



CSE- 321

Software Engineering

Software Design

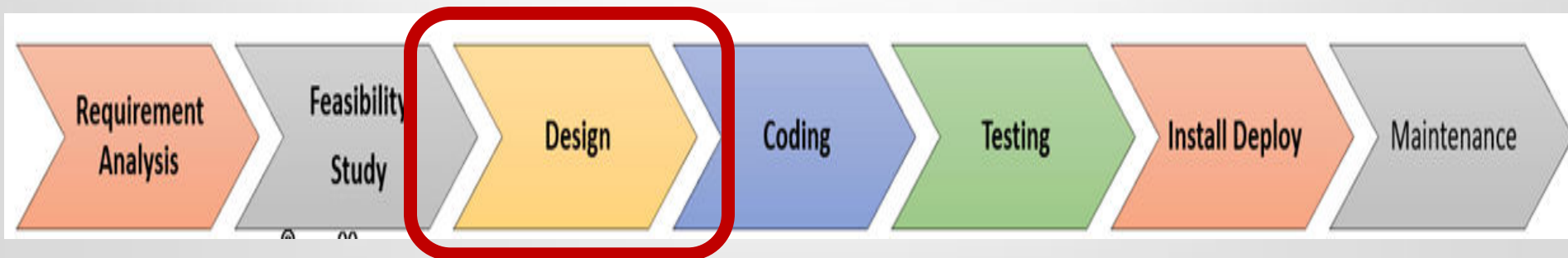
Lecture Outlines

- ✧ **Software design**
- ✧ **Design Principles**
- ✧ **Strategy of Design**
- ✧ **Coupling and Cohesion**
- ✧ **Architectural Design**
- ✧ **Abstract machine (layered) Design**
- ✧ **Distributed Systems Architectures**
- ✧ **Client-Server Architecture**
- ✧ **Broker Architectural Style : CORBA**
- ✧ **Service-Oriented Architecture (SOA)**

Software design

What is Software design ?

- ❖ Software design is a process to **transform user requirements** into some suitable form, which helps the programmer in **software coding and implementation**.
- ❖ Software design is the first step in SDLC (Software Design Life Cycle), which moves the concentration from **problem domain to solution domain**. It tries to specify how to fulfill the requirements mentioned in SRS.



Objectives of Software Design



Objectives of Software Design

Following are the purposes of Software design:

- 1. Correctness:** Software design should be correct as per requirement.
- 2. Completeness:** The design should have all components like data structures, modules, and external interfaces, etc.
- 3. Efficiency:** Resources should be used efficiently by the program.
- 4. Flexibility:** Able to modify on changing needs.
- 5. Consistency:** There should not be any inconsistency in the design.
- 6. Maintainability:** The design should be so simple so that it can be easily maintainable by other designers.

Software design yields three levels of results:

Architectural Design

- Highest abstract version of the system.
- It identifies the software as a system with many components interacting with each other.
- Designers get the idea of proposed solution domain.

High-level Design

- Breaks the 'single entity-multiple component' concept of architectural design into less-abstracted view of sub-systems and modules and depicts their interaction with each other.

Detailed Design

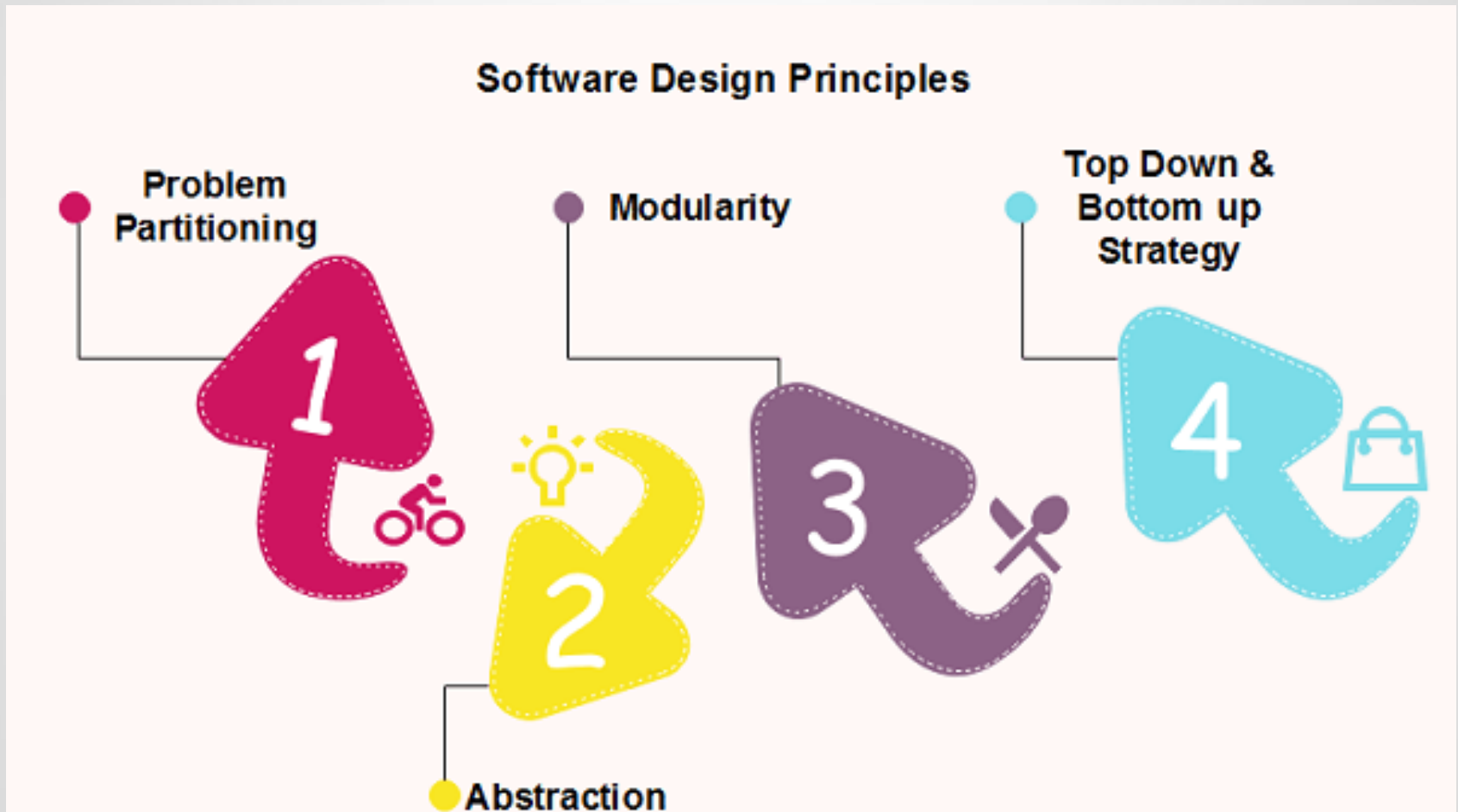
- Detailed design deals with the implementation part of what is seen as a system and its sub-systems in the previous two designs

