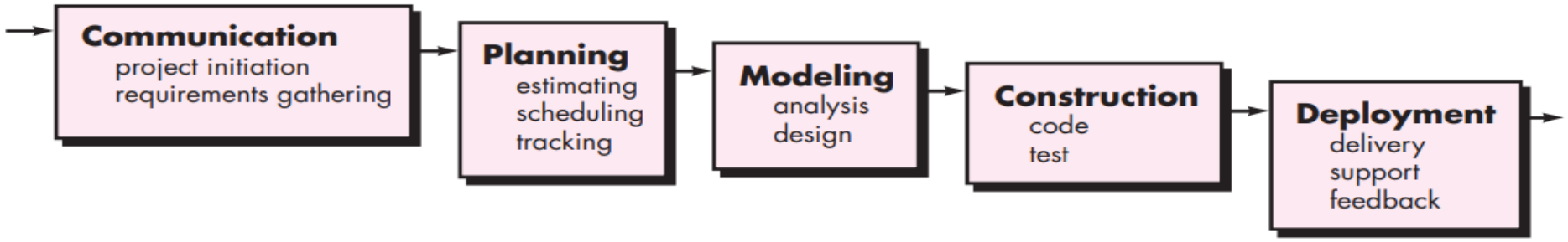
It’s where you stand with a foot in two worlds—the world of technology and the world of people and human purposes—and you try to bring the two together.

- Mitch Kapor, “software design manifesto”



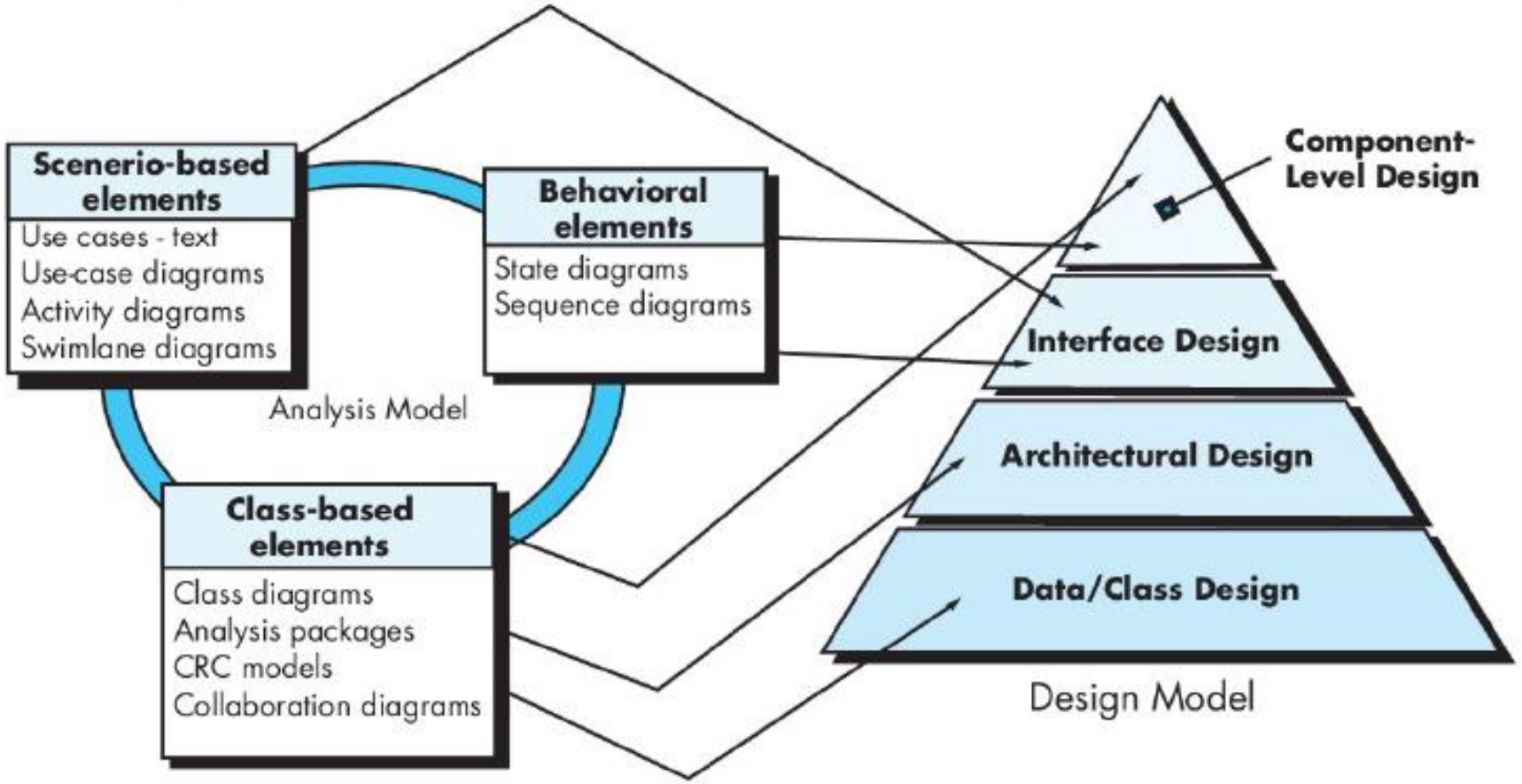
DESIGN IN SOFTWARE ENGINEERING

- Once software requirement is analyzed, software design begins
- Software design is the last software engineering action within the modeling activity and sets the stage for construction
- Software design is the bridge between software requirement and software implementation



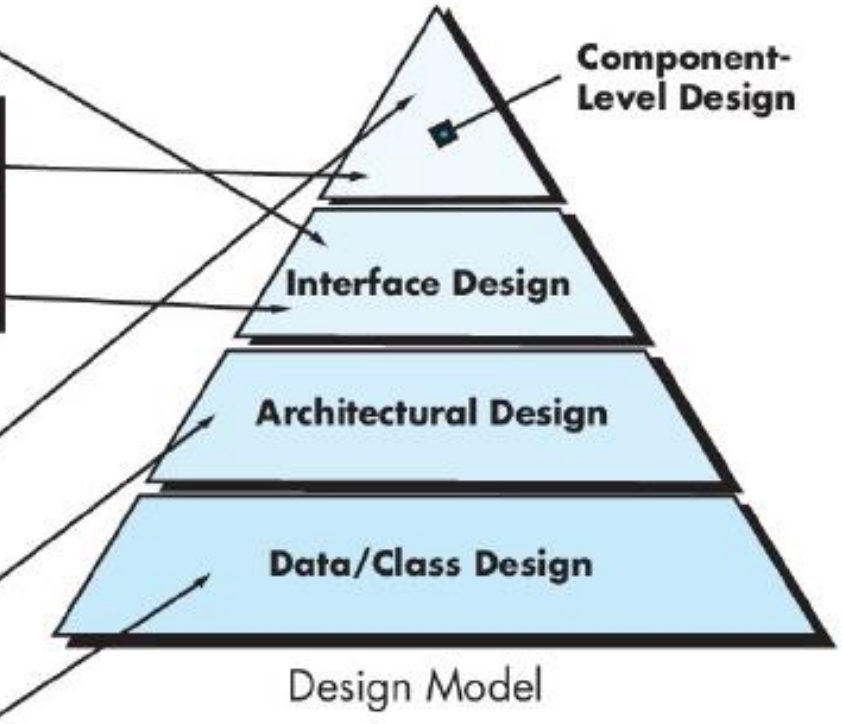
FROM THE REQUIREMENTS MODEL TO THE DESIGN MODEL

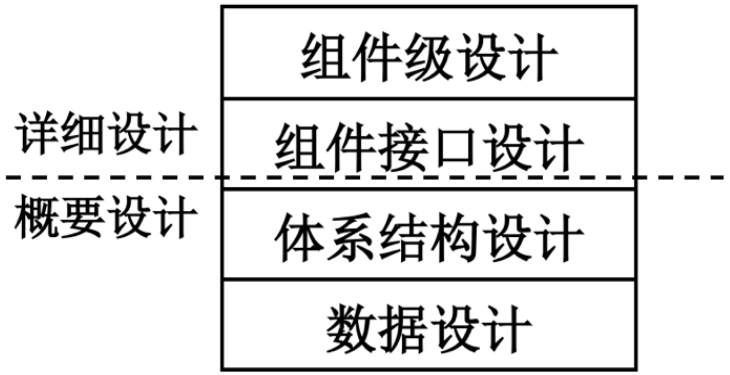
Each elements of the requirements model provides information that is necessary to create the four design models required for a complete specification of design.



FROM THE REQUIREMENTS MODEL TO THE DESIGN MODEL

Each elements of the requirements model provides information that is necessary to create the four design models required for a complete specification of design.





A design model that encompasses data, architectural, interface, component-level, and deployment representations is the primary work product that is produced during software design.

DATA DESIGN

- Data design creates a model of data and/or information at a high level of abstraction (data structures and relations from uses' view)
- The data model is then refined to more implementation-specific representations that can be processed by the computer-based system

DATA DESIGN IS IMPORTANT

- At the program component level, the design of **data structures** and associated algorithms is essential to create high-quality application
- At the application level, the translation of a data model into a **database** is pivotal to achieving the business objectives of a system
- At the business level, the collection of information stored in database enables data mining or knowledge discovery that affects the business success

ARCHITECTURAL DESIGN

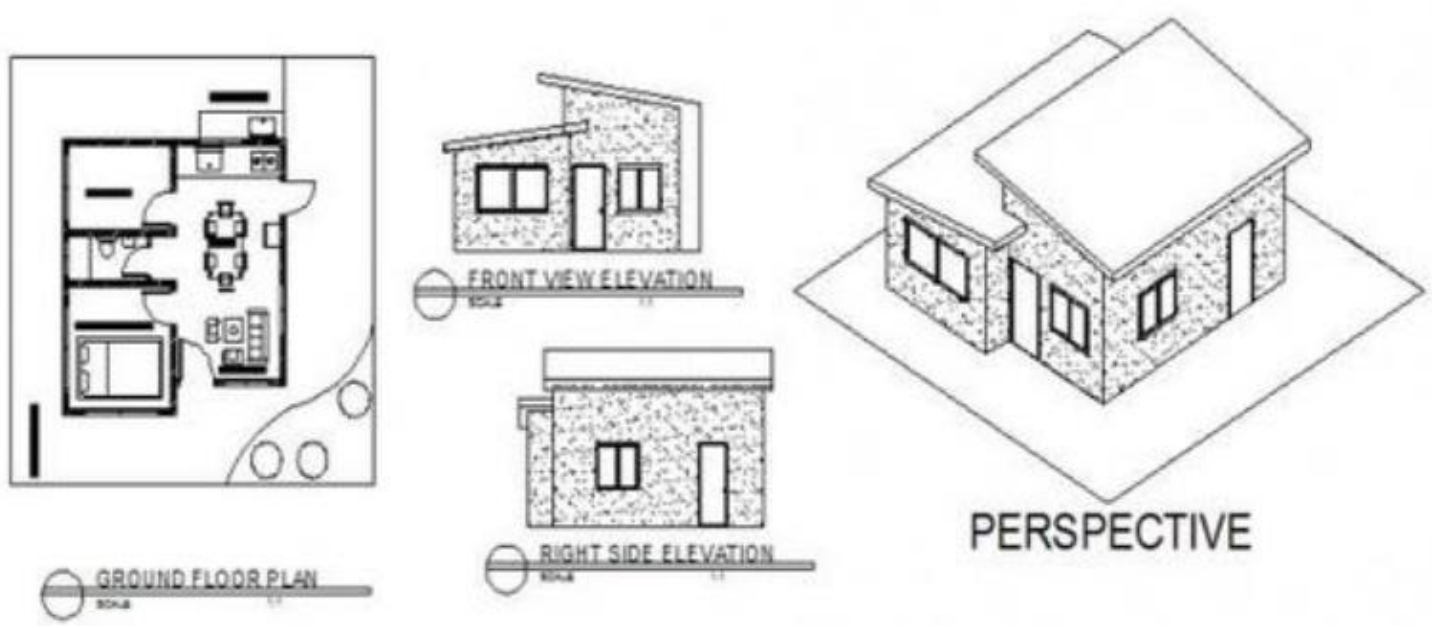
- The architectural design for software is the equivalent to the design plan of a house
  - The house design plan gives us an overall view of the house
  - The architectural design gives us an overall view of the software

- The architectural design model is derived from
  - Domain knowledge or information about the software to be built
  - Requirements model, e.g., class diagrams
  - Architectural styles and patterns

DESCRIBING SOFTWARE ARCHITECTURE

- Software architectural design is highly abstract and highly complicated.
- A single description won't be sufficient

To design a house, we need different types of house plans from different views



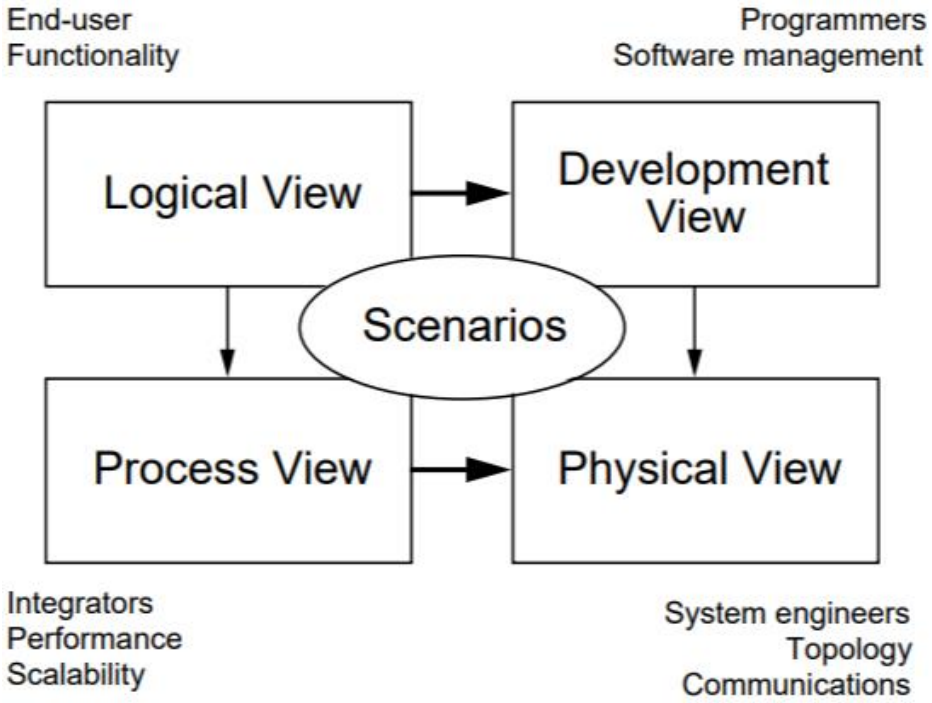
https://in.pinterest.com/pin/306315212155705228/

DESCRIBING SOFTWARE ARCHITECTURE

- Software architectural design is highly abstract and highly complicated.
- A single description won't be sufficient

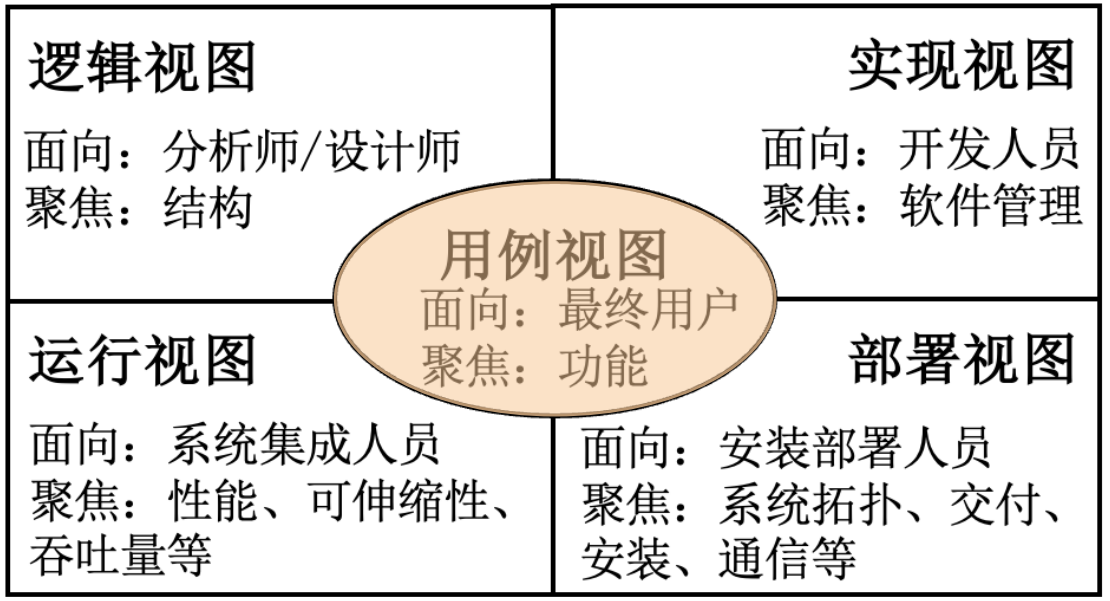
Similarly, to design a software architecture, we need different views that describe the concerns of different stakeholders

The "4+1" architectural view model proposed by Philippe Kruchten

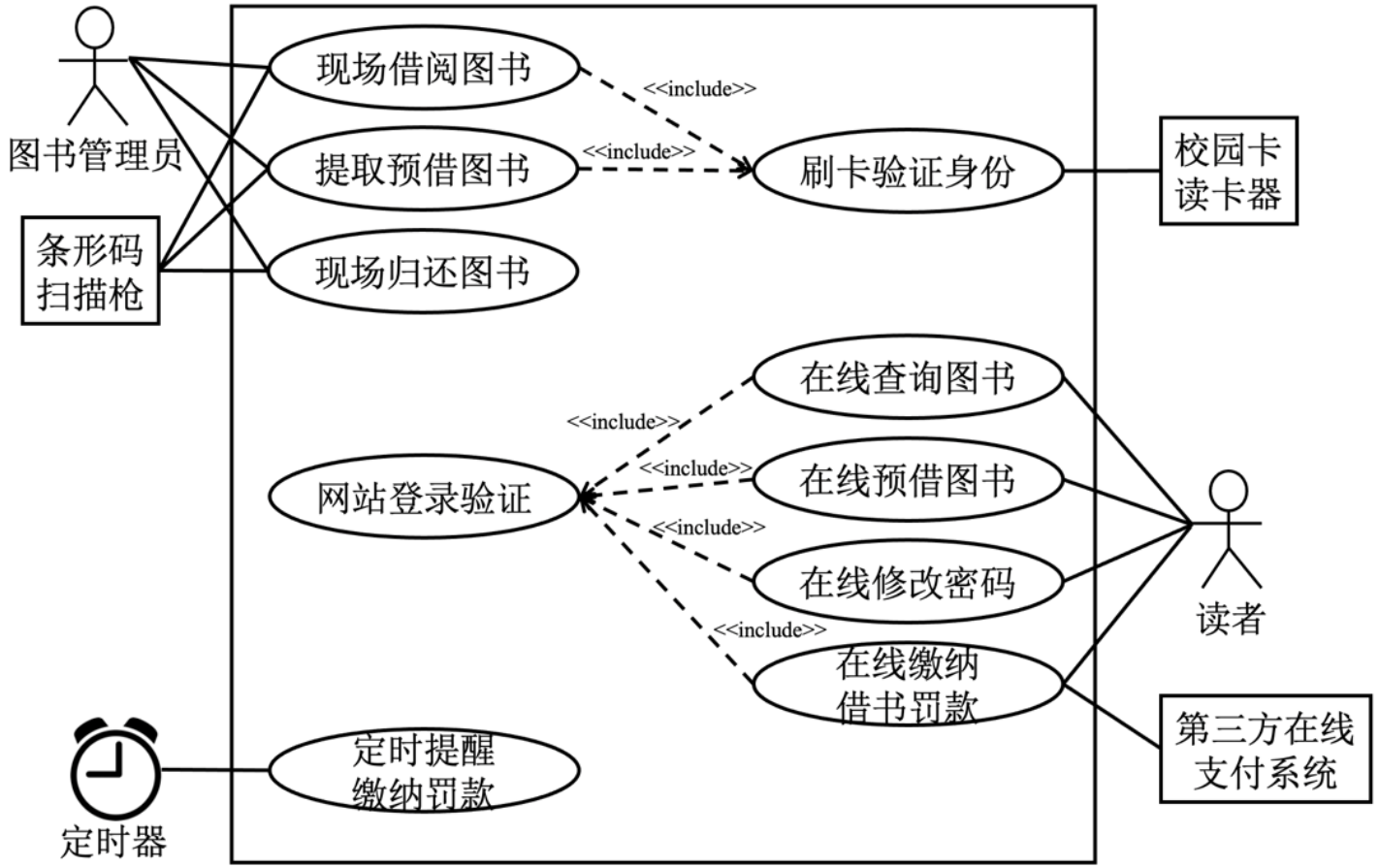


4+1 VIEW MODEL – USE CASE VIEW

- Use-case view, also known as scenario view, is concerned with the functionality that the system provides to end-users
- Use-case view serves as a starting point for tests of an architecture prototype.