Software Design Principles

- Design **is not coding**, coding **is not design**.
- The design should be **traceable** to the analysis model.
- The design should **not reinvent the wheel**.
- The design should “**minimize the intellectual distance**” between the software and the problem as it exists in the real world.
- The design should exhibit **uniformity and integration**.
- The design should be **structured to accommodate change**.
- The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered.
- The design should be assessed for quality as it is being created, not after the fact.
- The design should be reviewed to minimize conceptual (semantic) errors.

Software Design Principles

Software design principles are concerned with providing means to handle the complexity of the design process effectively.



Software Design Principles

Problem Partitioning

For small problem, we can handle the entire problem at once but for the significant problem, divide the problems and conquer the problem it means to divide the problem into smaller pieces so that each piece can be captured separately.

Benefits of Problem Partitioning

1. Software is easy to understand
2. Software becomes simple
3. Software is easy to test
4. Software is easy to modify
5. Software is easy to maintain
6. Software is easy to expand

As the number of partition increases = Cost of partition and complexity increases

Modularization

Modularization is a technique to divide a software system into multiple discrete and **independent modules**, which are expected to be capable of carrying out task(s) **independently**. These modules may work as basic constructs for the entire software.

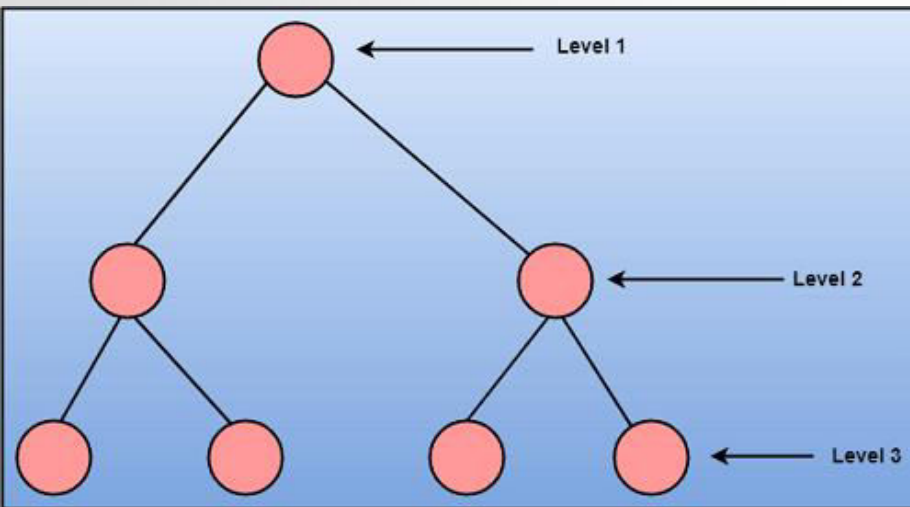
Advantage of modularization:

- Smaller components are easier to maintain
- Program can be divided based on functional aspects
- Desired level of abstraction can be brought in the program
- Components with high cohesion can be re-used again
- Concurrent execution can be made possible
- Desired from security aspect

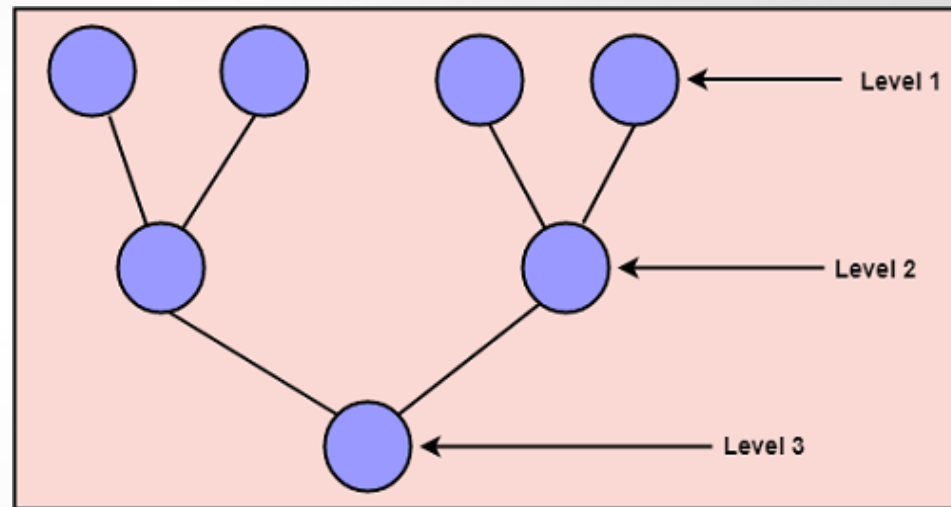
Strategy of Design

To design a system, there are two possible approaches:

1. Top-down Approach
2. Bottom-up Approach



Top-down Approach



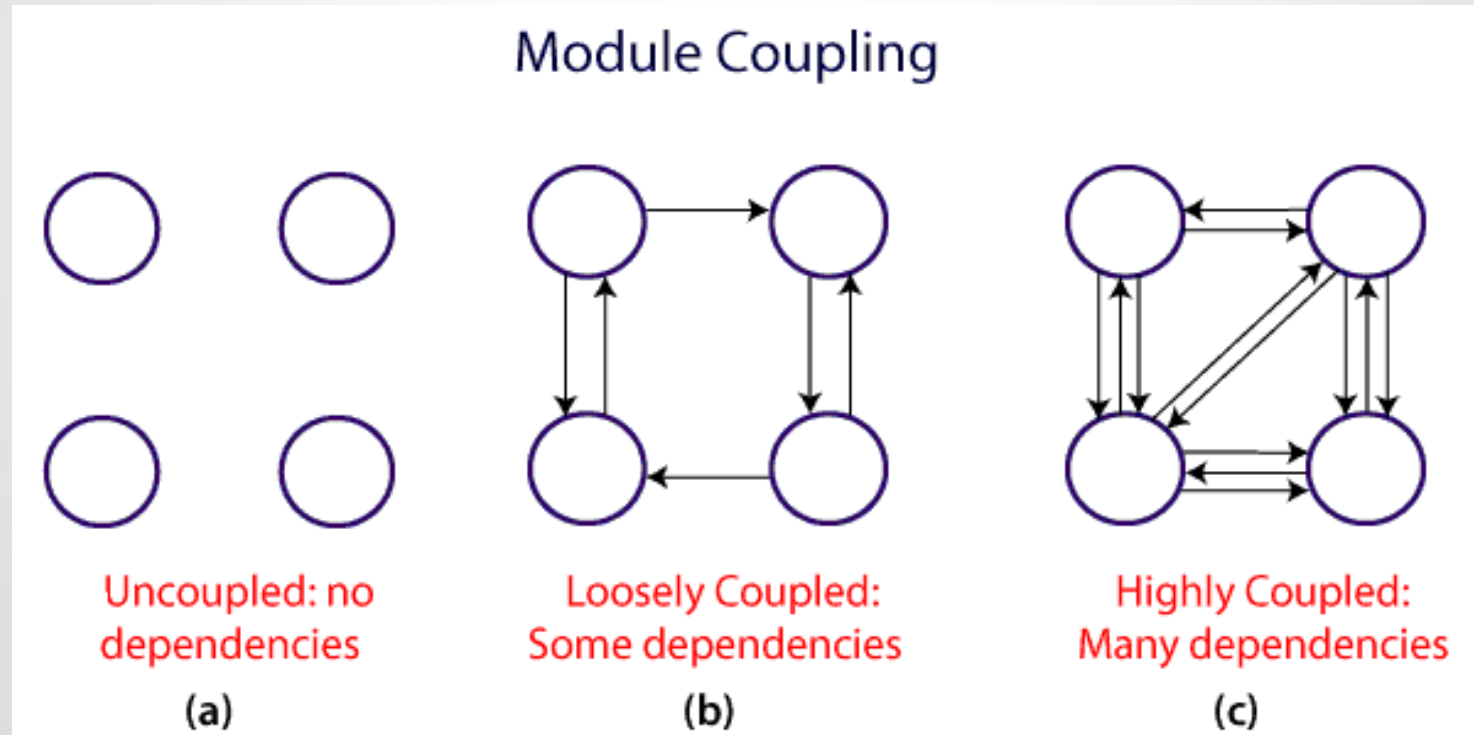
Bottom-up Approach

Coupling and Cohesion

Module Coupling

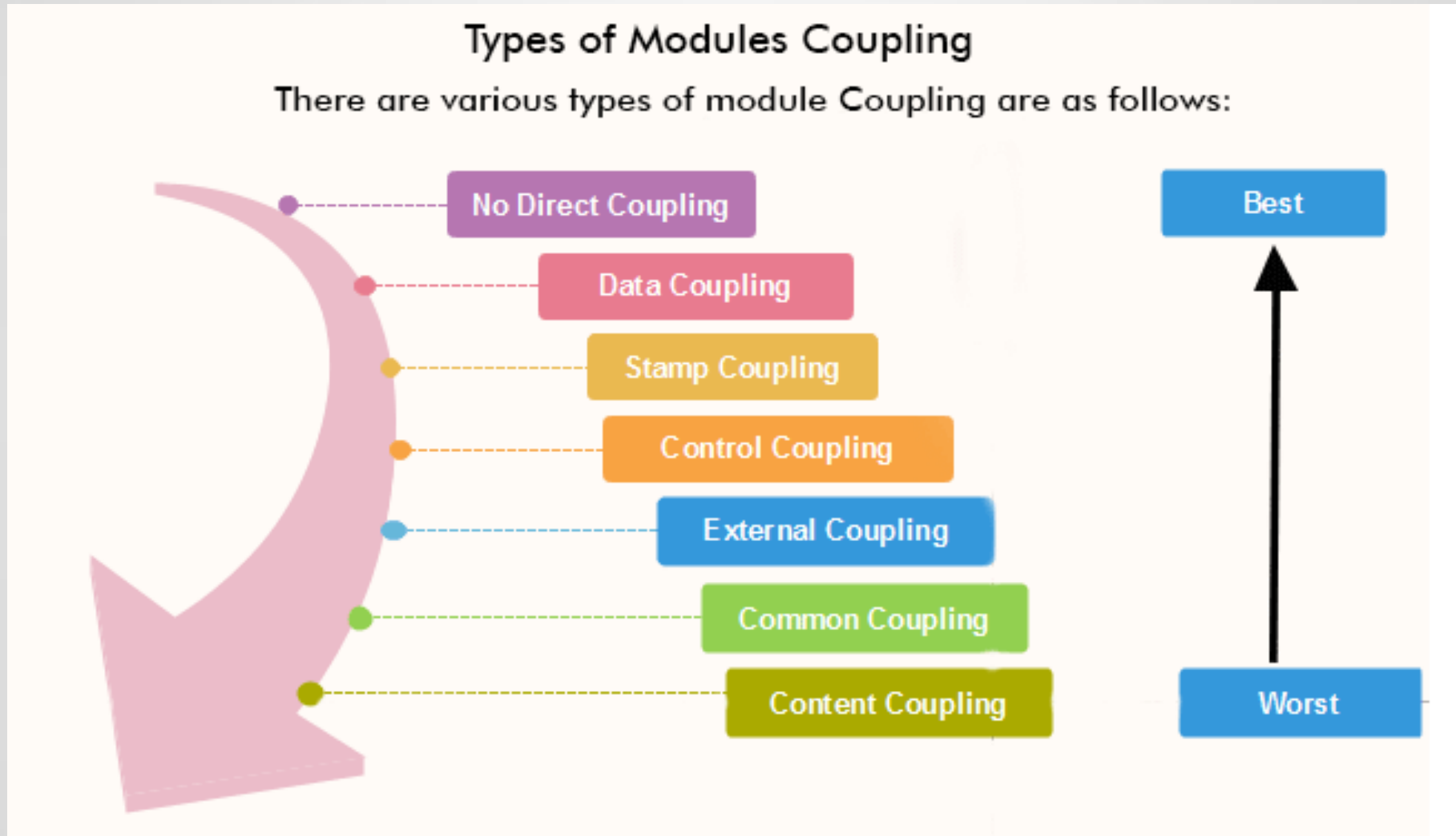
In software engineering, Coupling is measured by the **number of relations between the modules**. That means, the coupling is the **degree of interdependence between software modules**.