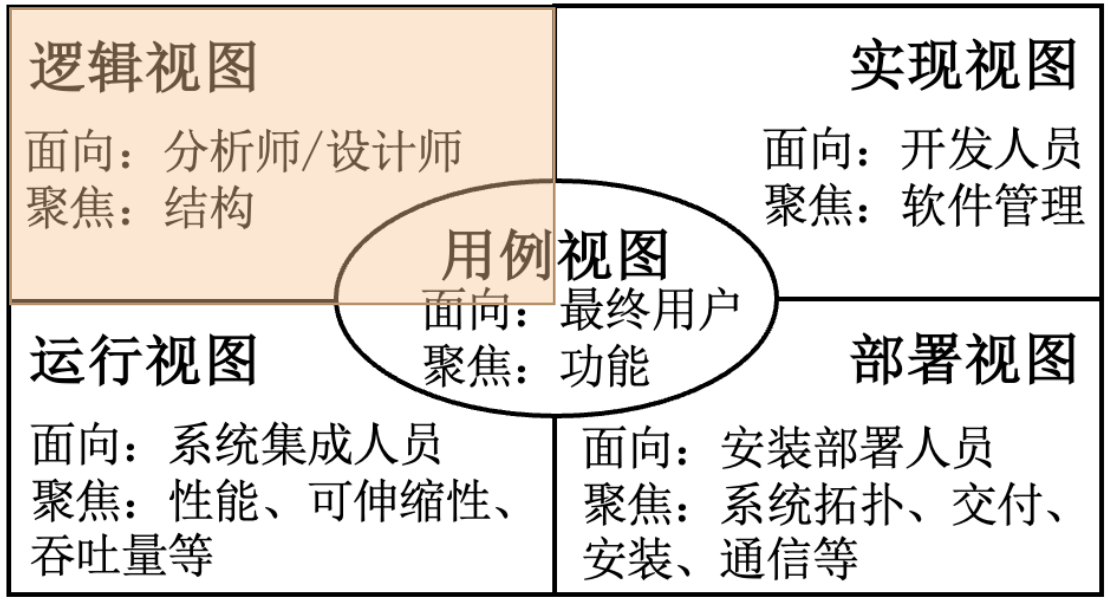
USE CASE VIEW - USE CASE DIAGRAM



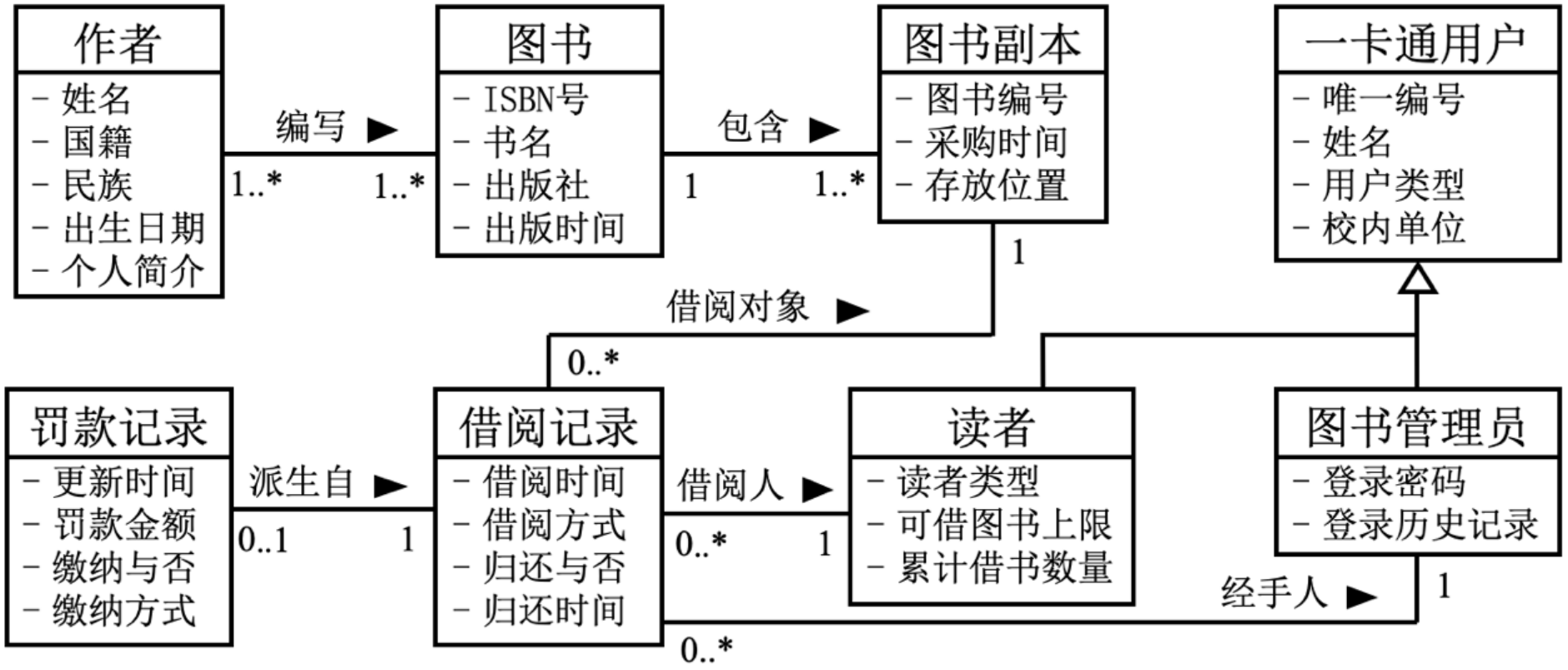
图书馆管理子系统部分-用例模型（用例视图）

4+1 VIEW MODEL – LOGICAL VIEW

- The logical view is concerned with domain concepts, components, relationships, and data models.
- Software designers or analyzers may use the logical view
- The logical view can be derived from UML diagrams such as class diagram, state diagram, and component diagram

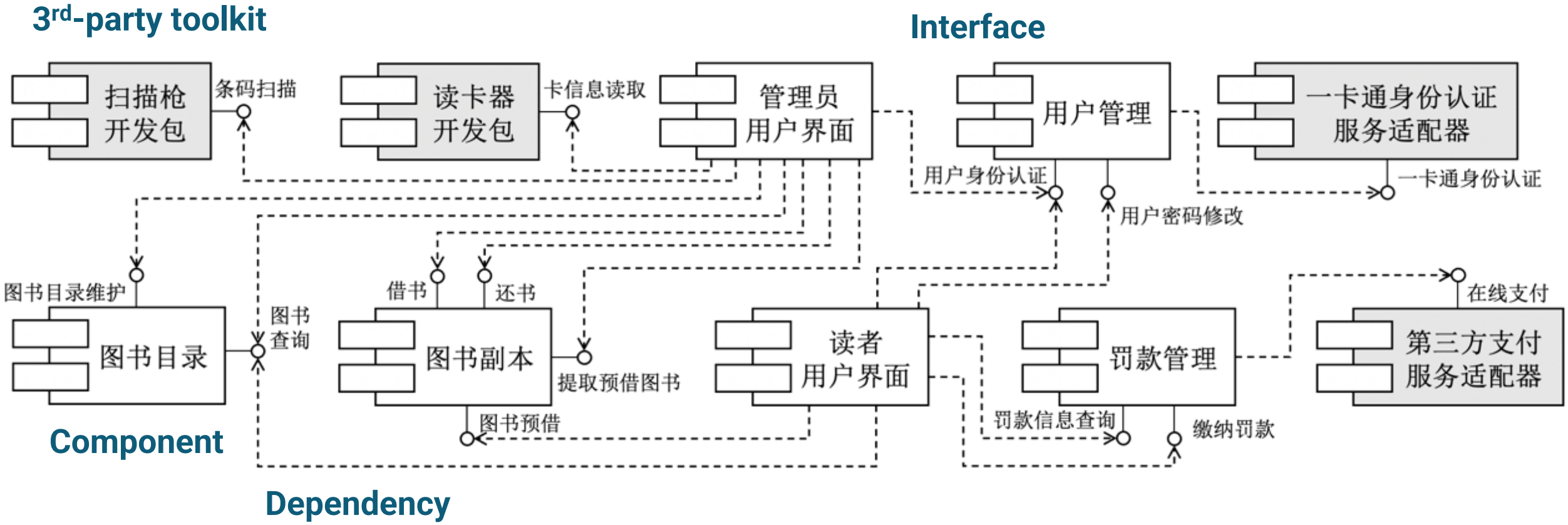


LOGICAL VIEW - CLASS DIAGRAM



图书馆管理子系统部分-数据模型（逻辑视图）

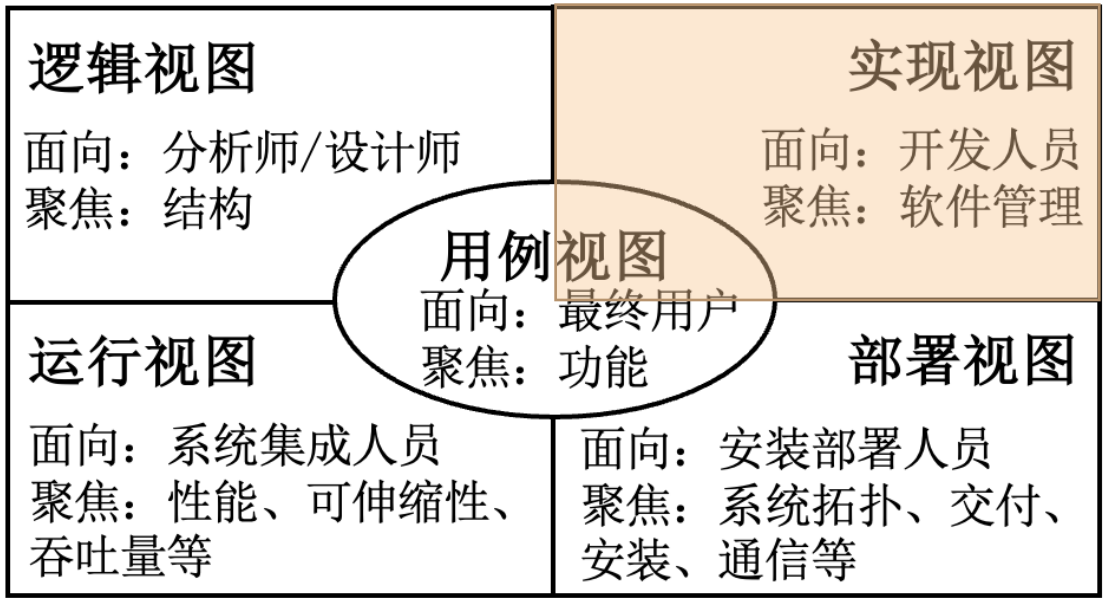
LOGICAL VIEW - COMPONENT DIAGRAM



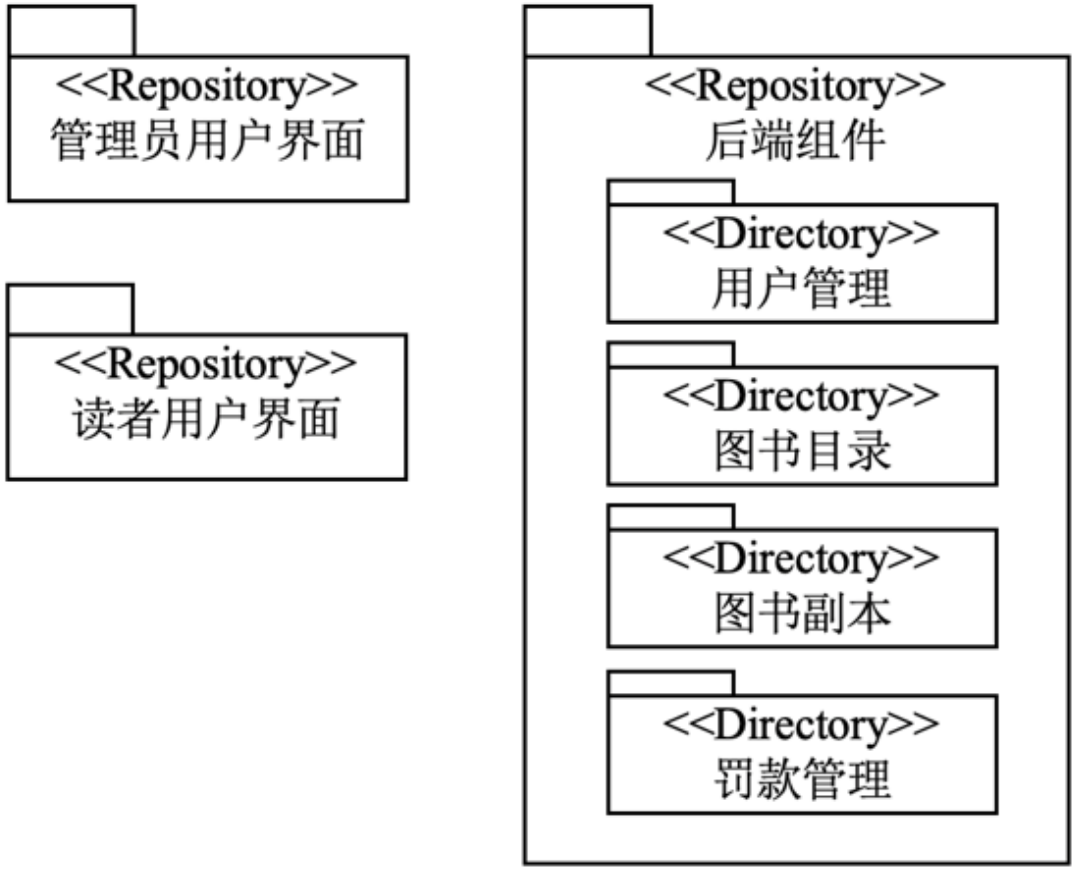
图书馆管理子系统部分-组件模型（逻辑视图）

4+1 VIEW MODEL – DEVELOPMENT VIEW

- Development view, aka implementation view, illustrates a system from a programmer's perspective and is concerned with software management
- Development view may describe how source code or components are organized and depend on each other
- Development view can be derived from UML diagrams such as package diagram and component diagram



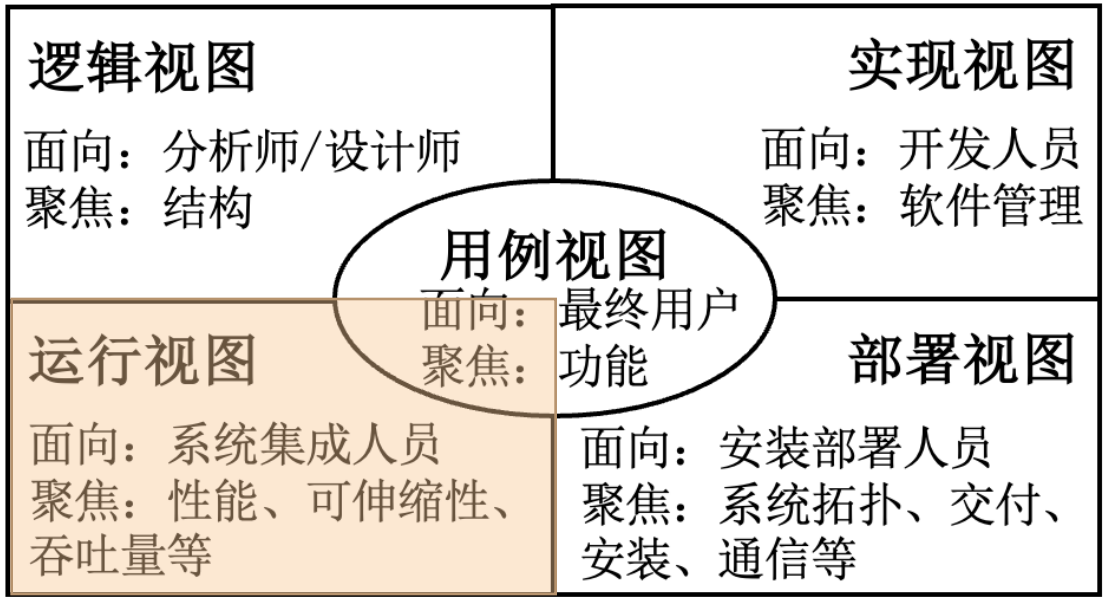
DEVELOPMENT VIEW - PACKAGE DIAGRAM



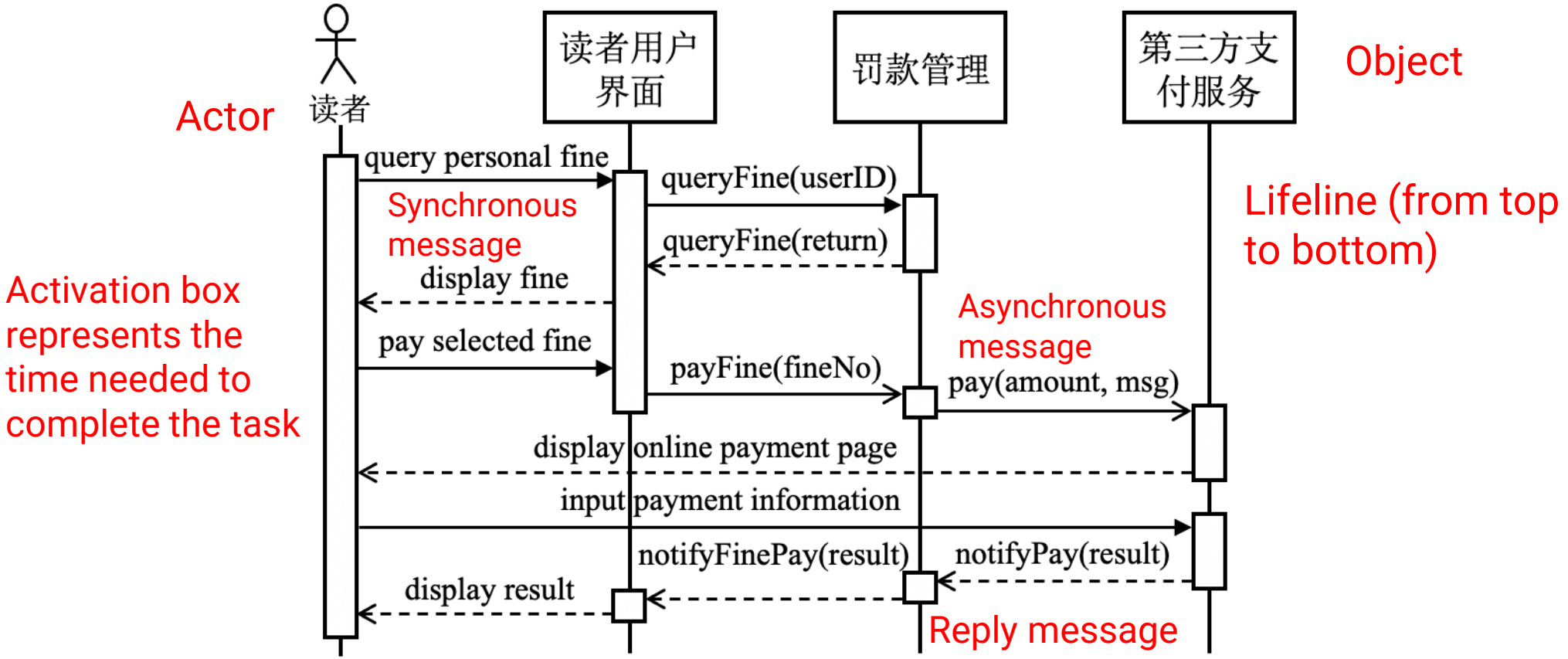
图书馆管理子系统部分-代码组织结构（实现视图）

4+1 VIEW MODEL – PROCESS VIEW

- The process view deals with the dynamic aspects of the system, explains **the system processes and how they communicate**, and focuses on the run time behavior of the system.
- The process view addresses concurrency, performance, etc.
- Process view can be derived from UML diagrams such as activity diagram and sequence diagram