A good design is the one that has low coupling.



Coupling and Cohesion

Types of Module Coupling



Coupling and Cohesion

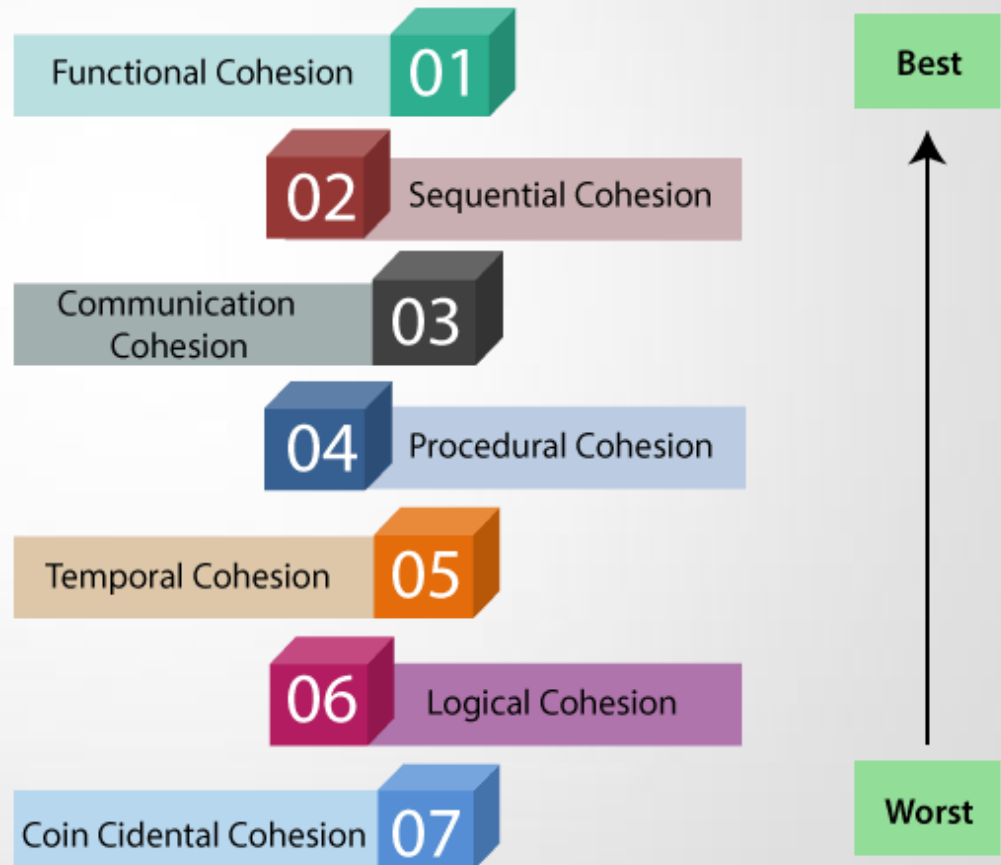
Module Cohesion

cohesion defines to the degree to which the elements of a module belong together. Thus, cohesion measures the **strength of relationships between pieces of functionality within a given module**.

Basically, cohesion is the internal glue that keeps the module together.

A good software design will have high cohesion.

Types of Modules Cohesion



Coupling vs Cohesion

Coupling	Cohesion
Coupling is also called Inter-Module Binding.	Cohesion is also called Intra-Module Binding.
Coupling shows the relationships between modules .	Cohesion shows the relationship within the module .
Coupling shows the relative independence between the modules.	Cohesion shows the module's relative functional strength .
While creating, you should aim for low coupling, i.e., dependency among modules should be less.	While creating you should aim for high cohesion, i.e., a cohesive component/ module focuses on a single function (i.e., single-mindedness) with little interaction with other modules of the system.
In coupling, modules are linked to the other modules.	In cohesion, the module focuses on a single thing.

Design Pattern

- ❖ A design pattern provides a general reusable solution for the common problems occurs in software design.
- ❖ The patterns typically **show relationships and interactions between classes or objects.**
- ❖ The idea is to speed up the development process by providing well tested, proven development/design paradigm. Design patterns are programming language independent strategies for solving a common problem.
- ❖ That means a design pattern represents an idea, not a particular implementation. By using the design patterns you can make your code more flexible, reusable and maintainable.

❖ Types of Design Patterns

There are mainly three types of design patterns:

1. Creational Design Pattern

- Factory Pattern
- Abstract Factory Pattern
- Singleton Pattern
- Prototype Pattern
- Builder Pattern.

2. Structural Design Pattern

- Adapter Pattern
- Bridge Pattern
- Composite Pattern
- Decorator Pattern
- Facade Pattern
- Flyweight Pattern
- Proxy Pattern

3. Behavioural Design Pattern

- Chain Of Responsibility Pattern
- Command Pattern
- Interpreter Pattern
- Iterator Pattern
- Mediator Pattern
- Memento Pattern
- Observer Pattern
- State Pattern
- Strategy Pattern
- Template Pattern
- Visitor Pattern

Difference Between Architectural Style, Architectural Patterns

Architectural Style

The architectural style shows how do we organize our code, or how the system will look like from 10000 feet helicopter view to show the highest level of abstraction of our system design. Furthermore, when building the architectural style of our system we focus on **layers** and **modules** and how they are communicating with each other.

Architectural Patterns

The architectural pattern shows how a solution can be used to solve a reoccurring problem. In another word, it reflects how a code or components **interact** with each other

Architectural Design



Architectural Design

- **Architectural design** is a process for identifying the sub-systems making up a system and the framework for sub-system control and communication.
- The output of this design process is a description of the **software architecture**.
- It represents the link between specification and design processes
- Often carried out in **parallel** with some specification activities.
- It involves identifying major system **components** and their **communications**.

Architectural Design

- A software architecture is a description of how a **software system is organized**.
- Architectural design decisions include decisions on the type of application, the distribution of the system, the architectural styles to be used.
- Architectures may be **documented from several different perspectives** or views such as a conceptual view, a logical view, a process view, and a development view.
- Architectural patterns are a means of reusing knowledge about generic system architectures.
- Architectural patterns are similar to software design pattern but have a broader scope.

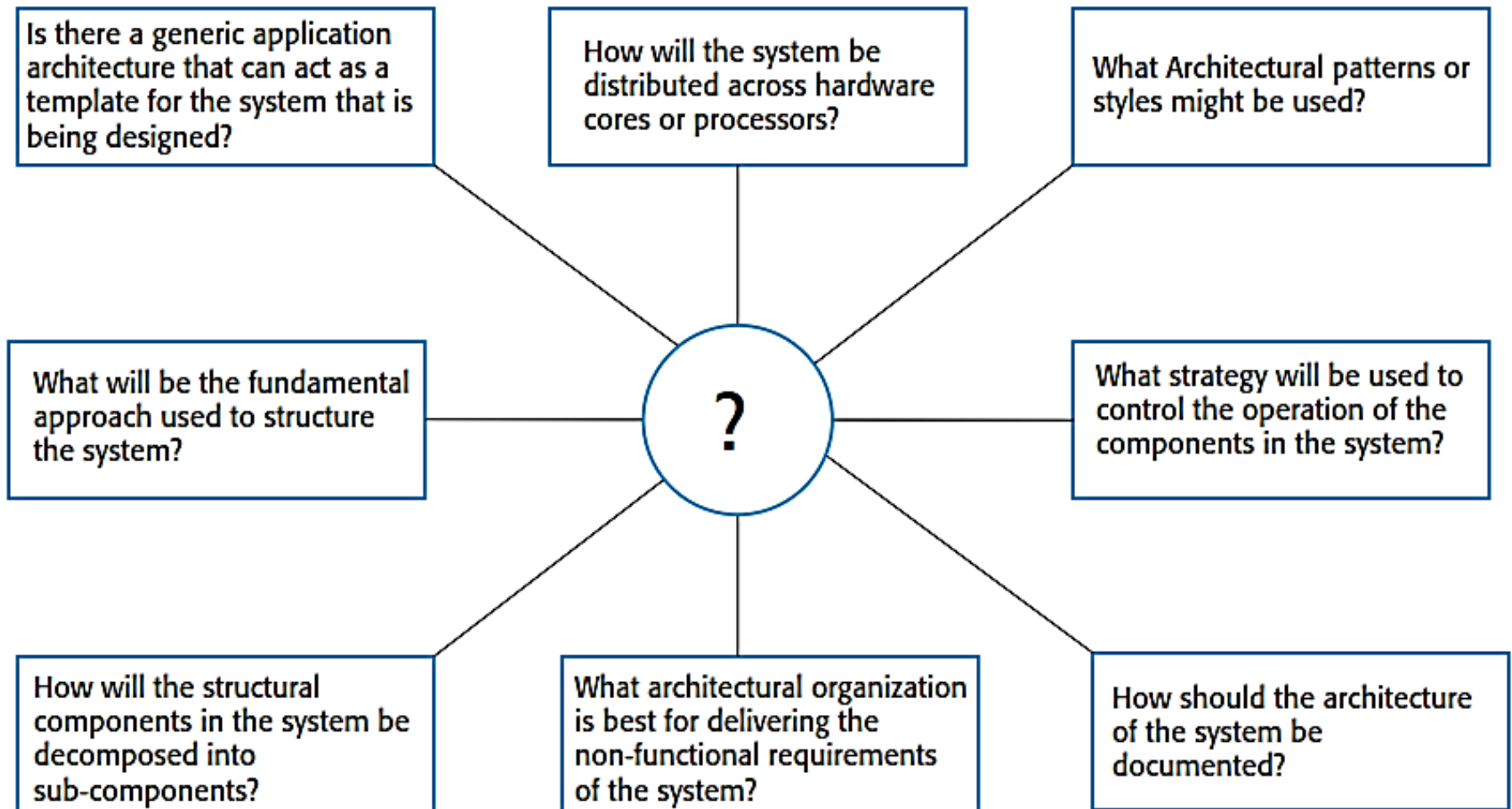
Architectural design decisions

Architectural design is a **creative process** so the process differs depending on the type of system being developed.

However, a number of **common decisions** span all design processes.

- Is there a **generic** application architecture that can be used?
- How will the system be **distributed**?
- What architectural **styles** are appropriate?
- What approach will be used to **structure** the system?
- How will the system be **decomposed** into modules?
- What **control strategy** should be used?
- How will the architectural design be **evaluated**?
- How should the architecture be **documented**?

Architectural design decisions



4 + 1 view model of software architecture

- A **logical view**, which shows the key abstractions in the system as objects or object classes.
- A **process view**, which shows how, at run-time, the system is composed of interacting processes.
- A **development view**, which shows how the software is decomposed for development.
- A **physical view**, which shows the system hardware and how software components are distributed across the processors in the system.
- *Related using use cases or scenarios (+1)*

Architectural patterns

- Patterns are a means of representing, sharing and reusing knowledge.
- An architectural pattern is a **stylized description** of good design practice, which has been tried and tested in different environments.
- Patterns should include **information about** when **they are** and when **they are not useful**.
- Patterns may be represented using tabular and graphical descriptions.