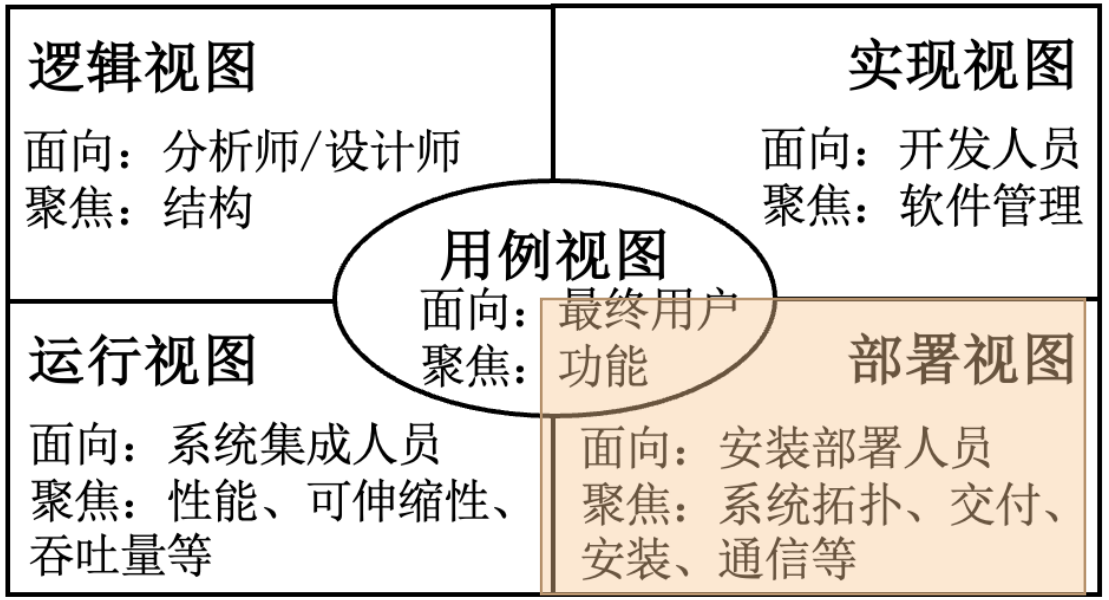
PROCESS VIEW - SEQUENCE DIAGRAM



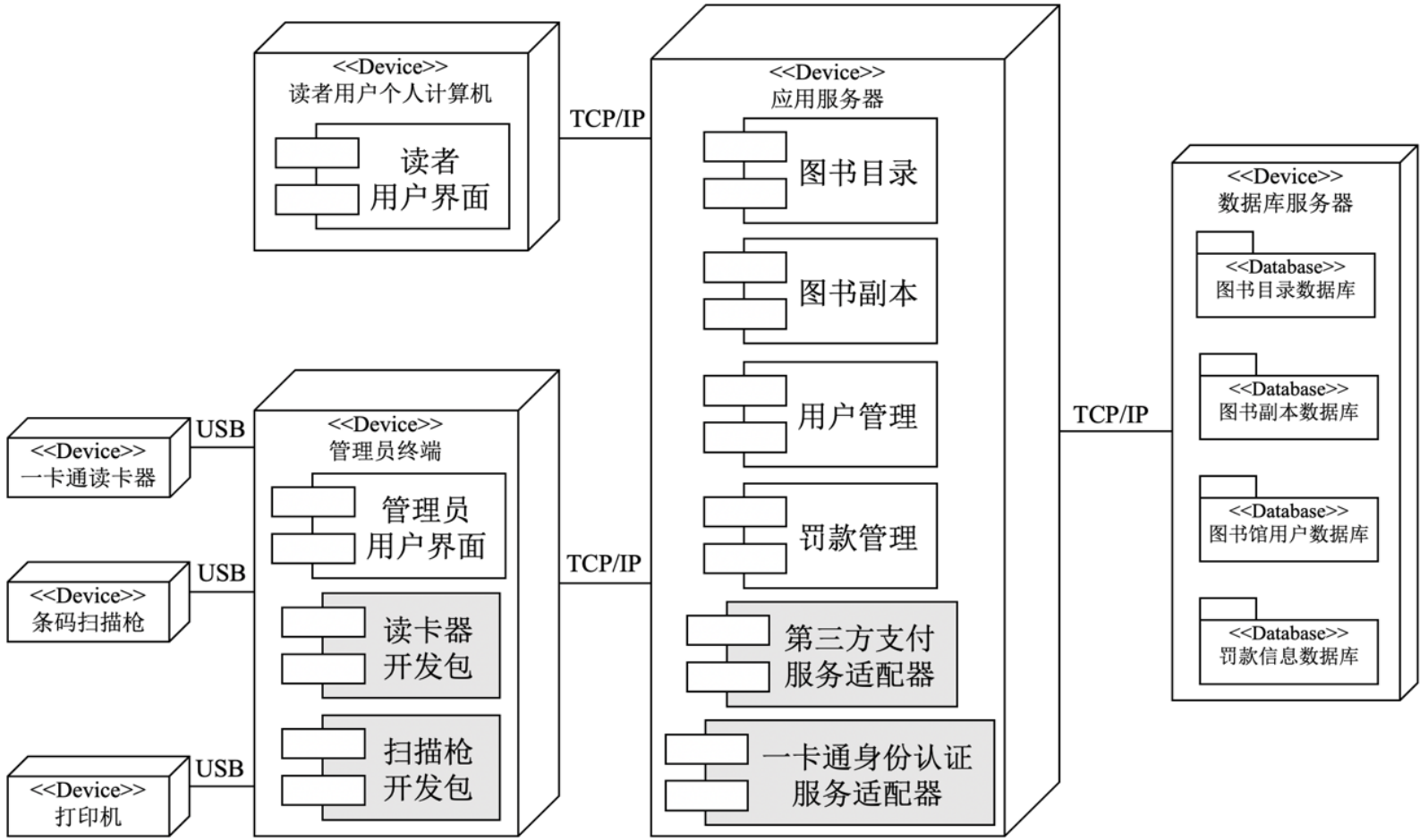
图书馆管理子系统部分-进程交互（运行视图）

4+1 VIEW MODEL – DEPLOYMENT VIEW

- Deployment view (or physical view) depicts the system from a system engineer's point of view
- It is concerned with the topology of software components on the physical layer as well as the physical connections between these components
- Deployment view can be derived from UML diagrams such as deployment diagram

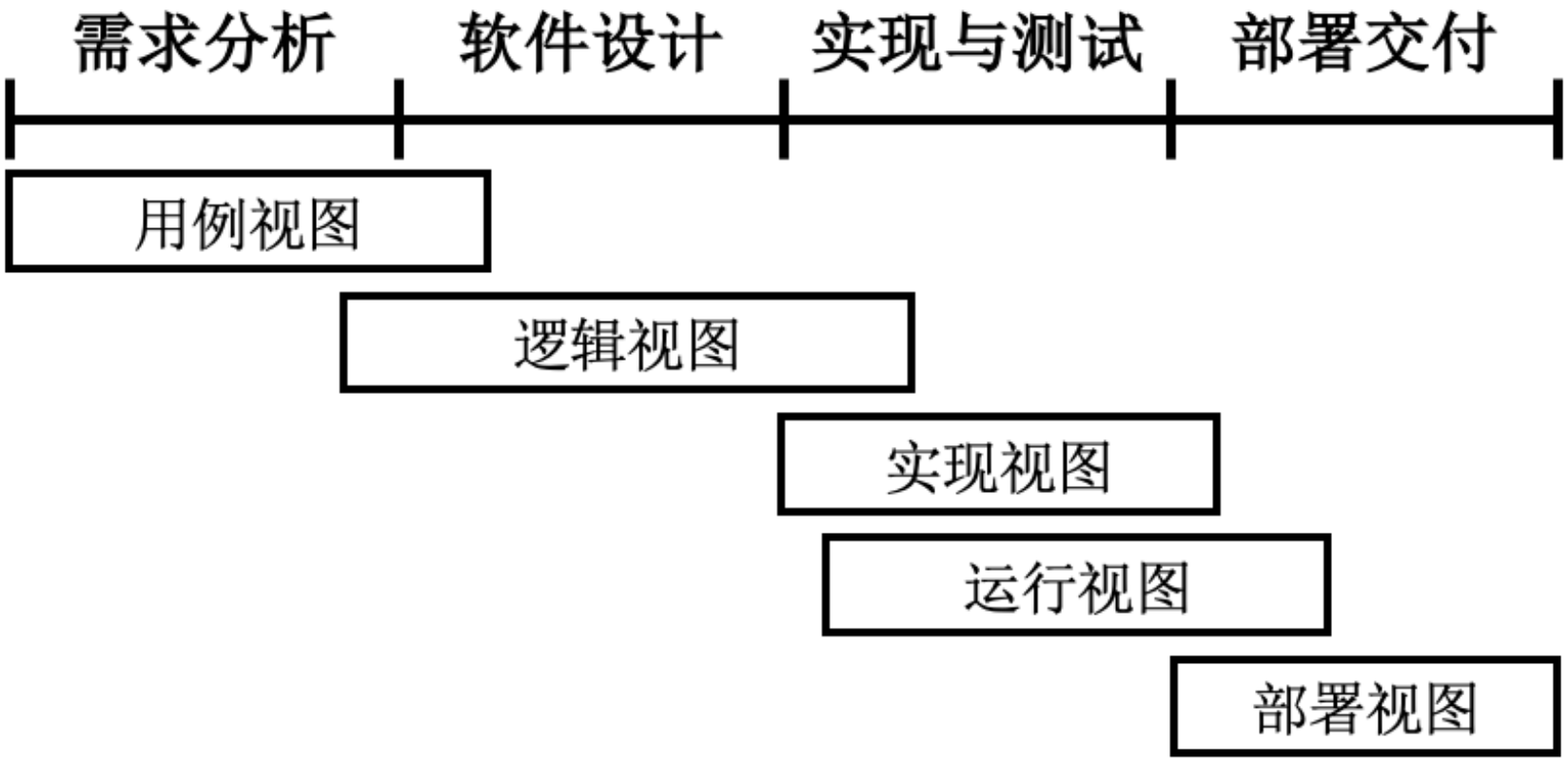


DEPLOYMENT VIEW - DEPLOYMENT DIAGRAM



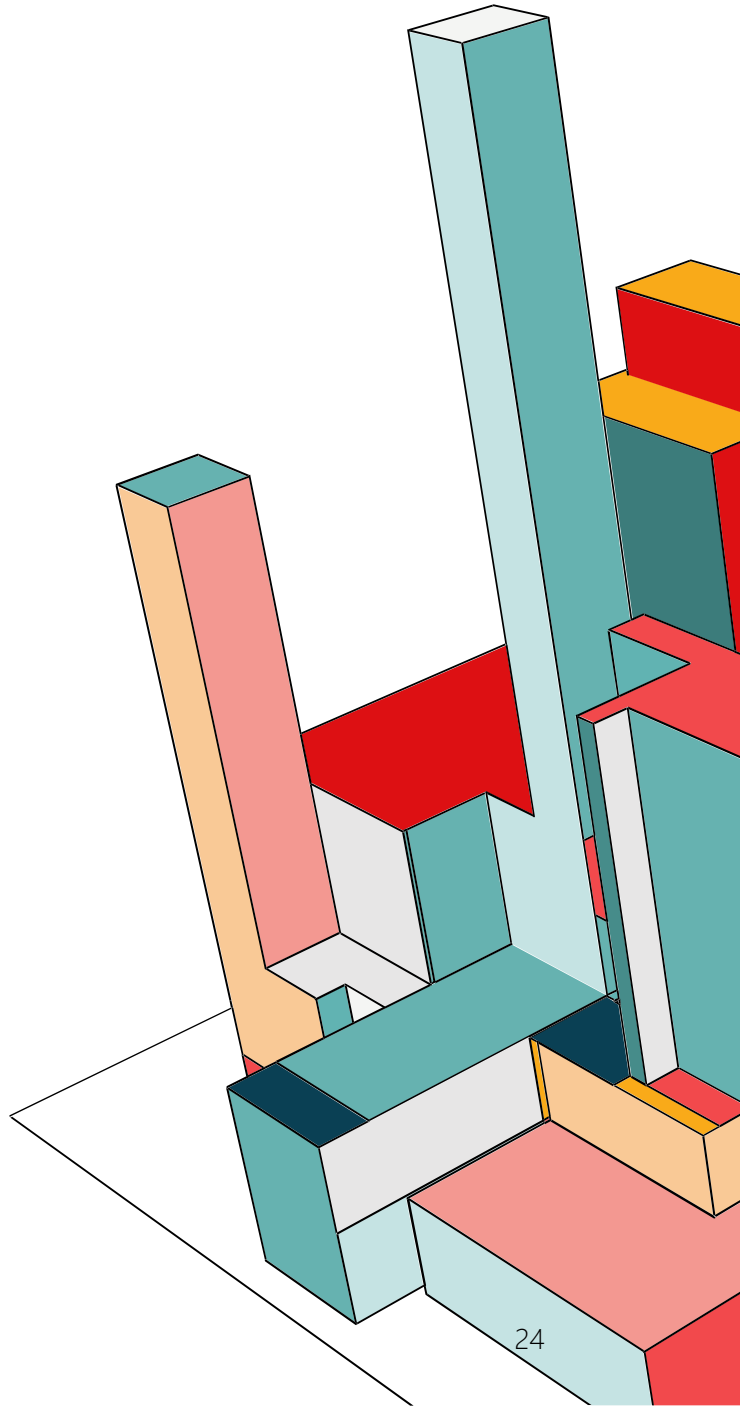
图书馆管理子系统部分-部署结构（部署视图）

ARCHITECTURAL VIEWS IN DIFFERENT PHASES



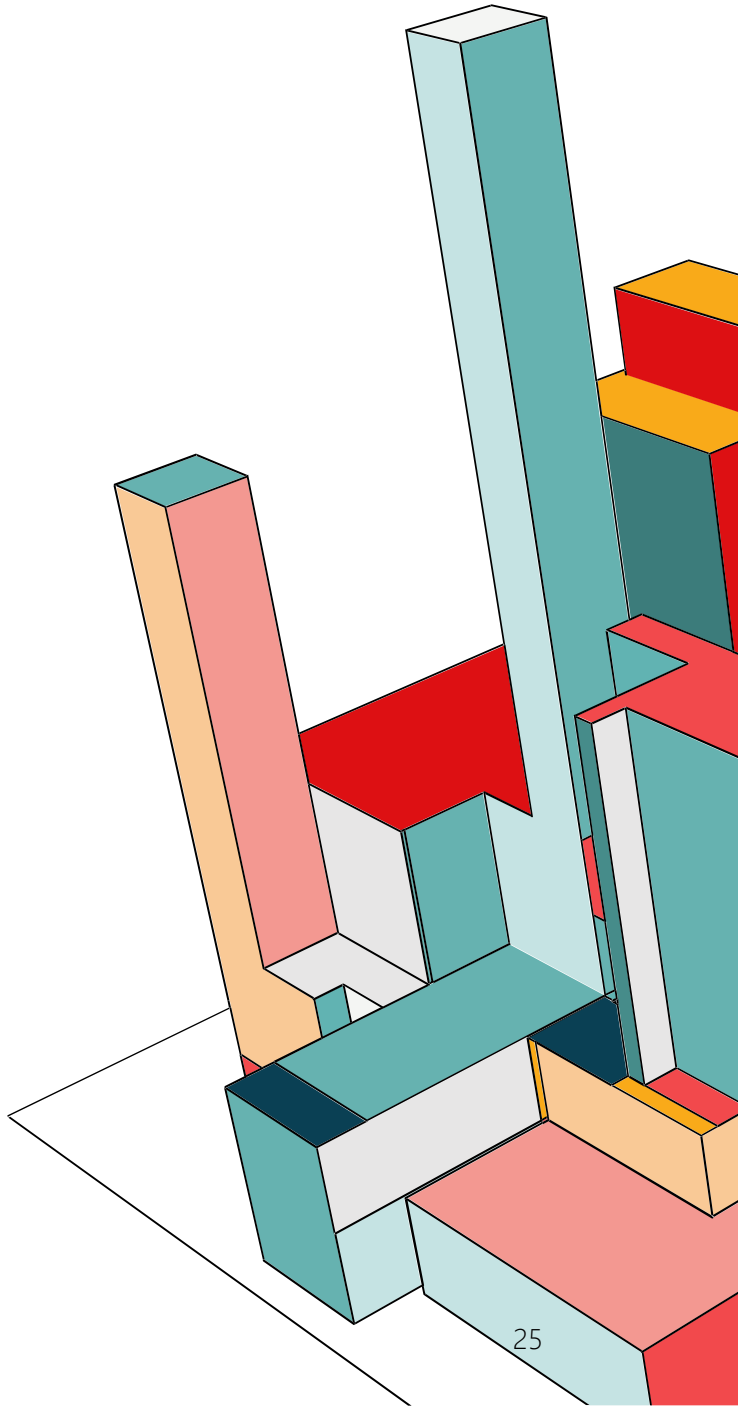
ARCHITECTURAL STYLE

- An architectural style is a transformation that is imposed on the design of an entire system.
- The intent is to establish a structure for all components of the system.
- When an existing architecture is to be reengineered, the imposition of an architectural style will result in fundamental changes to the structure of the software including a reassignment of the functionality of components

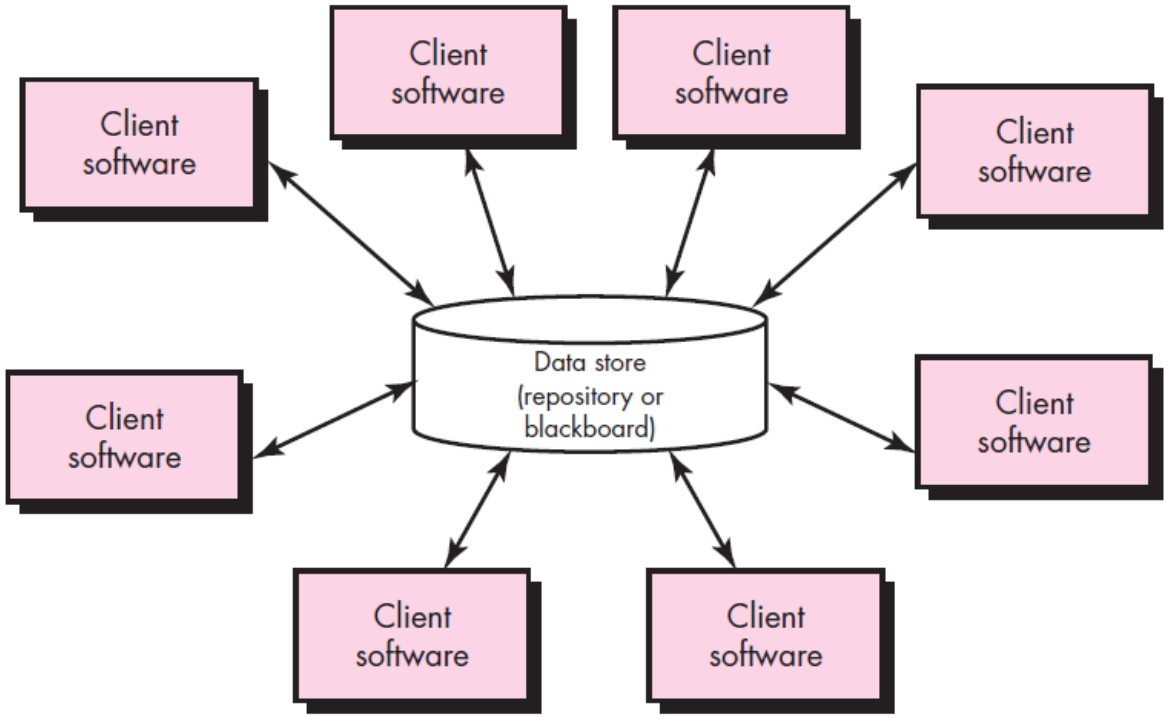


ARCHITECTURAL STYLE

- Data-centered architecture
- Data-flow architecture
- Call-and-return architecture
- Layered architecture
- Object-oriented architecture



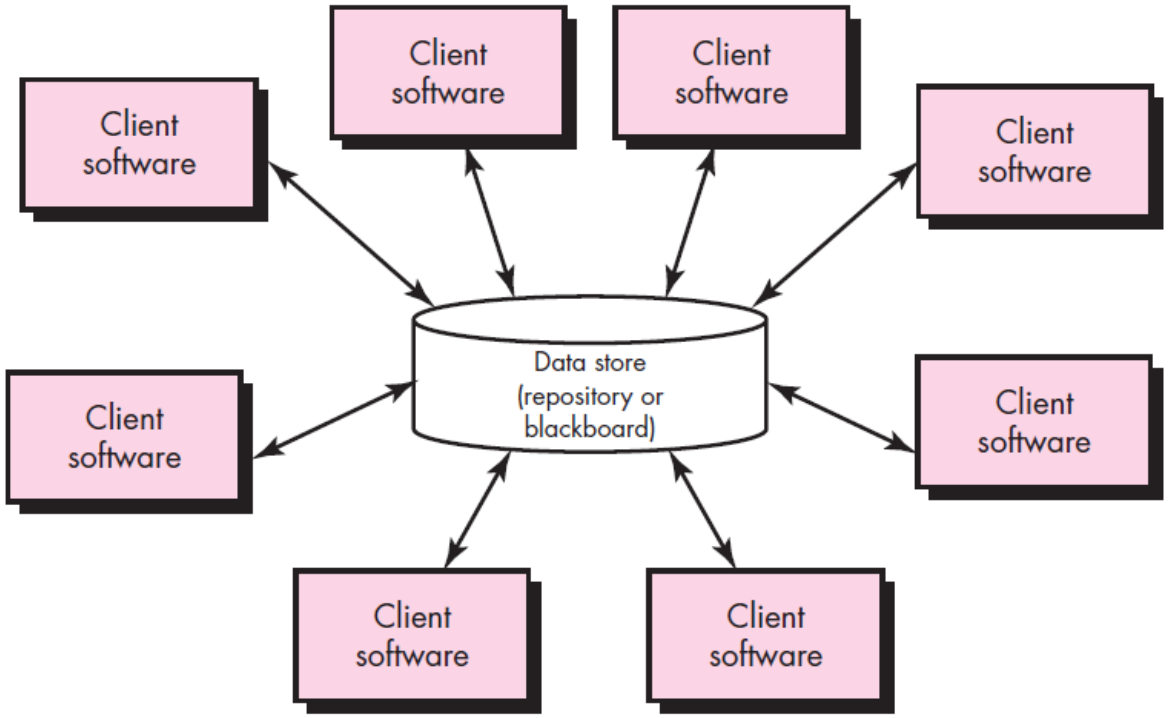
DATA-CENTERED ARCHITECTURES



以数据为中心的体系结构

- A data store (e.g., a file or database) resides at the center of this architecture
- The data store is accessed frequently by other components that update, add, delete, or modify data within the store
- A variation on this approach transforms the repository into a “blackboard” that sends notifications to client software when data of interest to the client changes.

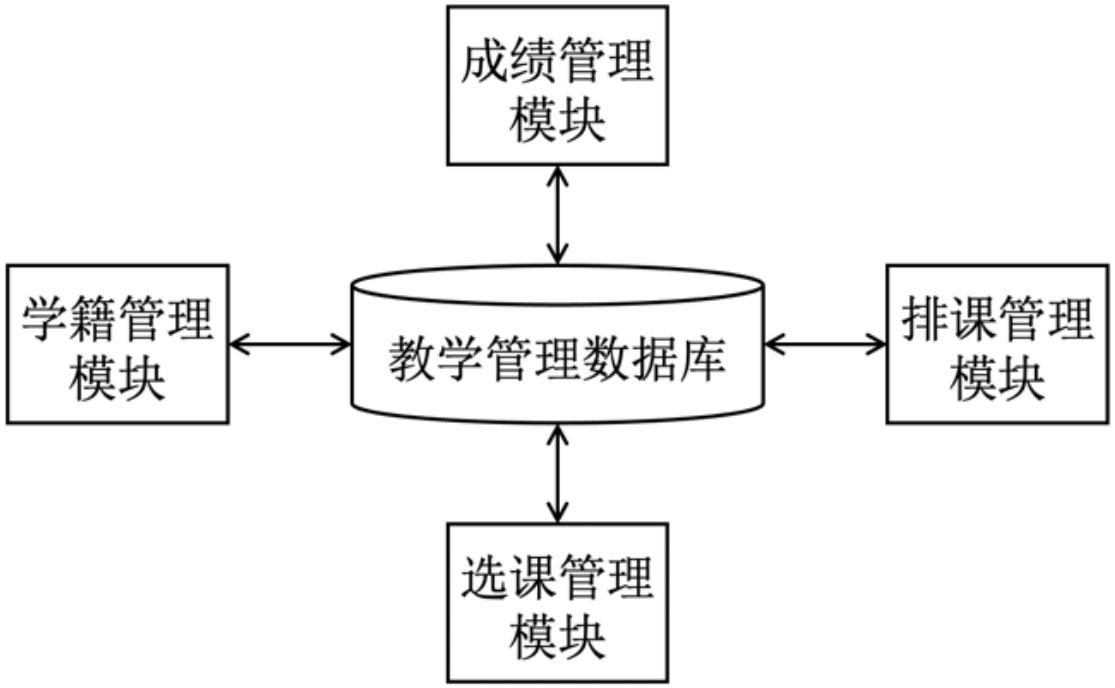
DATA-CENTERED ARCHITECTURES



以数据为中心的体系结构

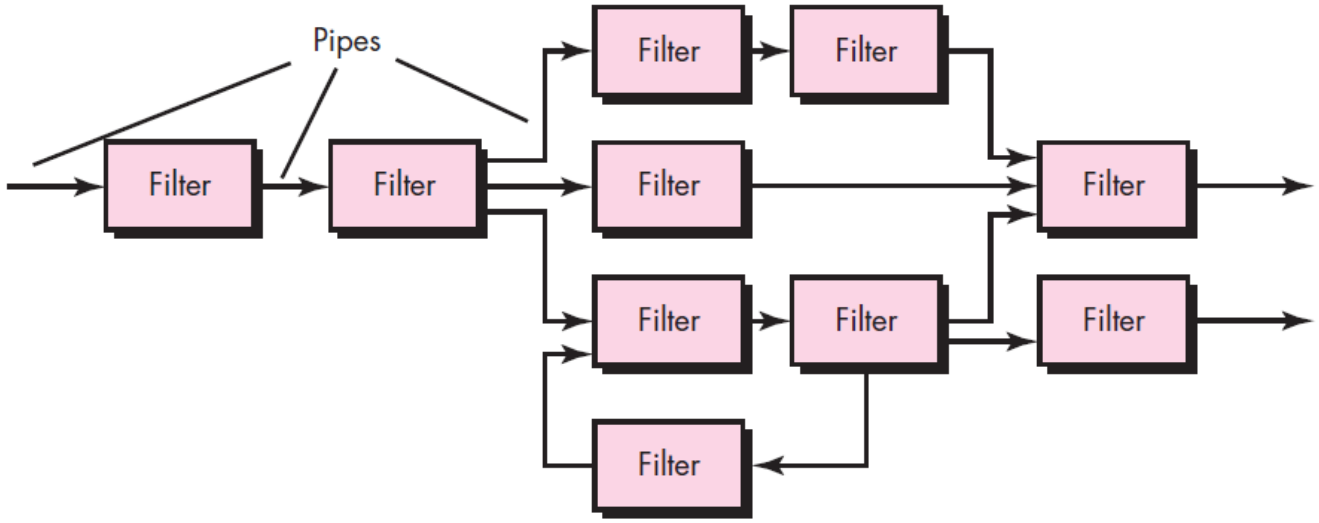
- This architecture promotes integrability (可集成性): ability to make separately developed components to work correctly together
- Client components operate independently
- Data can be passed among clients using the blackboard mechanism

DATA-CENTERED ARCHITECTURES



以数据为中心的体系结构示例

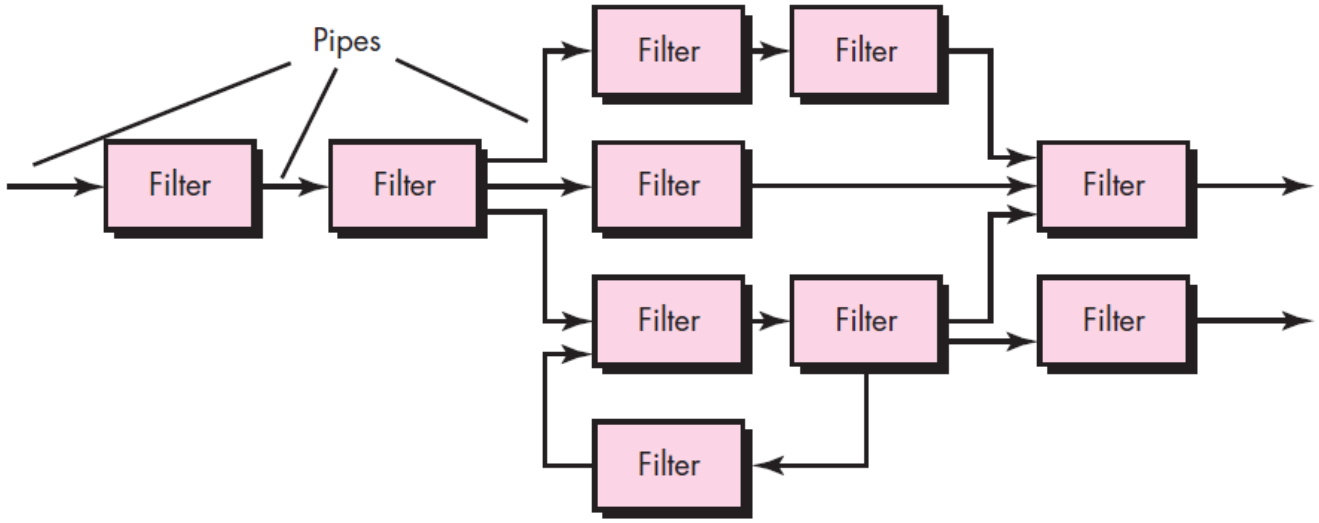
DATA-FLOW ARCHITECTURES



数据流体系结构

This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data.

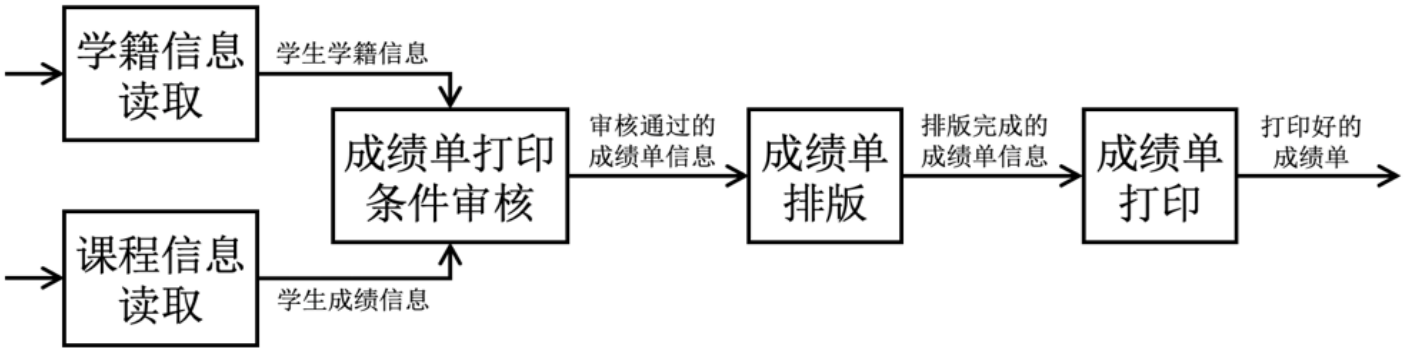
DATA-FLOW ARCHITECTURES



数据流体系结构（管道和过滤器）

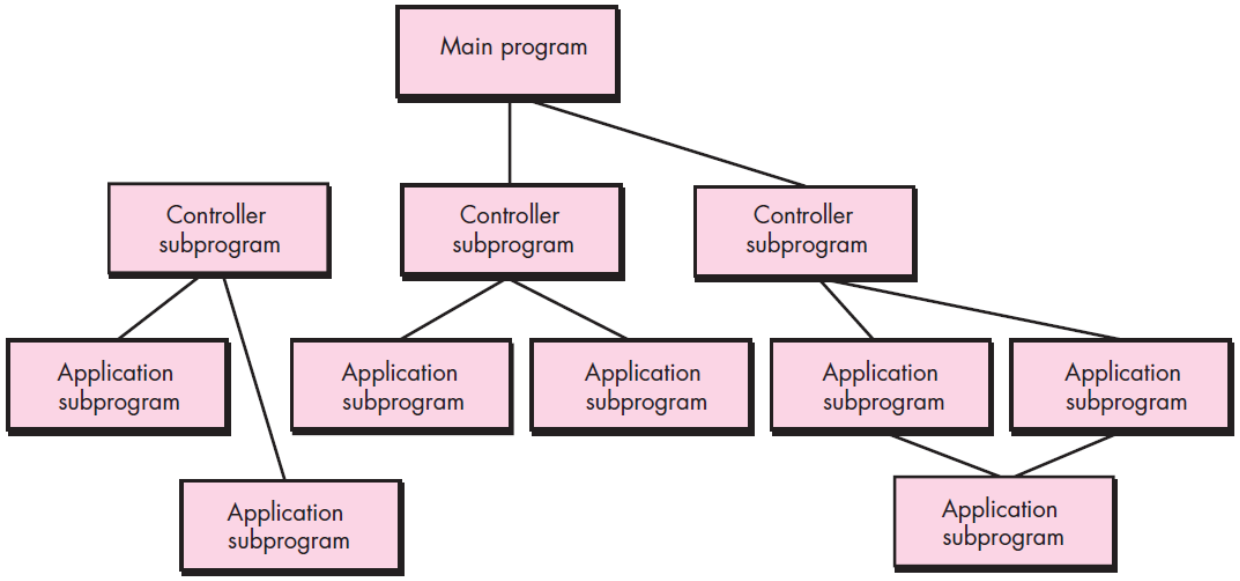
- A pipe-and-filter pattern has a set of components, called filters, connected by pipes that transmit data from one component to the next.
- Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form.
- The filter does not require knowledge of the workings of its neighboring filters.

DATA-FLOW ARCHITECTURES



- Data-flow architecture is suitable for automated data analysis and transmission systems
- Such systems contain a series of data analysis components, with almost no user interaction
- Data-flow architecture may not be suitable for GUI intensive systems

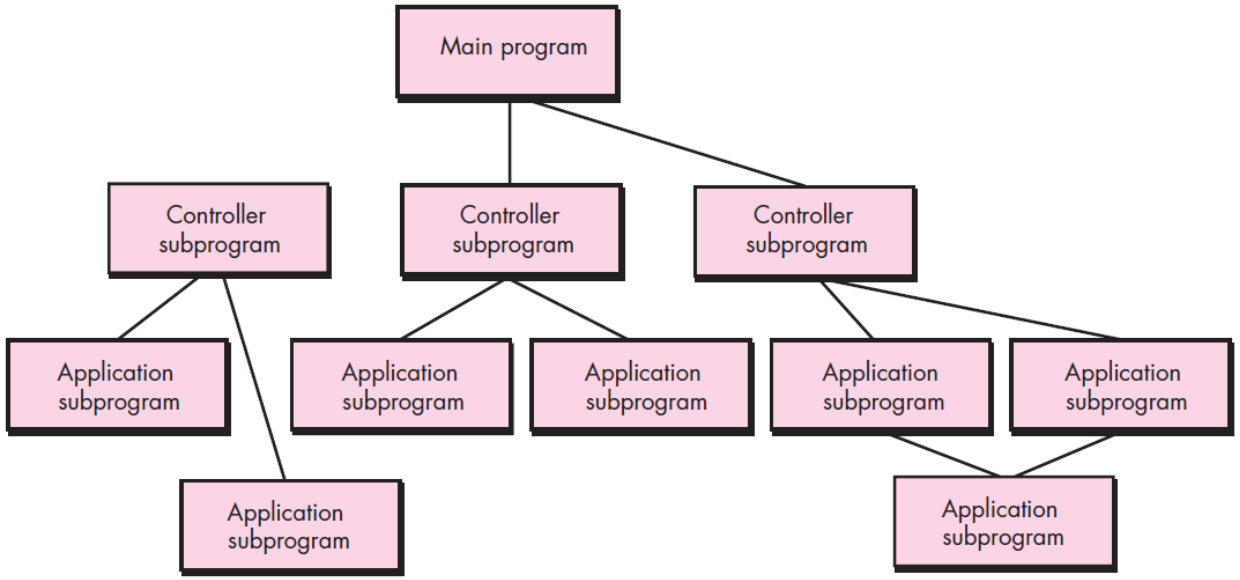
CALL AND RETURN ARCHITECTURES



主程序/子程序体系结构

- This architectural style enables you to achieve a program structure that is relatively easy to modify and scale.
- There are a few substyles (next slide)

CALL AND RETURN ARCHITECTURES

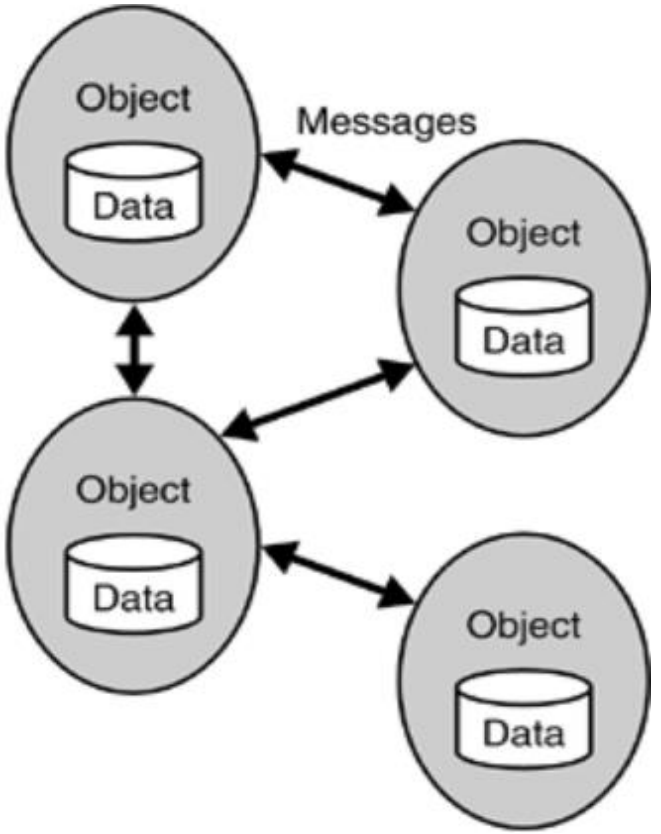


主程序/子程序体系结构

- **Main program/subprogram architecture** decomposes function into a control hierarchy where a “main” program invokes a number of program components that in turn may invoke still other components.
- **Remote procedure call architecture**: The components of a main program/subprogram architecture are distributed across multiple computers on a network.

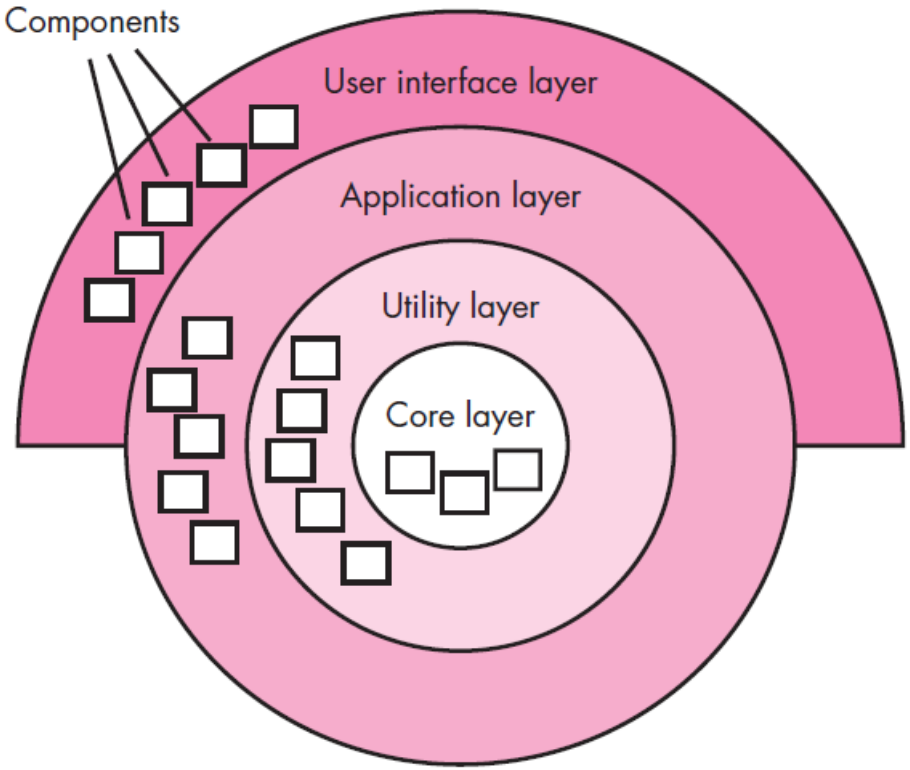
OBJECT-ORIENTED ARCHITECTURES

- The components of a system encapsulate data and the operations that must be applied to manipulate the data.
- Communication and coordination between components are accomplished via message passing

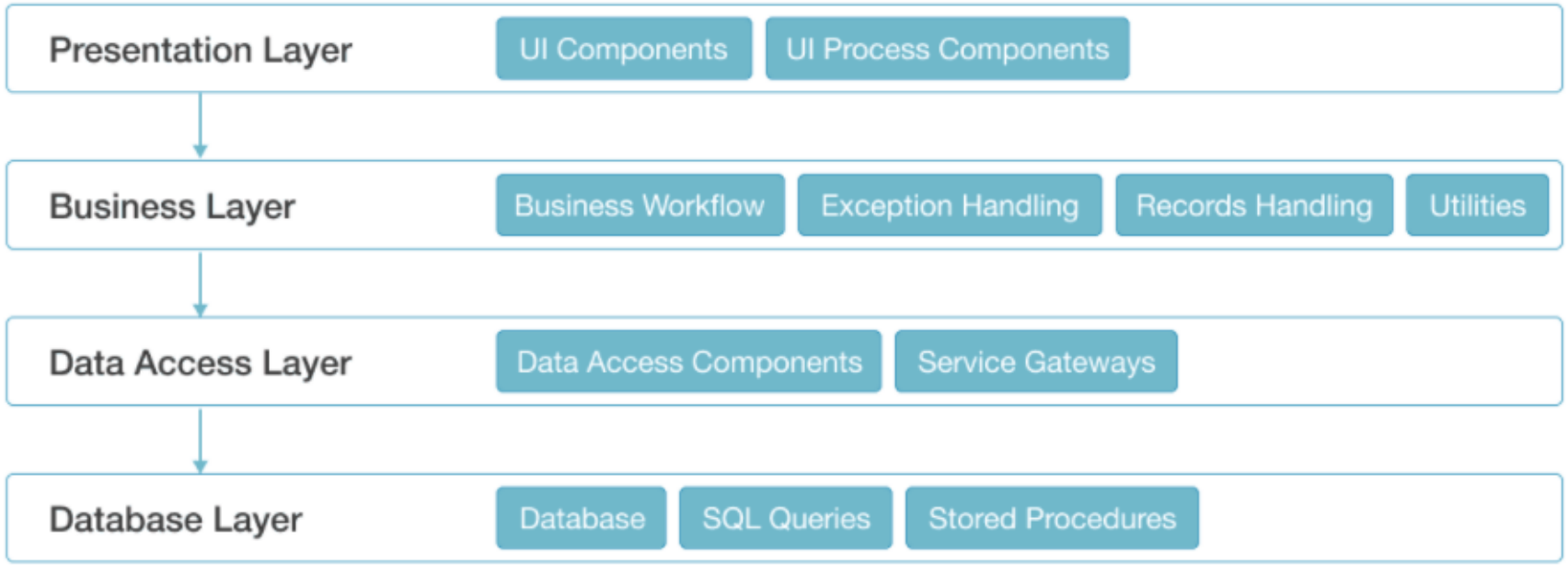


LAYERED ARCHITECTURES

- A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set.
- At the outer layer, components service user interface operations.
- At the inner layer, components perform operating system interfacing.
- Intermediate layers provide utility services and application software functions.