The Model-View-Controller (MVC) pattern

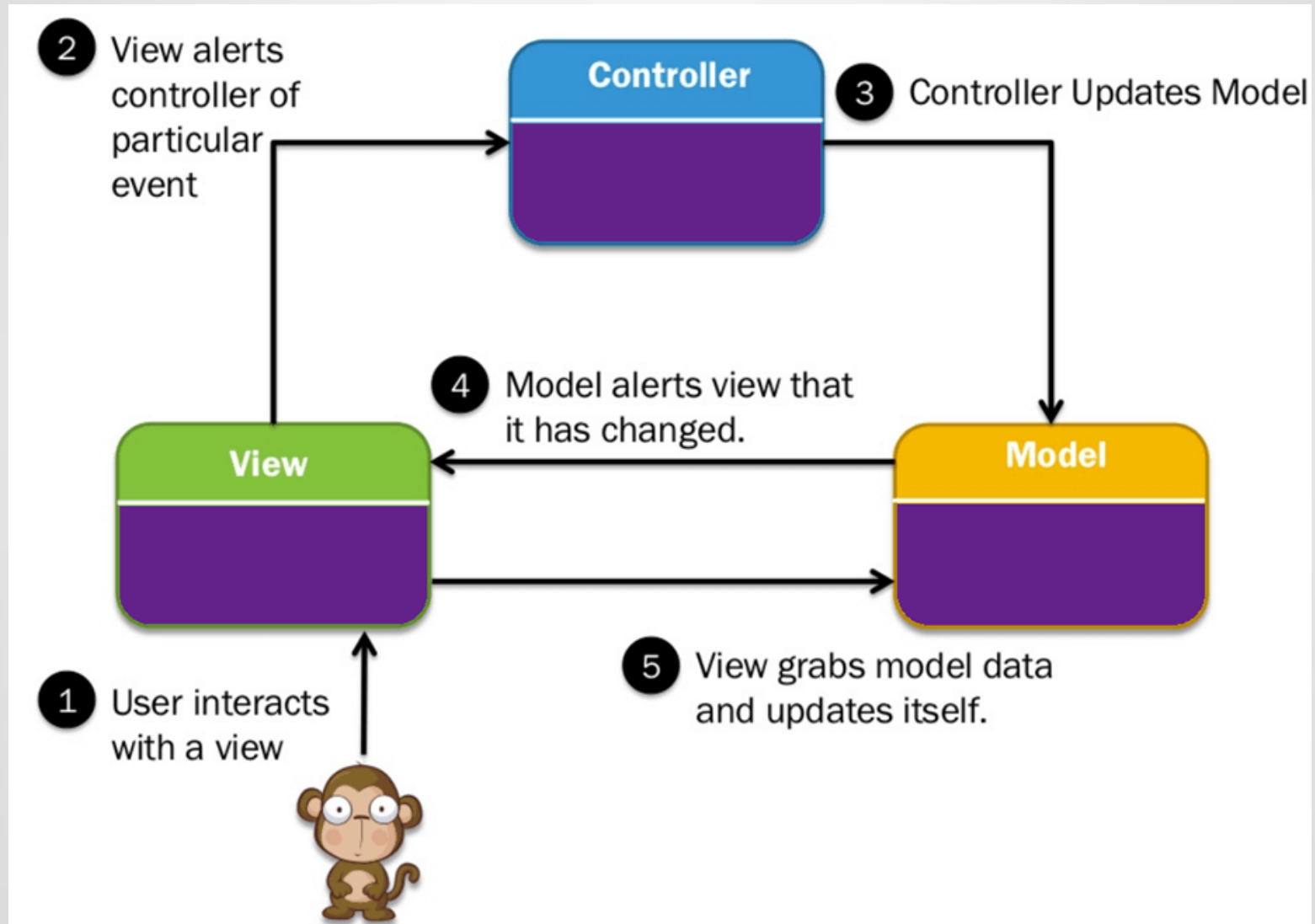
- **MVC** (Model-View-Controller) is a pattern in software design commonly used to implement user interfaces, data, and controlling logic.
- It emphasizes a separation between the software's business logic and display.
- This "separation of concerns" provides for a better division of labour and improved maintenance.
- Some other design patterns are based on MVC, such as
 - **MVVM (Model-View-Viewmodel),**
 - **MVP (Model-View-Presenter), and**
 - **MVW (Model-View-Whatever).**

The Model-View-Controller (MVC) pattern

- Serves as a basis of interaction management in many web-based systems.
- Decouples three major interconnected components:
 - The **model** is the central component of the pattern that directly manages the data, logic and rules of the application. It is the application's dynamic data structure, independent of the user interface.
 - A **view** can be any output representation of information, such as a chart or a diagram. Multiple views of the same information are possible.
 - The **controller** accepts input and converts it to commands for the model or view.
- Supported by most language frameworks.

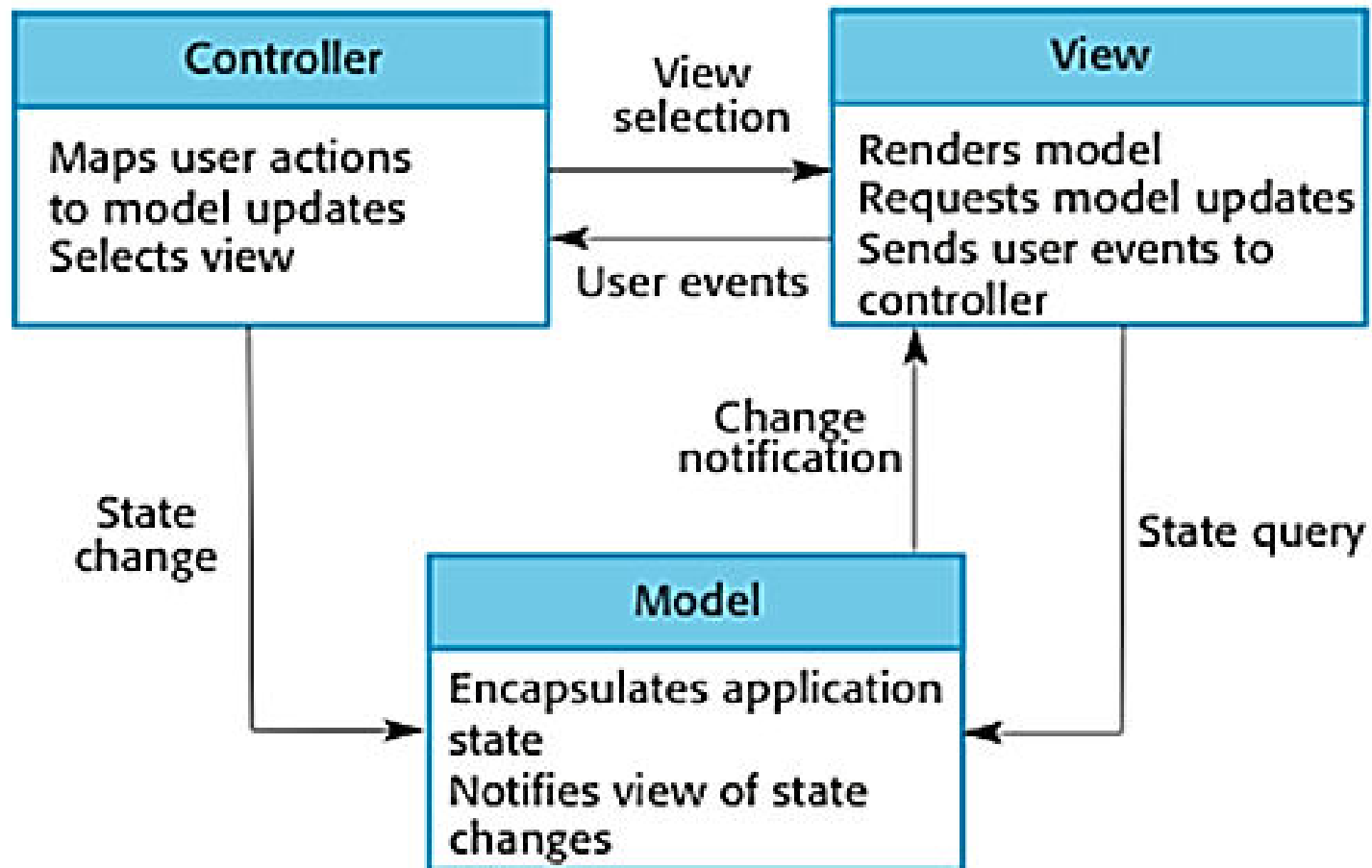
The Model-View-Controller (MVC) pattern

MVC architectural pattern



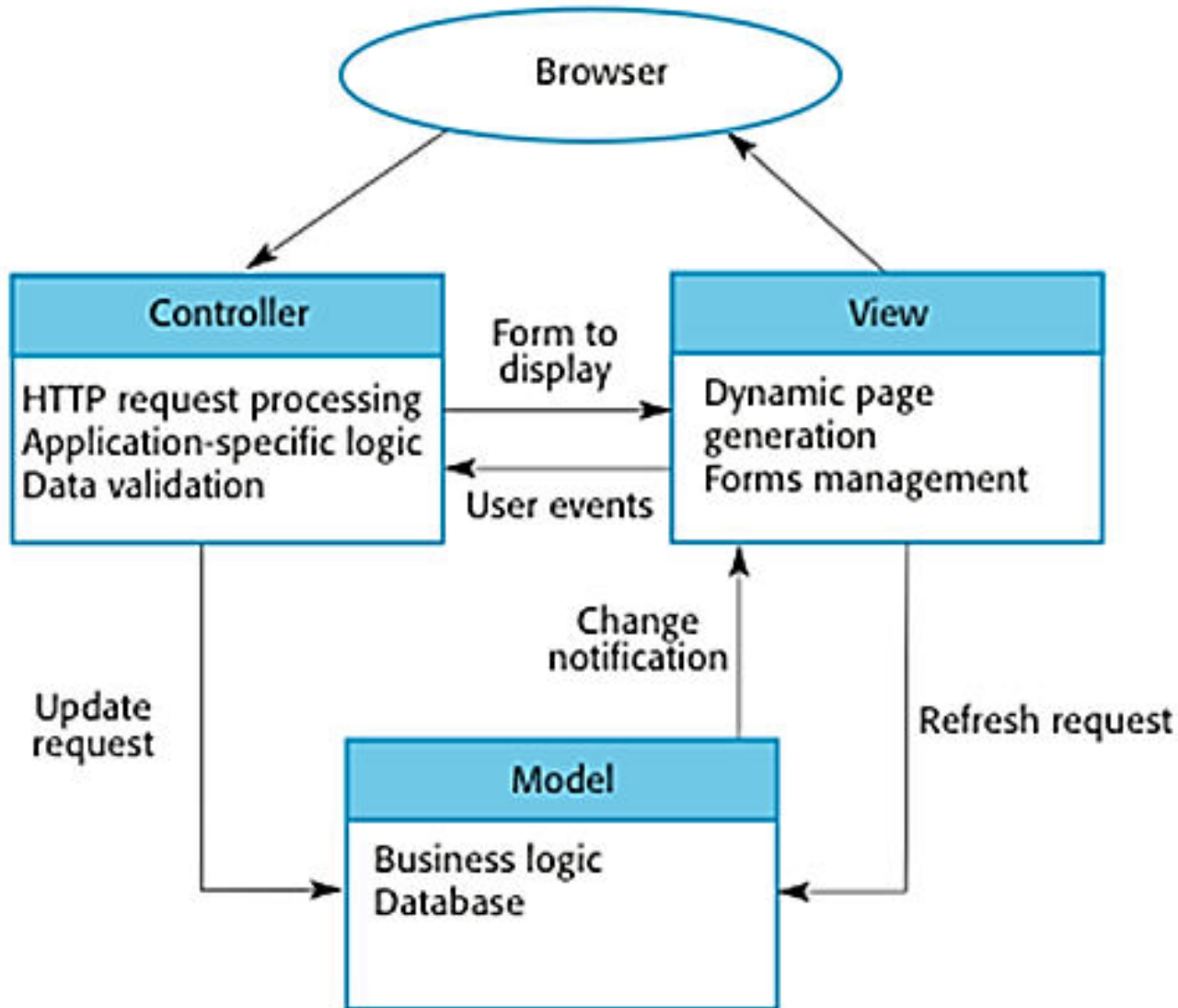
The Model-View-Controller (MVC) pattern

MVC architectural pattern



The Model-View-Controller (MVC) pattern

Web application architecture using the MVC pattern



The Model-View-Controller (MVC) pattern

Case- Study : **restaurant**

1. Let's assume you go to a restaurant. You will not go to the kitchen and prepare food which you can surely do at your home. Instead, you just go there and wait for the waiter to come on.
2. Now the waiter comes to you, and you just order the food. The waiter doesn't know who you are and what you want he just written down the detail of your food order.
3. Then, the waiter moves to the kitchen. In the kitchen waiter not prepare your food.
4. The cook prepares your food. The waiter is given your order to him along with your table number.
5. Cook then prepared food for you. He uses ingredients to cooks the food. Let's assume that your order a vegetable sandwich. Then he needs bread, tomato, potato, capsicum, onion, bit, cheese, etc. which he sources from the refrigerator
6. Cook final hand over the food to the waiter. Now it is the job of the waiter to moves this food outside the kitchen.
7. Now waiter knows which food you have ordered and how they are served.