LAYERED ARCHITECTURES



Layered architecture for web applications

https://www.simform.com/blog/web-application-architecture/

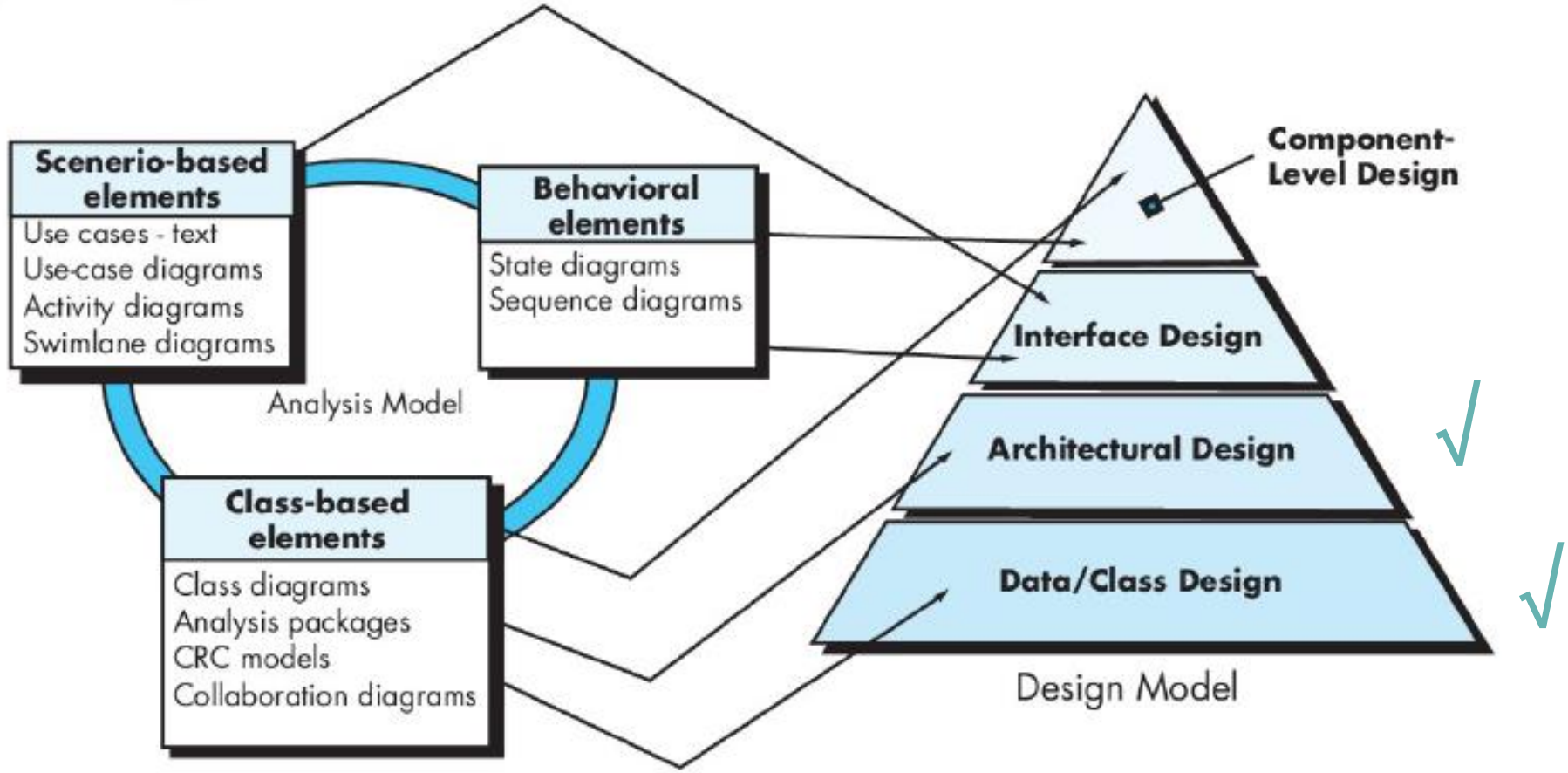
CHOOSING ARCHITECTURAL STYLES

- We've introduced only a subset of available architectural styles
- Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural style that best fits those characteristics and constraints can be chosen.
- Different architectural styles are NOT mutually exclusive; instead, they are often applied in combination

CHOOSING ARCHITECTURAL STYLES

- For example, a layered style can be combined with a data-centered architecture in many database applications.
- In a browser-server web application, both layered architecture and object-oriented architecture can be applied

NEXT: INTERFACE DESIGN



INTERFACE DESIGN

The interface design for software depict information flows into and out of the system and how it is communicated among the components defined as part of the architecture.

Three important elements of interface design:

- UI Design
- External interfaces to other systems, devices, networks, or other producers or consumers of information
- Internal interfaces between various design components

These interface design elements allow the software to communicate externally and enable internal communication and collaboration among the components of software architecture.

UI DESIGN

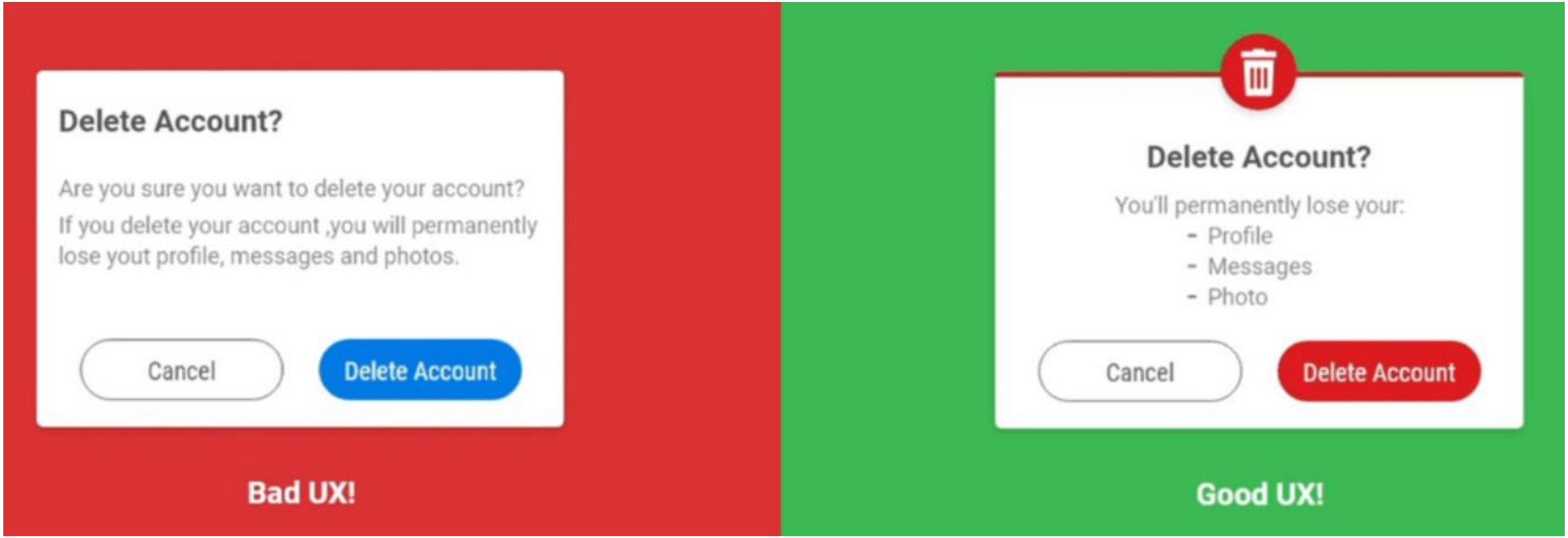
- **UI design** (increasingly called *usability design*) is a major software engineering action

- UI design incorporates different elements
  - Aesthetic elements (美学): layout, color, graphics, interaction mechanisms, etc.
  - Ergonomic elements (人体工程学): information layout and placement, UI navigation
  - Technical elements (技术元素): UI patterns, reusable components

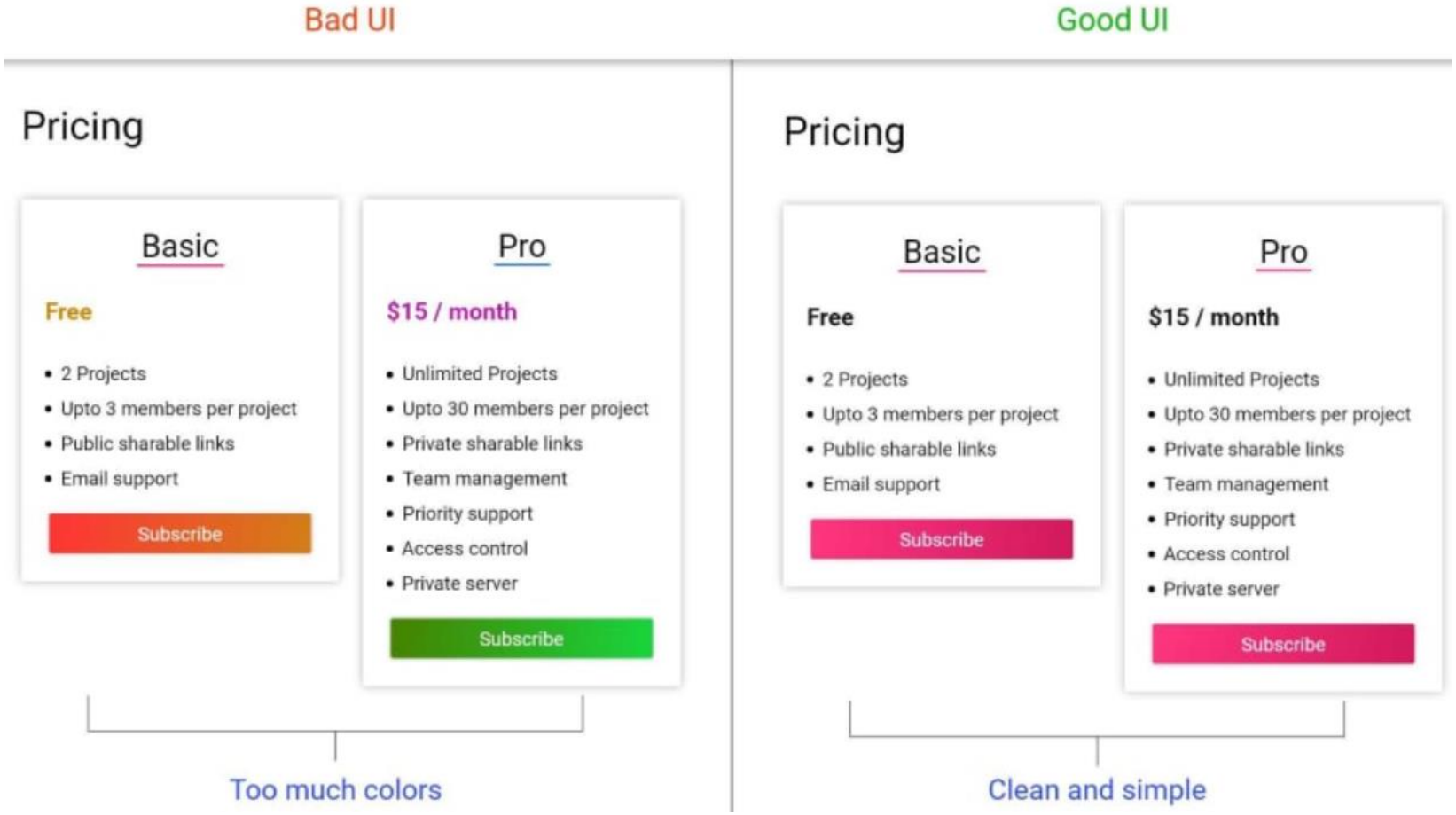
- The goal of UI design is to make the user's interaction as simple and efficient as possible, in terms of accomplishing user goals

BAD UI DESIGN

BAD UI DESIGN



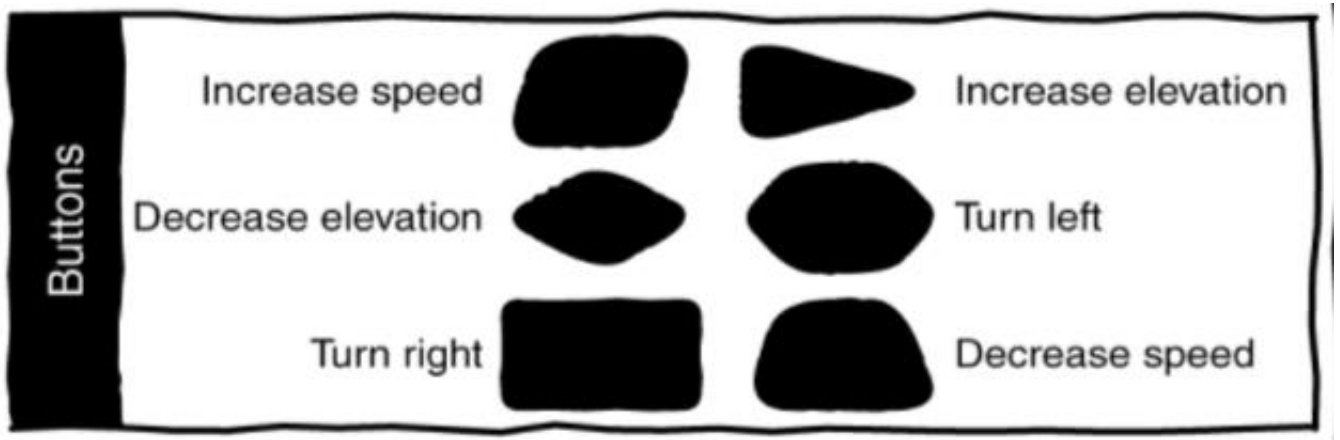
BAD UI DESIGN



https://dev.to/piyushpawar17/ui-design-for-front-end-developers-2f2o

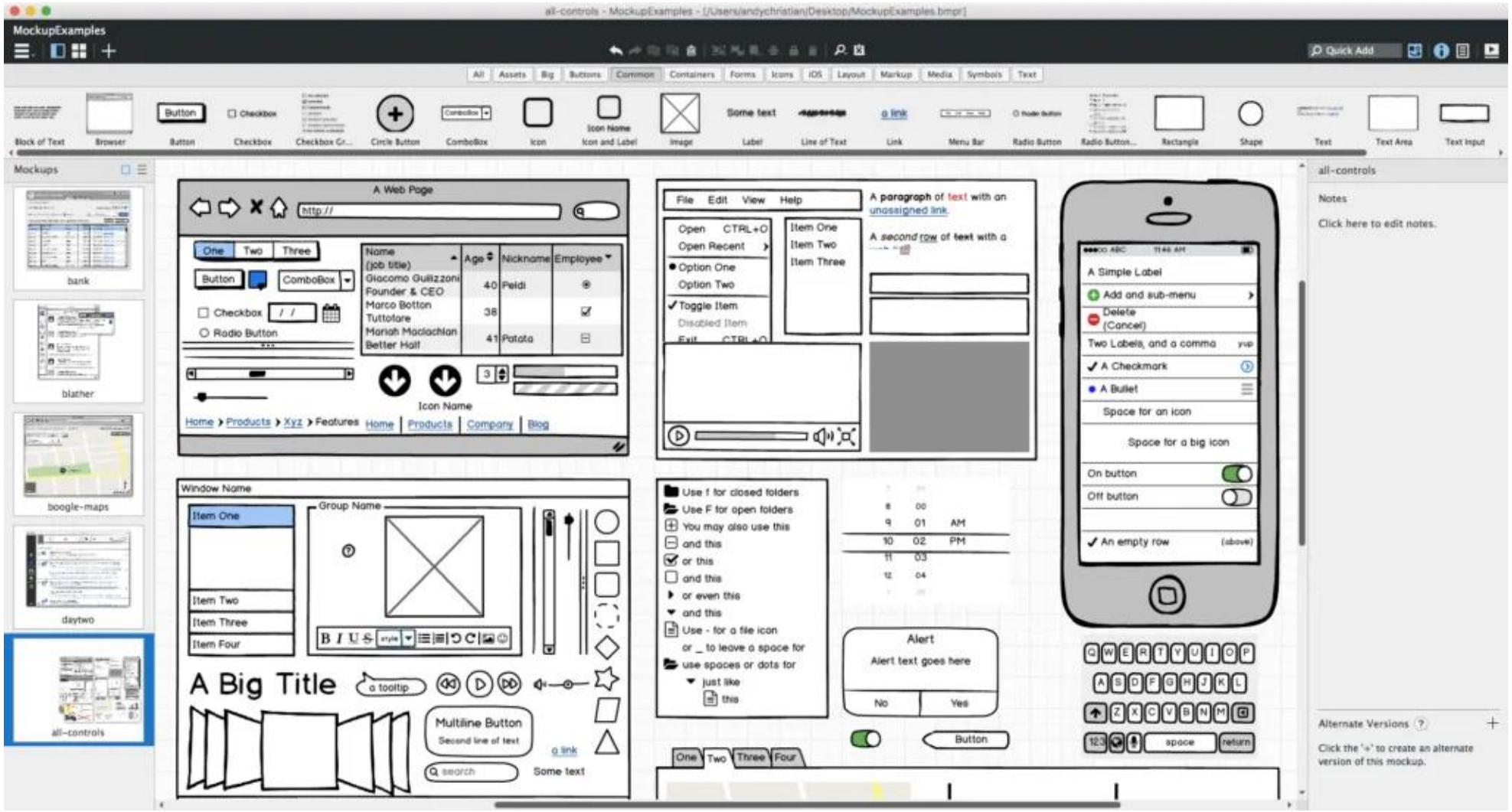
BAD UI DESIGN

Drone remote controller



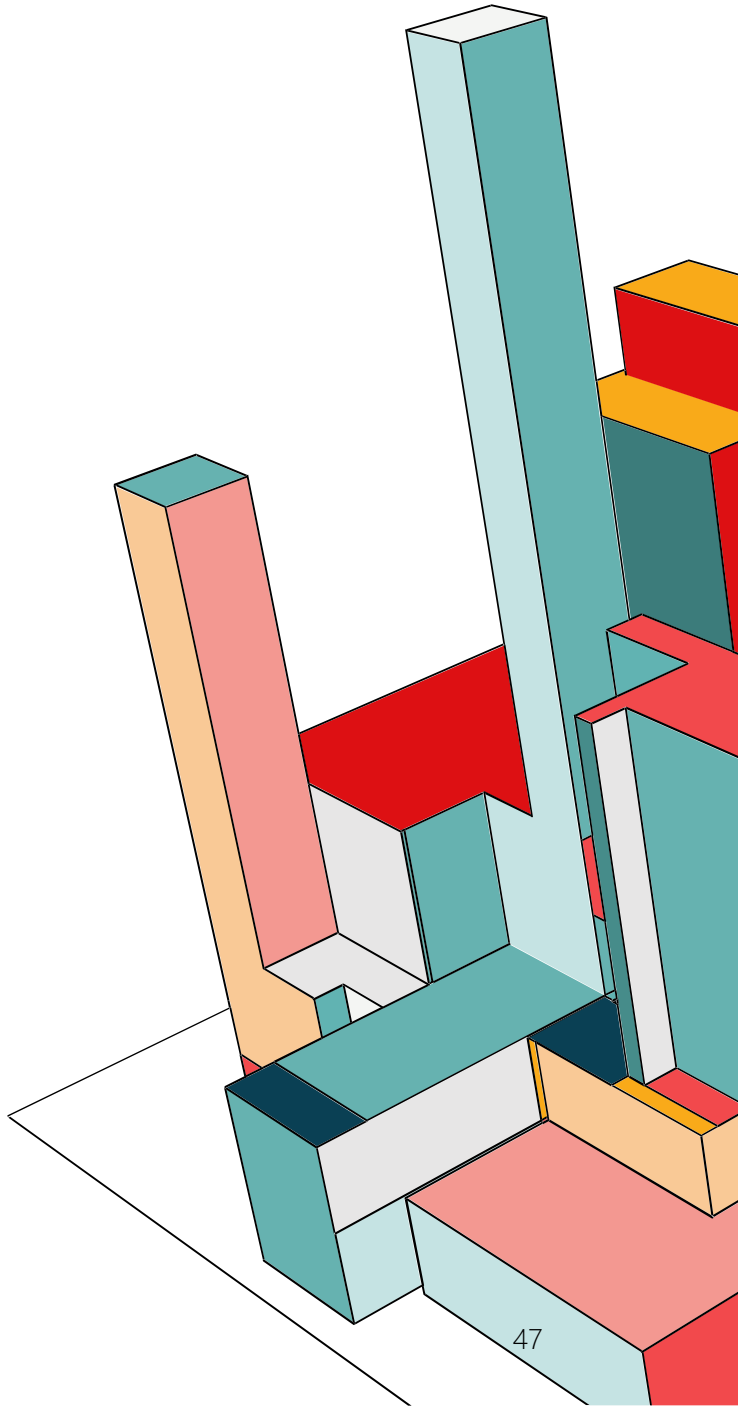
https://livebook.manning.com/book/the-design-of-web-apis/chapter-1/57

You may find various tools for UI design



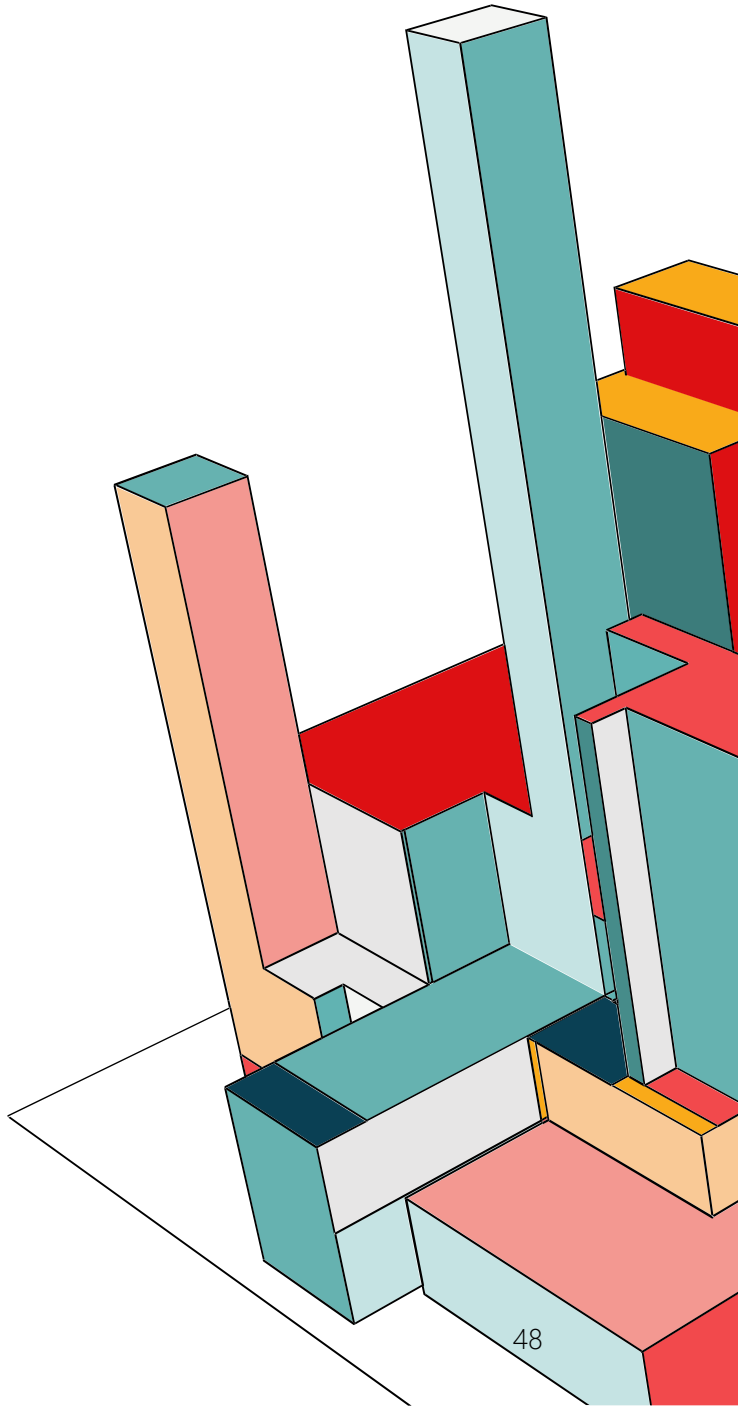
EXTERNAL INTERFACE DESIGN

- Designing external interfaces requires definitive information about the entity to which information is sent (e.g., system) or received (user)
- This information should be collected during requirements engineering
- The design of external interfaces should incorporate error checking and (when necessary) appropriate security features.

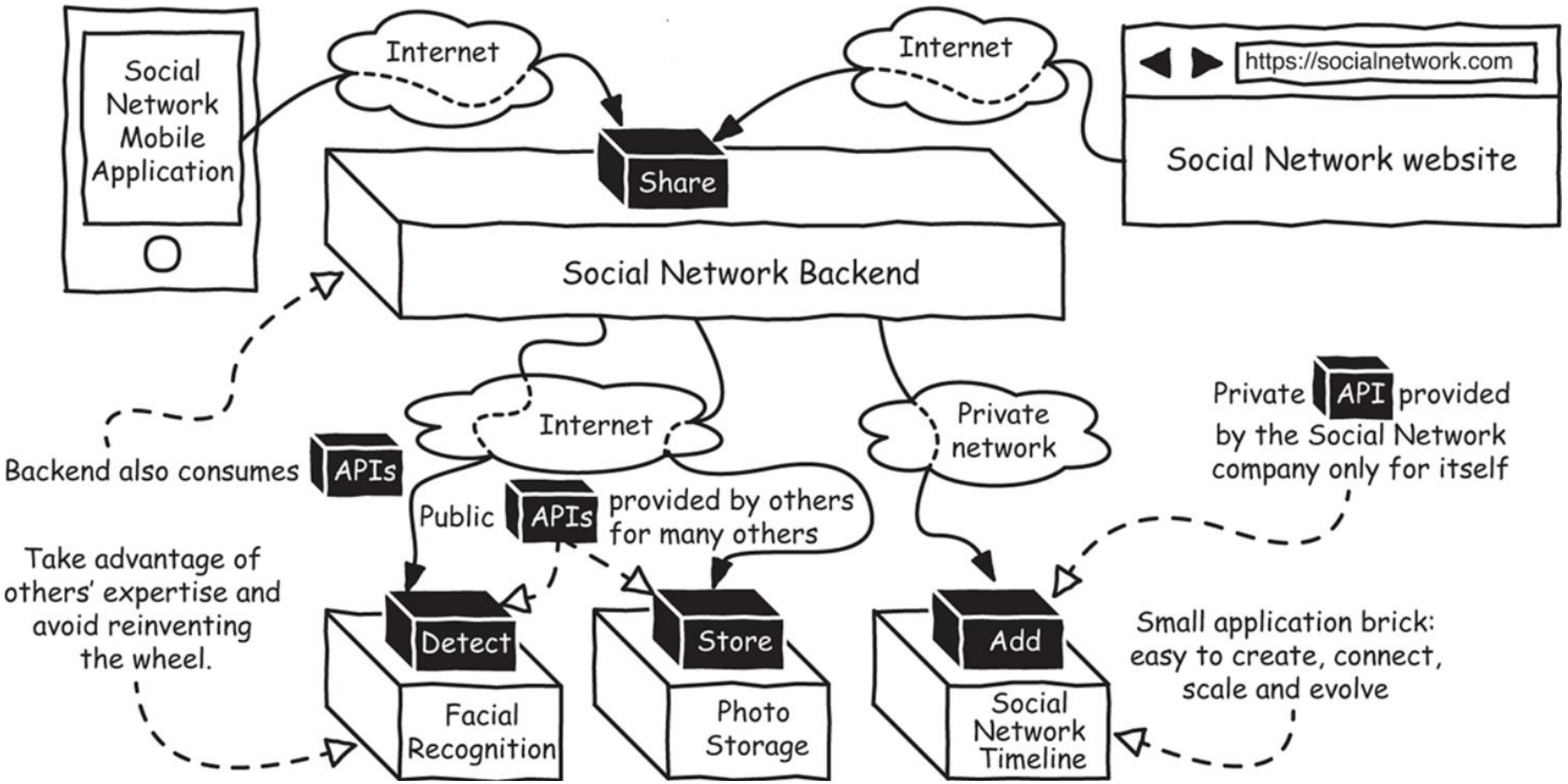


INTERNAL INTERFACE DESIGN

- Internal interfaces represent all operations and the messaging schemes required to enable communication and collaboration between operations in various internal classes.
- Internal interfaces should be designed to satisfy the information and functionality requirements of the requested operations



EXAMPLE OF EXTERNAL/INTERNAL INTERFACES

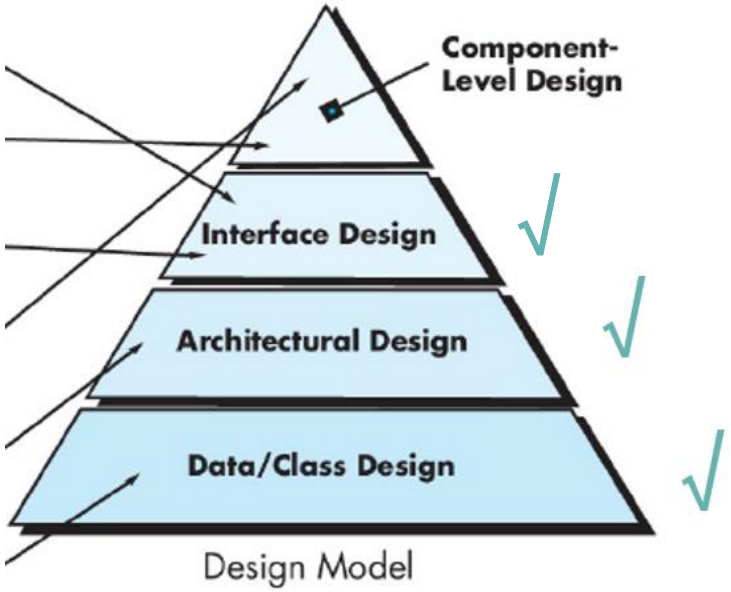


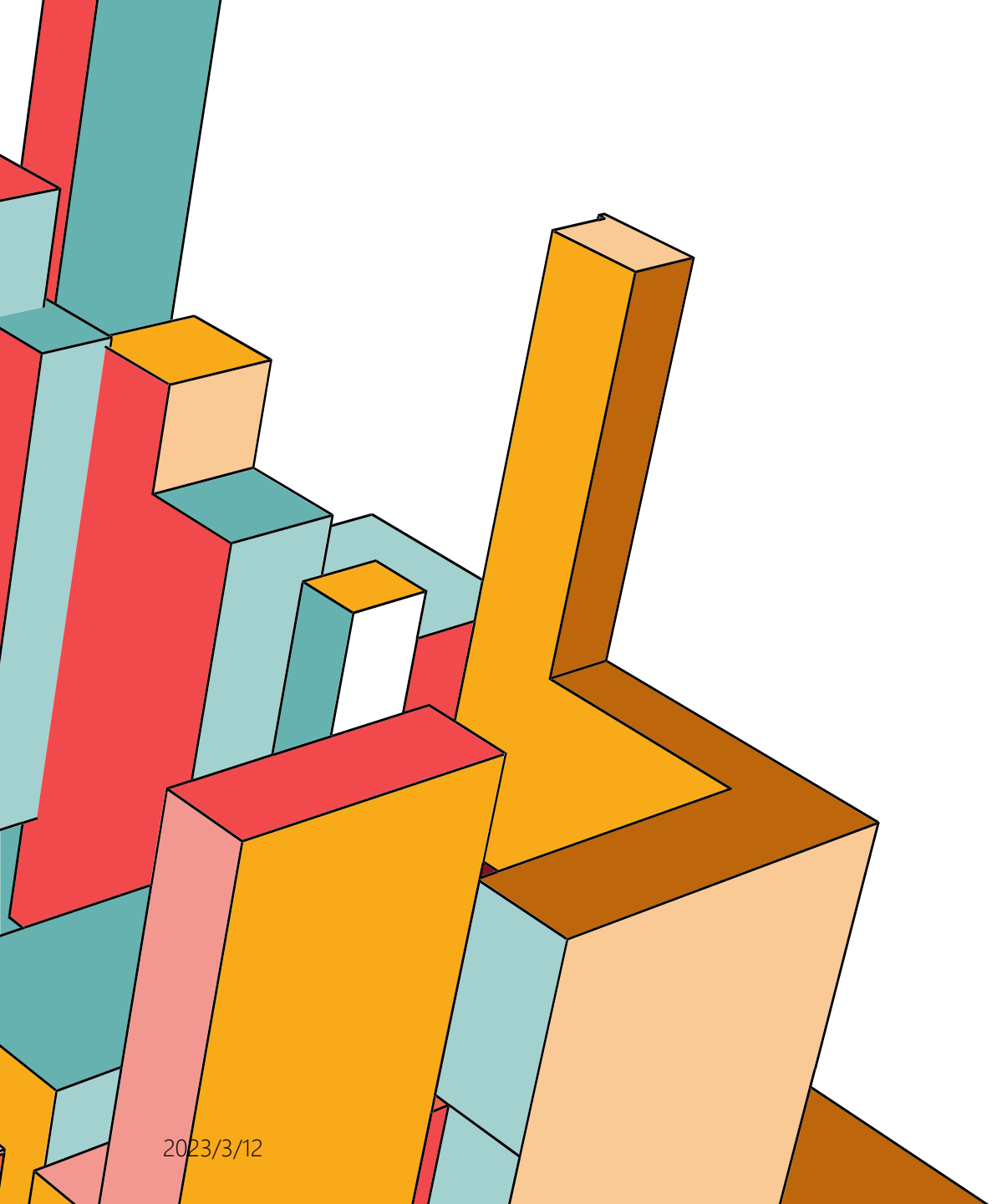
https://livebook.manning.com/book/the-design-of-web-apis/chapter-1/26

COMPONENT-LEVEL DESIGN

The component-level design for software fully describes the internal detail of each software component.

- **Data structures** for all local data objects
- **Algorithmic details** for all processing that occurs within a component
- **Interfaces** that allow access to all component operations (behaviors)





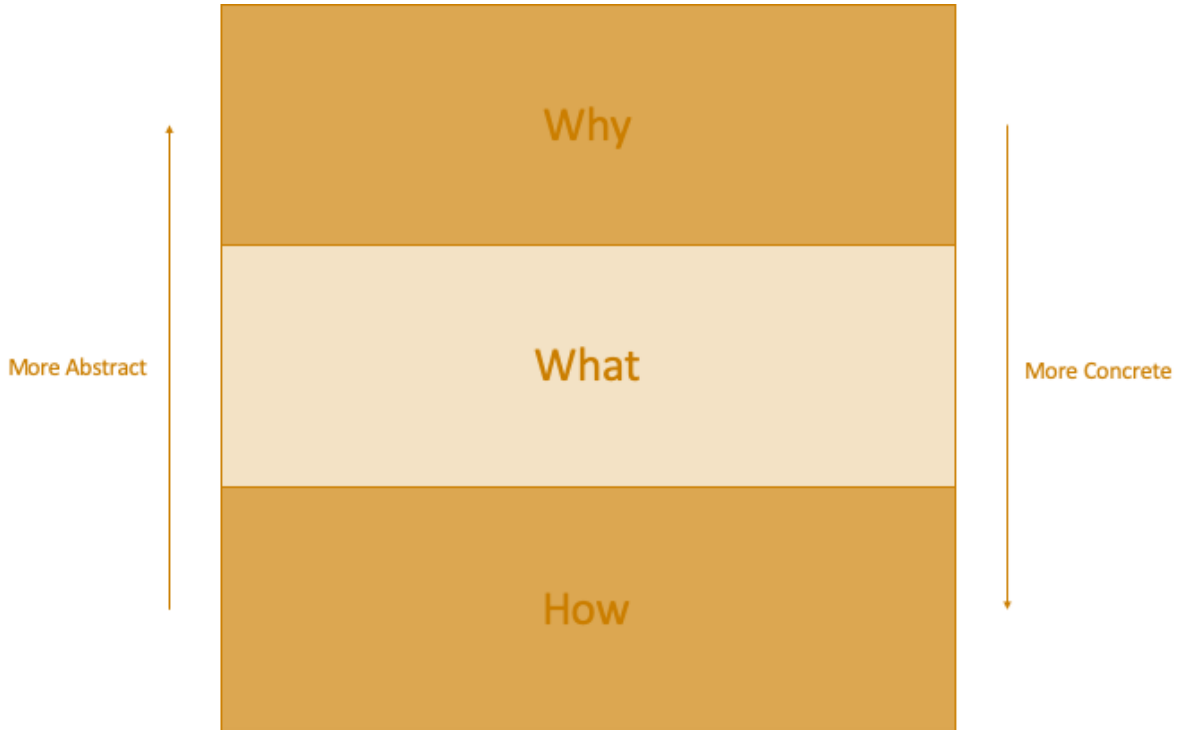
DESIGN CONCEPTS

"The beginning of wisdom for a software engineer is to recognize the difference between getting a program to work, and getting it right."

- M. A. Jackson

Fundamental software design concepts provide the necessary framework for "**getting it right**."

DESIGN CONCEPTS - ABSTRACTION

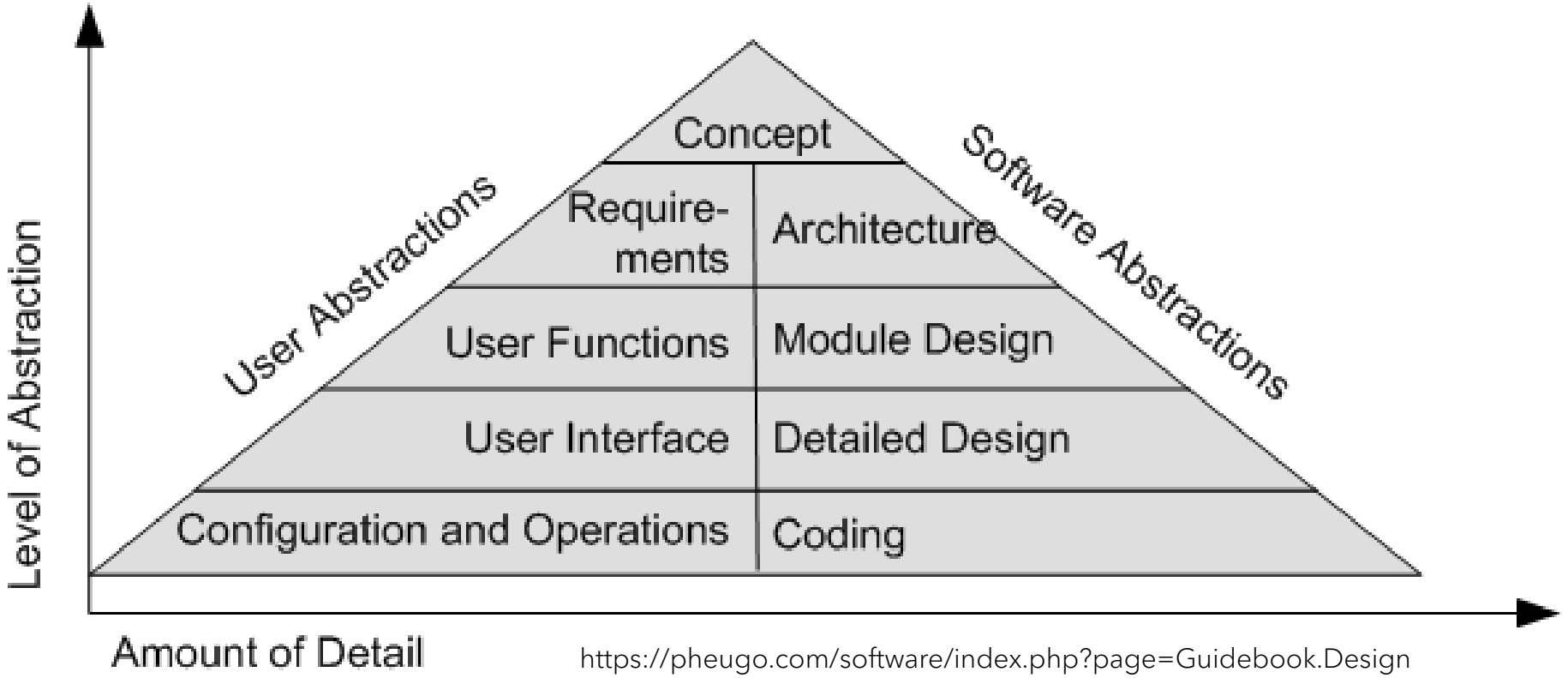


At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment

At lower levels of abstraction, a more detailed description of the solution is provided.

At the lowest level of abstraction, the solution is stated in a manner that can be directly implemented.

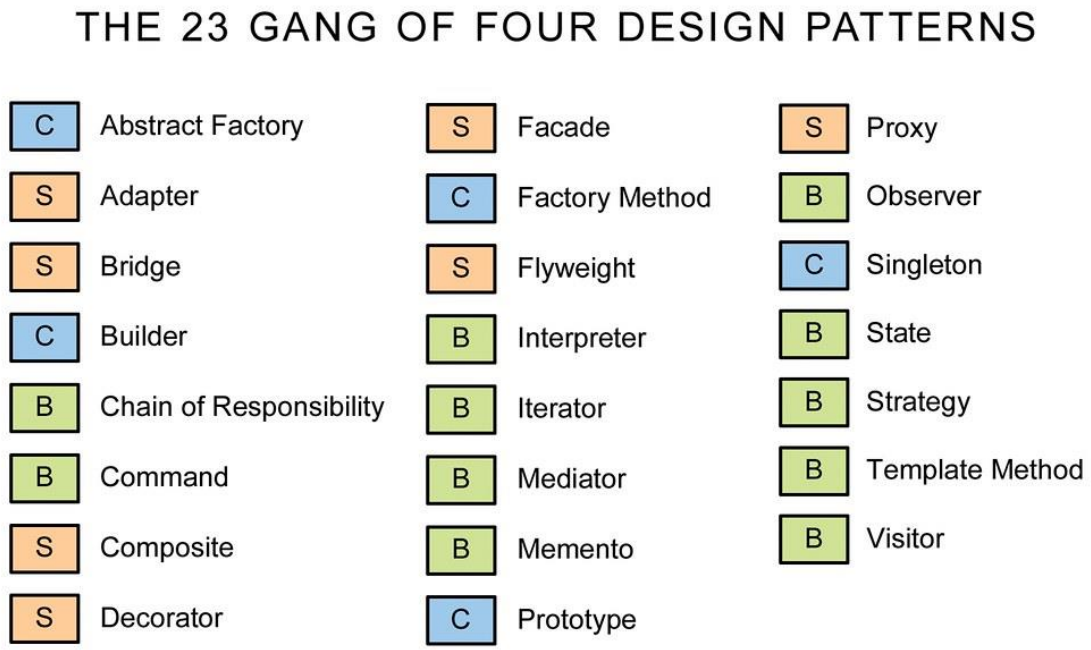
DESIGN CONCEPTS - ABSTRACTION



https://pheugo.com/software/index.php?page=Guidebook.Design

DESIGN CONCEPTS - PATTERNS

- A software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design.
- It is a description or template for how to solve a problem that can be used in many different situations.
- Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.



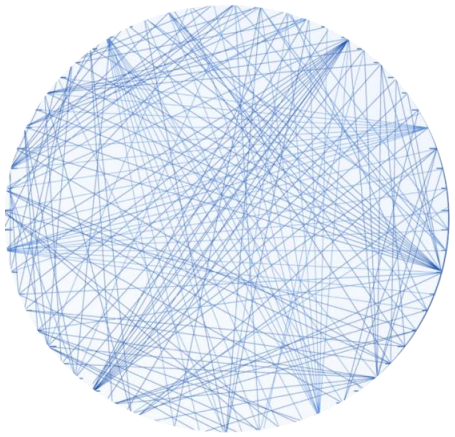
Gangs of Four Design Patterns is the collection of 23 design patterns from the book "Design Patterns: Elements of Reusable Object-Oriented Software".

DESIGN CONCEPTS - SEPARATION OF CONCERNS

- Separation of concerns (关注点分离) is a design concept that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently.
- A concern is a feature or behavior that is specified as part of the requirements model for the software.
- By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.

DESIGN CONCEPTS - MODULARITY

- Modularity (模块化) is the most common manifestation of separation of concerns.
- Software is divided into separately named and addressable components, sometimes called modules, that are integrated to satisfy problem requirements.

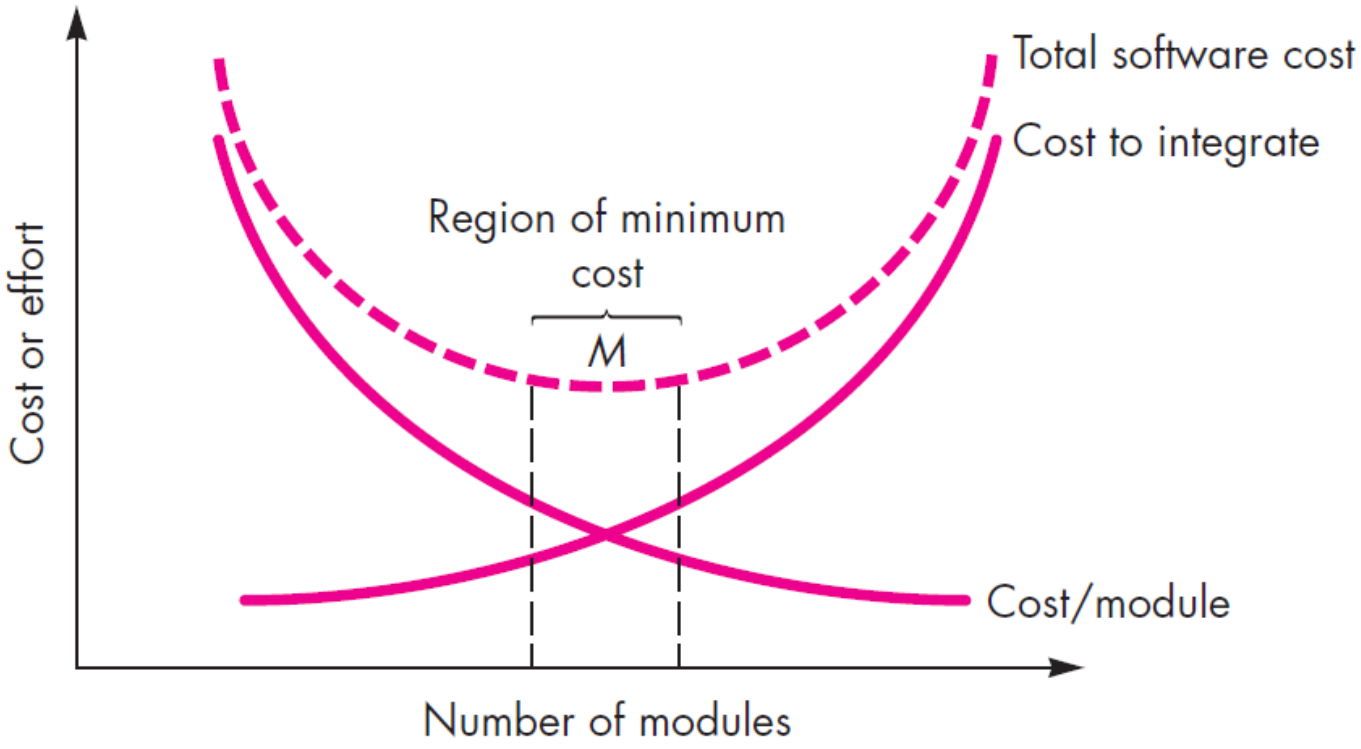


Non-modular

Modular

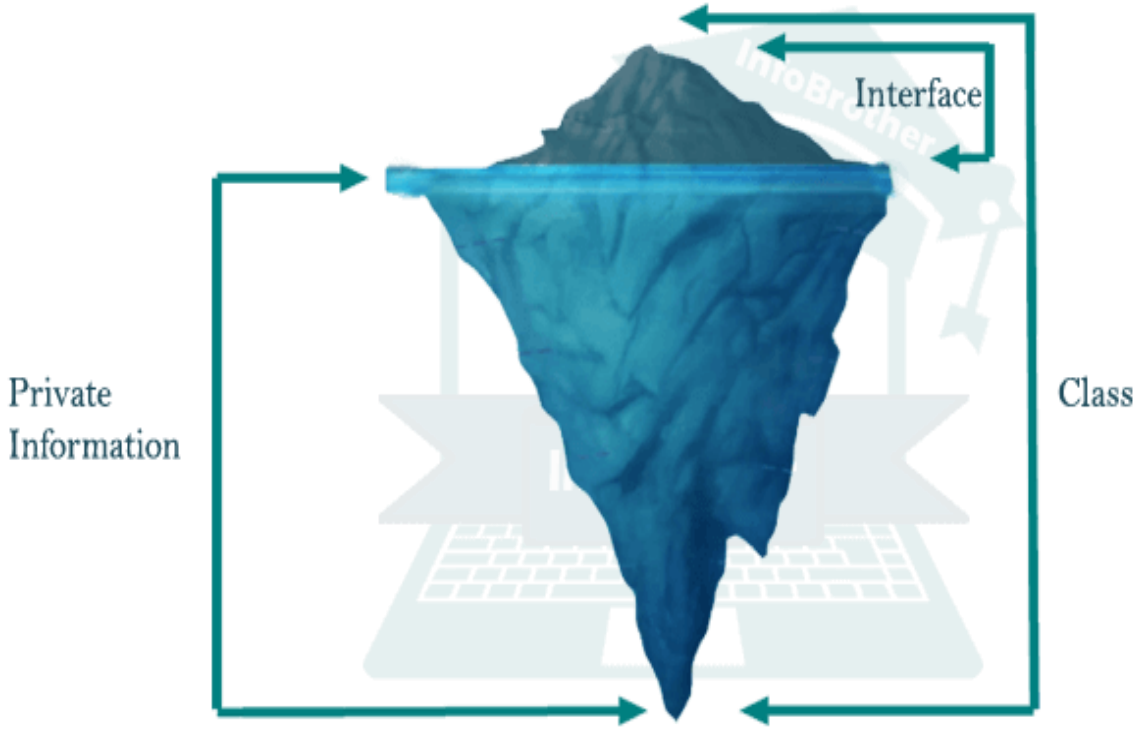
DESIGN CONCEPTS - MODULARITY

- The cost to develop an individual software module decrease as the total number of modules increases
- However, as the number of modules grows, the cost associated with integrating the modules also grows.
- We should avoid undermodularity and overmodularity



DESIGN CONCEPTS - INFORMATION HIDING

- The principle of information hiding suggests that modules be "characterized by design decisions that (each) hides from all others.“
- In other words, modules should be specified and designed so that information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information.

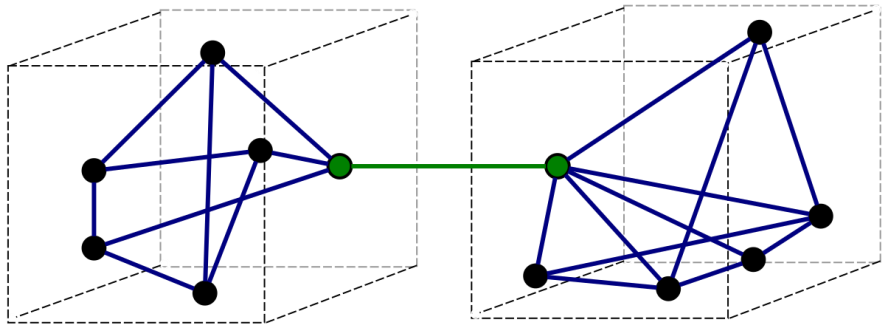


DESIGN CONCEPTS - FUNCTIONAL INDEPENDENCE

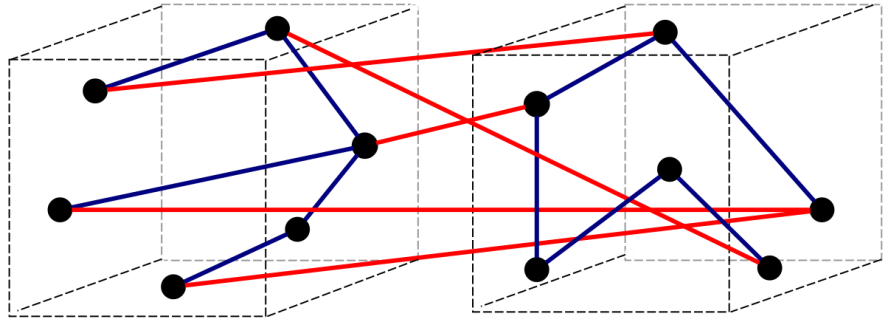
- The concept of functional independence is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding
- Functional independence is achieved by developing modules with “single-minded” function and an “aversion” to excessive interaction with other modules.
- Software with effective modularity or independent modules, is easier to develop, test, and maintain

DESIGN CONCEPTS - FUNCTIONAL INDEPENDENCE

- Independence is assessed using two qualitative criteria: cohesion and coupling
- Cohesion (内聚) indicates the relationship within the module and represents the functional strength of modules
- Coupling (耦合) indicates the relationship between modules and represents the independence among modules

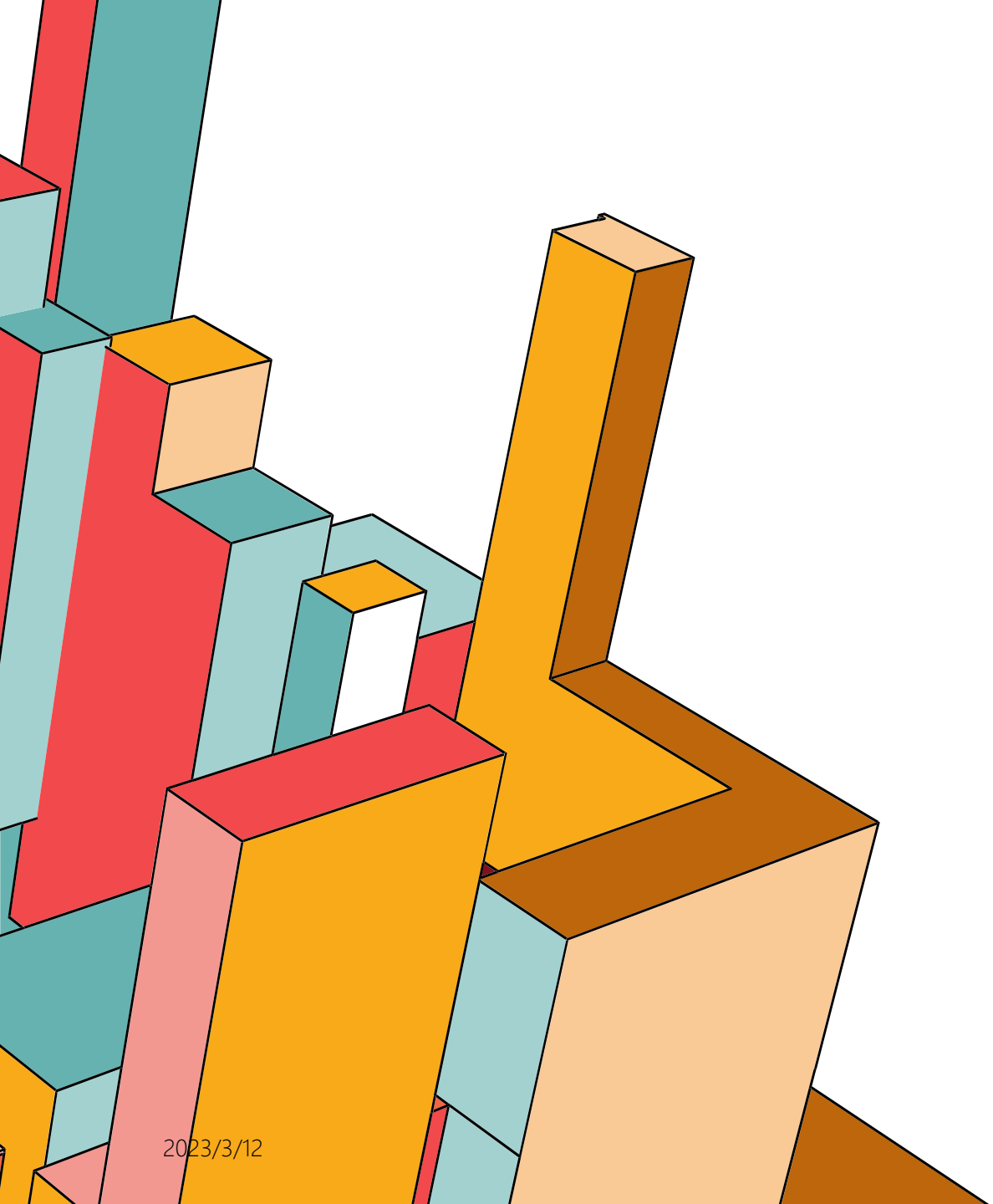


a) Good (loose coupling, high cohesion)



b) Bad (high coupling, low cohesion)

https://en.wikipedia.org/wiki/Coupling_%28computer_programming%29

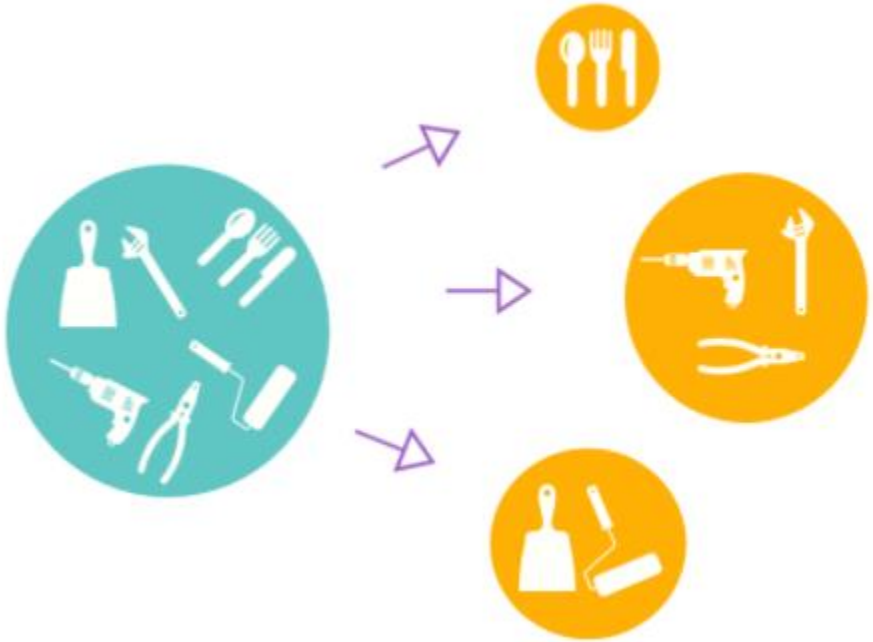


OO DESIGN CONCEPTS (SOLID)

- **S**ingle Responsibility Principle
- **O**pen Closed Principle
- **L**iskov Substitution Principle
- **L**aw of Demeter
- **I**nterface Segregation Principle
- **D**ependence Inversion Principle

SINGLE RESPONSIBILITY PRINCIPLE

- Every class, module, or function in a program should have one responsibility/purpose in a program.
- Every class, module, or function should have only one reason to change



SINGLE RESPONSIBILITY PRINCIPLE

```
public class Order{
   …
   //根据订单金额和折扣规则计算需要支付的金额
   public float calculateAmountToPay(){
      …
   }
}
```

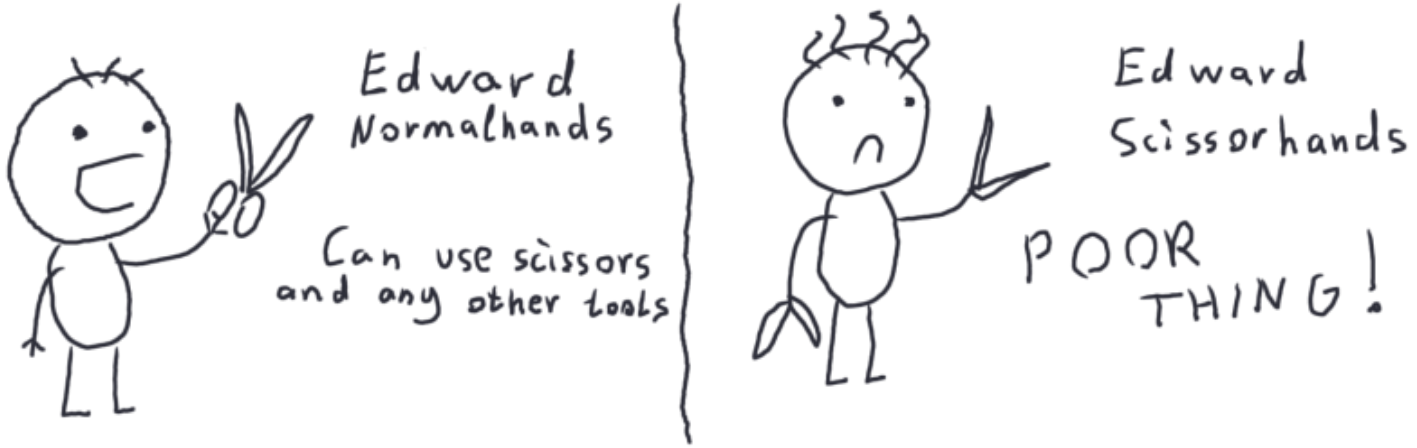
No single responsibility: changes to total and discount both affect this function

```
public class Order{
   …
   //计算订单总金额
   public float calculateTotal(){
      …
   }
   //计算折扣金额
   public float calculateDiscount(){
      …
   }
}
```

High cohesion for functions

OPEN CLOSED PRINCIPLE

- Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification
- The entity should be extendable without modifying itself
- You should write your modules in a way that wouldn't require you to modify it's code in order to extend it's behavior.



https://dev.to/satansdeer/openclosed-principle-86a

OPEN CLOSED PRINCIPLE

```
public void drawFigure(Shape[] shapeList){
  …
  for(int i=0;i<shapeList.length();i++){
    //如果是圆形
    if(shapeList[i].type.equals("circle")){
      …
    }
    //如果是矩形
    if(shapeList[i].type.equals("rectangle")){
      …
    }
  }
}
```

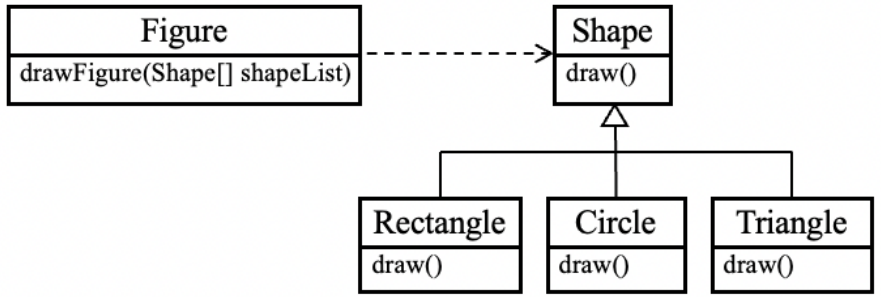
×

```
public void drawFigure(Shape[] shapeList){
  …
  for(int i=0;i<shapeList.length();i++){
    //调用图形类的抽象绘制方法
    shapeList[i].draw();
  }
}
```

```
public abstract class Shape{
  //图形绘制的抽象方法
  public abstract void draw();
  …
}
```

```
public class Circle extends Shape{
  //重写父类的图形绘制方法，提供针对圆形的具体实现
  public void draw(){
    …
  }
  …
}
```

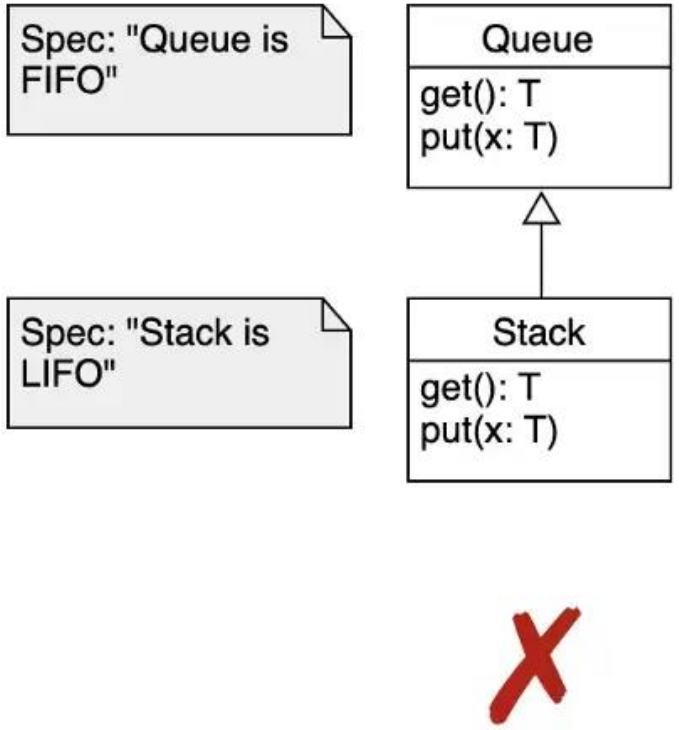
√



LISKOV SUBSTITUTION PRINCIPLE

- The Liskov Substitution Principle (里式替换原则) states that objects of a superclass should be replaceable with objects of its subclasses without breaking the application.
- Objects of subclasses should behave in the same way as the objects of superclass.
- The Liskov Substitution Principle is supported by object-oriented design abstraction concepts of inheritance and polymorphism.

LISKOV SUBSTITUTION PRINCIPLE



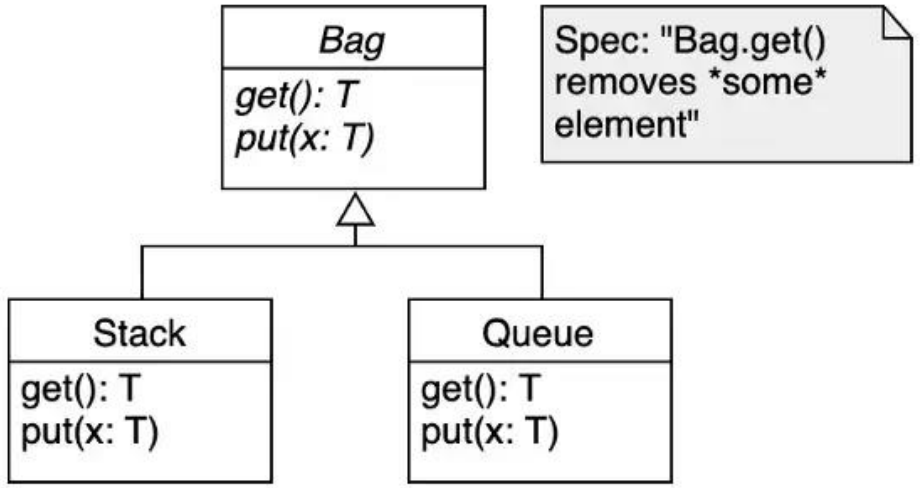
- This example violates behavioral subtyping because type Stack is not a behavioral subtype of type Queue: it is not the case that the behaviors allowed by the specification of Stack are also allowed by the specification of Queue.
- Clients accessing a Stack object through a reference of type Queue would, based on Queue's documentation, expect FIFO behavior but observe LIFO behavior, invalidating these clients' correctness proofs and potentially leading to incorrect behavior of the program as a whole.

https://medium.com/@bart.jacobs/liskov-substitution-principle-a-misnomer-3a891c29d359

LISKOV SUBSTITUTION PRINCIPLE

- A program where both Stack and Queue are subclasses of a type Bag, whose specification for get is merely that it removes some element, does satisfy behavioral subtyping and allows clients to safely reason about correctness based on the presumed types of the objects they interact with.

- Any object that satisfies the Stack or Queue specification also satisfies the Bag specification.

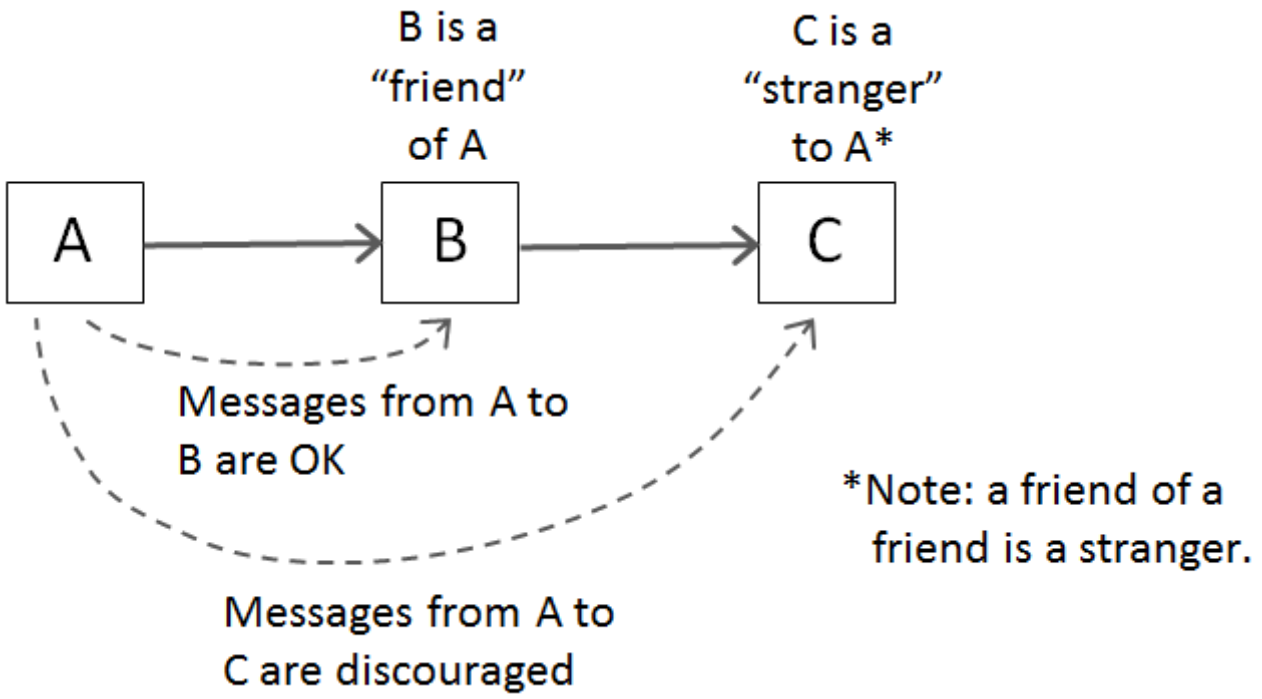


https://medium.com/@bart.jacobs/liskov-substitution-principle-a-misnomer-3a891c29d359

LAW OF DEMETER

The Law of Demeter (LoD), or principle of least knowledge, is a specific case of loose coupling.

- Each unit should have only limited knowledge about other units: only units "closely" related to the current unit.
- Each unit should only talk to its friends; don't talk to strangers.
- Only talk to your immediate friends.



https://betterprogramming.pub/demeters-law-don-t-talk-to-strangers-87bb4af11694

LAW OF DEMETER

```
//展示一本图书的信息
public void displayBook(Book book){
   …
   //展示图书作者信息
   Author author=book.getAuthor();
   display(author.getName());
   …
}
```

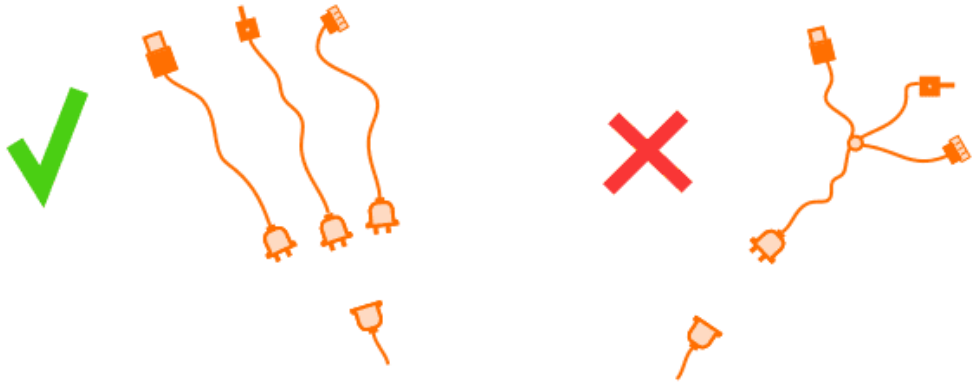
×

```
//展示一本图书的信息
public void displayBook(Book book){
   …
   //展示图书作者信息.
   display(book.getAuthorName());
   …
}
```

√

INTERFACE SEGREGATION PRINCIPLE

- The Interface Segregation Principle (ISP) states that a client should not be exposed to methods it doesn't need.
- Declaring methods in an interface that the client doesn't need pollutes the interface and leads to a “bulky” or “fat” interface, which should be avoided



https://accesto.com/blog/solid-php-solid-principles-in-php/

INTERFACE SEGREGATION PRINCIPLE

```php
interface Exportable
{
    public function getPDF();
    public function getCSV();
}
```

```php
class Invoice implements Exportable
{
    public function getPDF() {
        // ...
    }
    public function getCSV() {
        // ...
    }
}
```

```php
class CreditNote implements Exportable
{
    public function getPDF() {
        throw new \NotUsedFeatureException();
    }
    public function getCSV() {
        // ...
    }
}
```

https://accesto.com/blog/solid-php-solid-principles-in-php/

```php
interface ExportablePdf
{
    public function getPDF();
}
```

```php
interface ExportableCSV
{
    public function getCSV();
}
```

https://accesto.com/blog/solid-php-solid-principles-in-php/

```php
class Invoice implements ExportablePdf, ExportableCSV
{
    public function getPDF() {
        //
    }
    public function getCSV() {
        //
    }
}
```

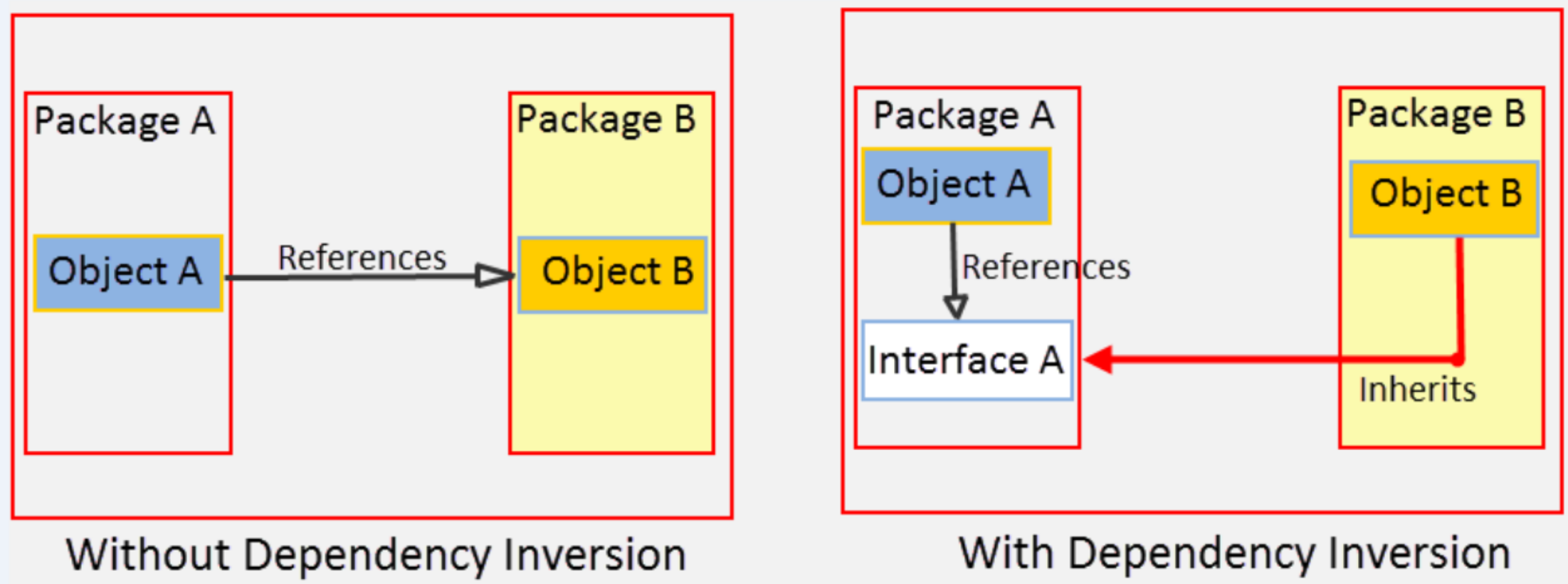
```php
class CreditNote implements ExportableCSV
{
    public function getCSV() {
        //
    }
}
```

DEPENDENCE INVERSION PRINCIPLE

- The Dependence inversion principle states:
  - High-level modules should not import anything from low-level modules. Both should depend on abstractions (e.g., interfaces).
  - Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.

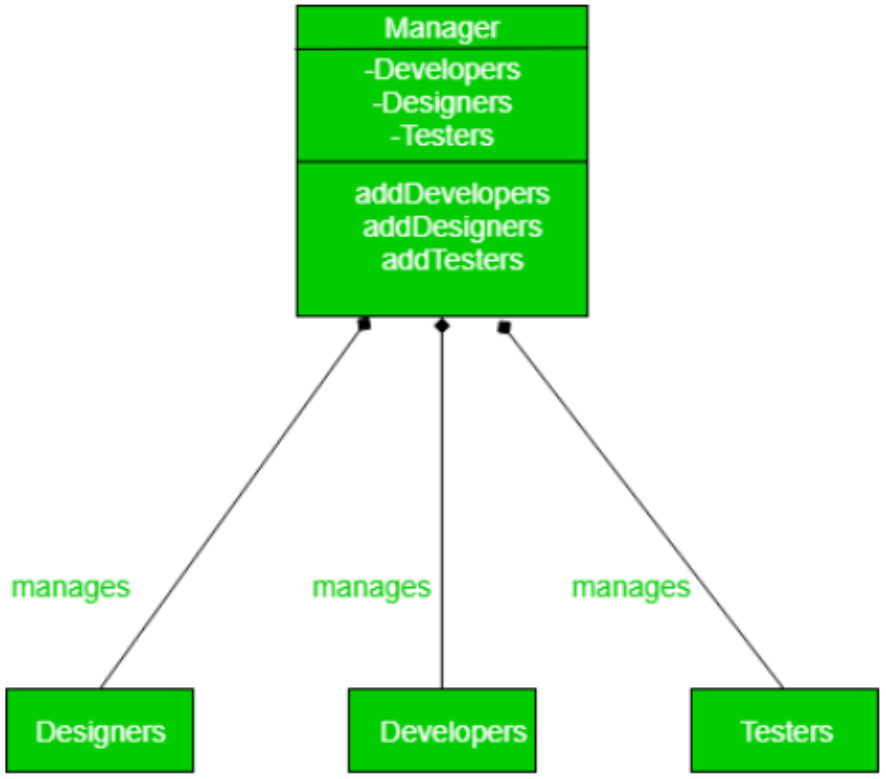
- In other words, we should reduce dependencies to specific implementations, but rely on interfaces.

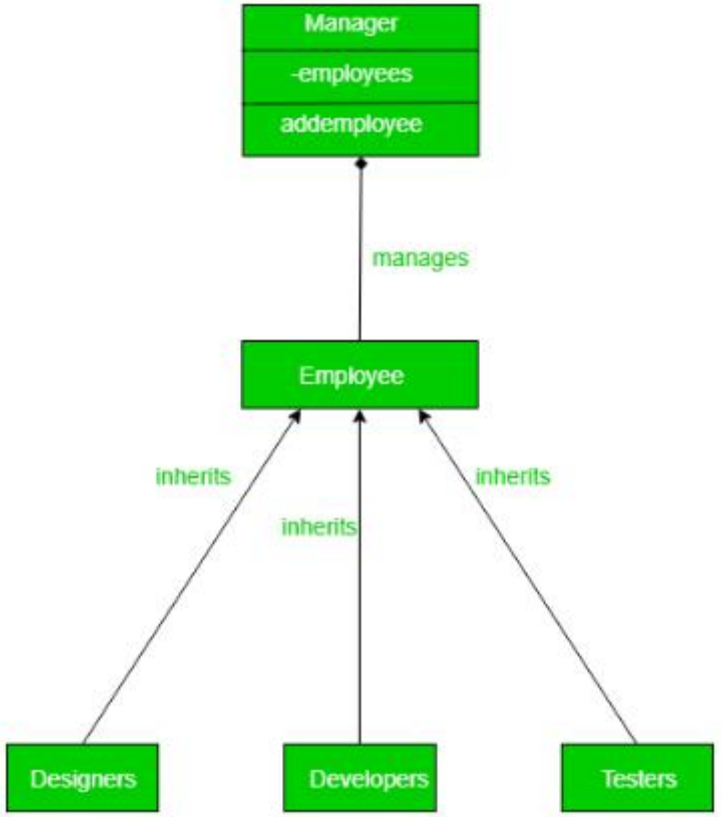
DEPENDENCE INVERSION PRINCIPLE



https://springframework.guru/principles-of-object-oriented-design/dependency-inversion-principle/

DEPENDENCE INVERSION PRINCIPLE



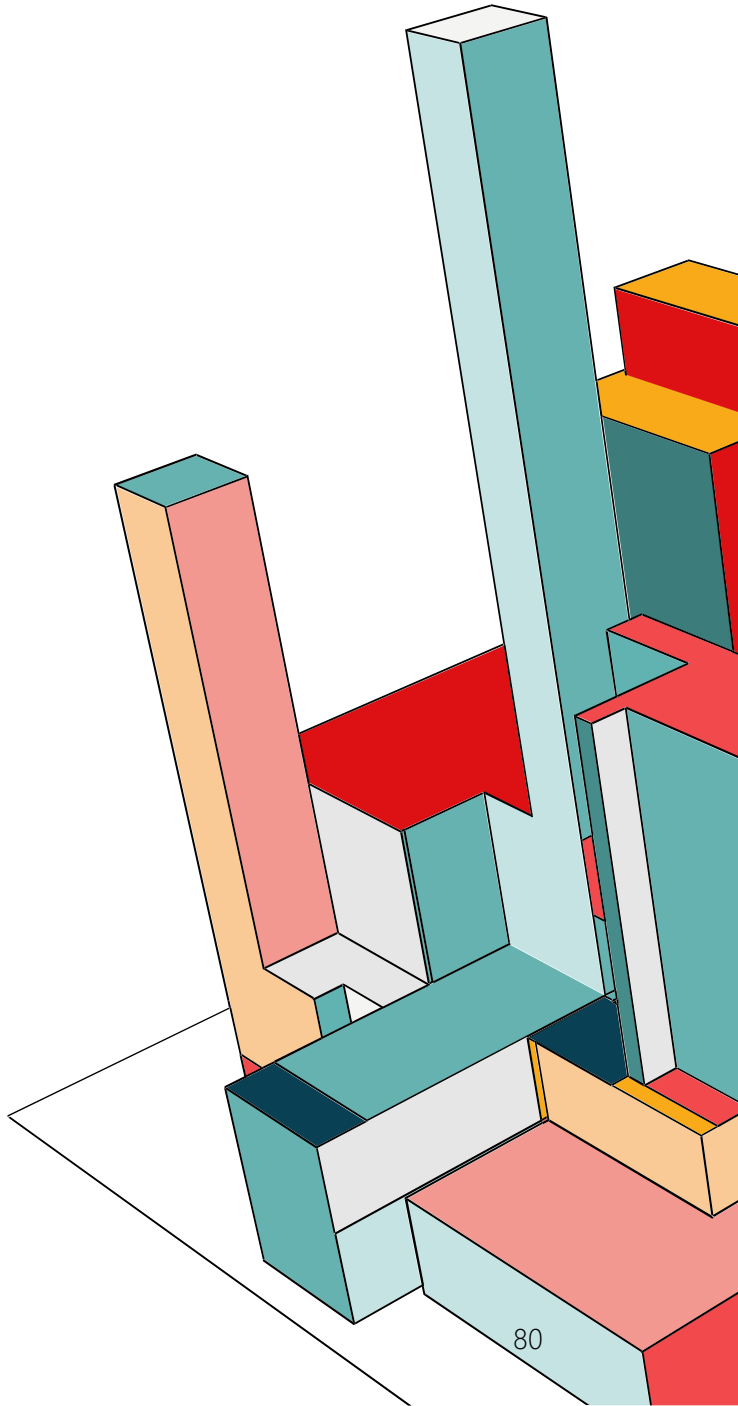


https://www.geeksforgeeks.org/dependecy-inversion-principle-solid/

READINGS

- Chapter 12-13. Software Engineering A Practitioner's Approach by Roger Pressman, 8th edition.

- 第7章 软件体系结构. 现代软件工程基础 by 彭鑫 et al.



NEXT

- Build Systems