The Model-View-Controller (MVC) pattern

Case- Study : restaurant

MVC Example



In this case,

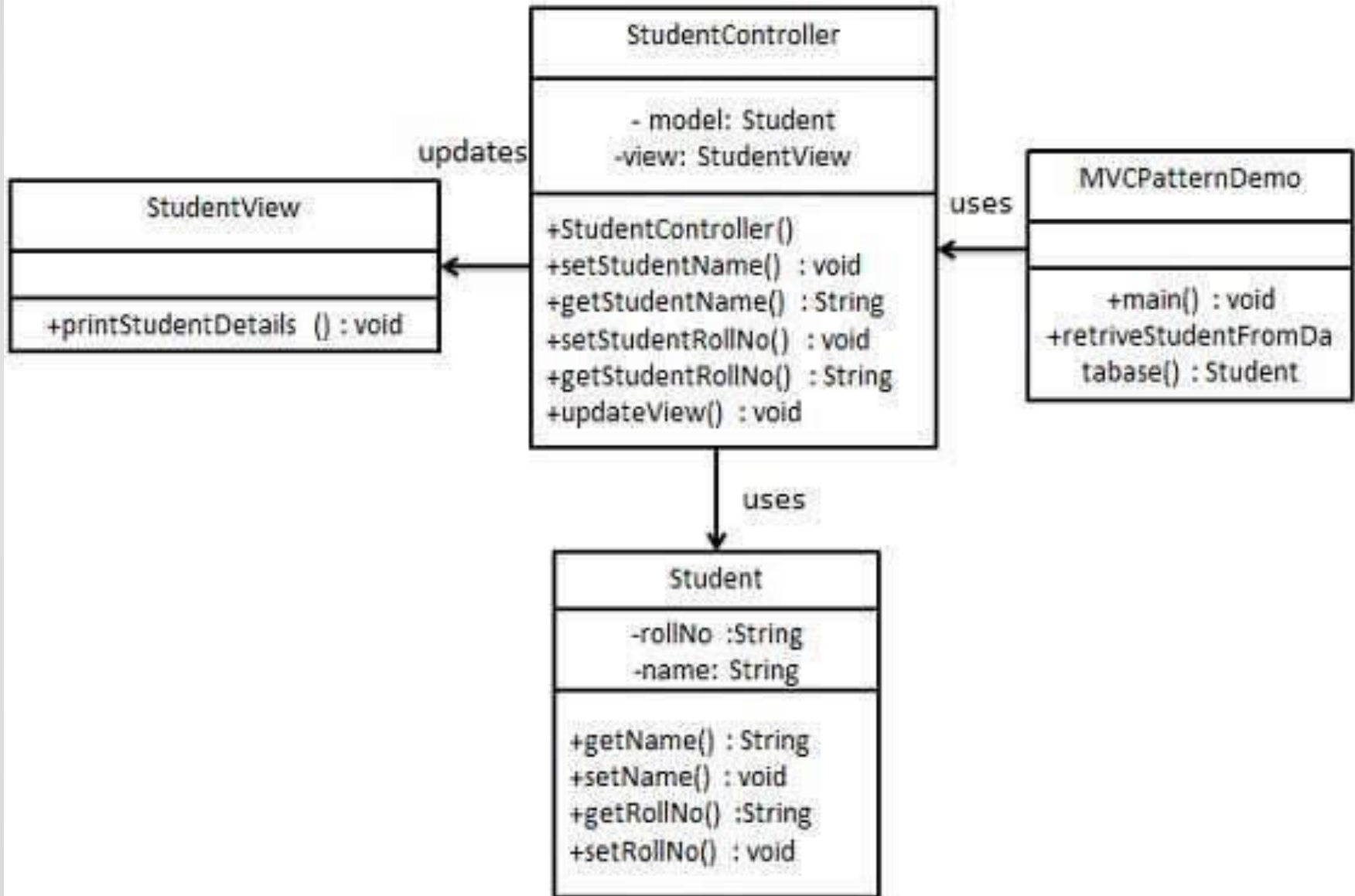
You= View

Waiter= Controller

Cook= Model

Refrigerator= Data

Coding Style : MVC



Design Patterns - MVC

Step 1 Create Model

```
public class Student {  
    private String rollNo;  
    private String name;  
  
    public String getRollNo() {  
        return rollNo;  
    }  
  
    public void setRollNo(String rollNo) {  
        this.rollNo = rollNo;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

Design Patterns - MVC

Step 2 Create view

```
public class StudentView {  
    public void printStudentDetails(String studentName, String studentRollNo){  
        System.out.println("Student: ");  
        System.out.println("Name: " + studentName);  
        System.out.println("Roll No: " + studentRollNo);  
    }  
}
```

Design Patterns - MVC

Step 3 Create Controller

```
public class StudentController {  
    private Student model;  
    private StudentView view;  
  
    public StudentController(Student model, StudentView view){  
        this.model = model;  
        this.view = view;  
    }  
  
    public void setStudentName(String name){  
        model.setName(name);  
    }  
    public String getStudentName(){  
        return model.getName();  
    }  
    public void setStudentRollNo(String rollNo){  
        model.setRollNo(rollNo);  
    }  
    public String getStudentRollNo(){  
        return model.getRollNo();  
    }  
  
    public void updateView(){  
        view.printStudentDetails(model.getName(), model.getRollNo());  
    }  
}
```

Design Patterns - MVC

Step 4 MVCPatternDemo

```
public class MVCPatternDemo {  
    public static void main(String[] args) {  
  
        //fetch student record based on his roll no from the database  
        Student model = retrieveStudentFromDatabase();  
  
        //Create a view : to write student details on console  
        StudentView view = new StudentView();  
  
        StudentController controller = new StudentController(model, view);  
  
        controller.updateView();  
  
        //update model data  
        controller.setStudentName("John");  
  
        controller.updateView();  
    }  
  
    private static Student retrieveStudentFromDatabase(){  
        Student student = new Student();  
        student.setName("Robert");  
        student.setRollNo("10");  
        return student;  
    }  
}
```

In Python

https://www.tutorialspoint.com/python_design_patterns/python_design_patterns_model_view_controller.htm

Step 5

Verify the output.

Student:

Name: Robert

Roll No: 10

What is MVVM?

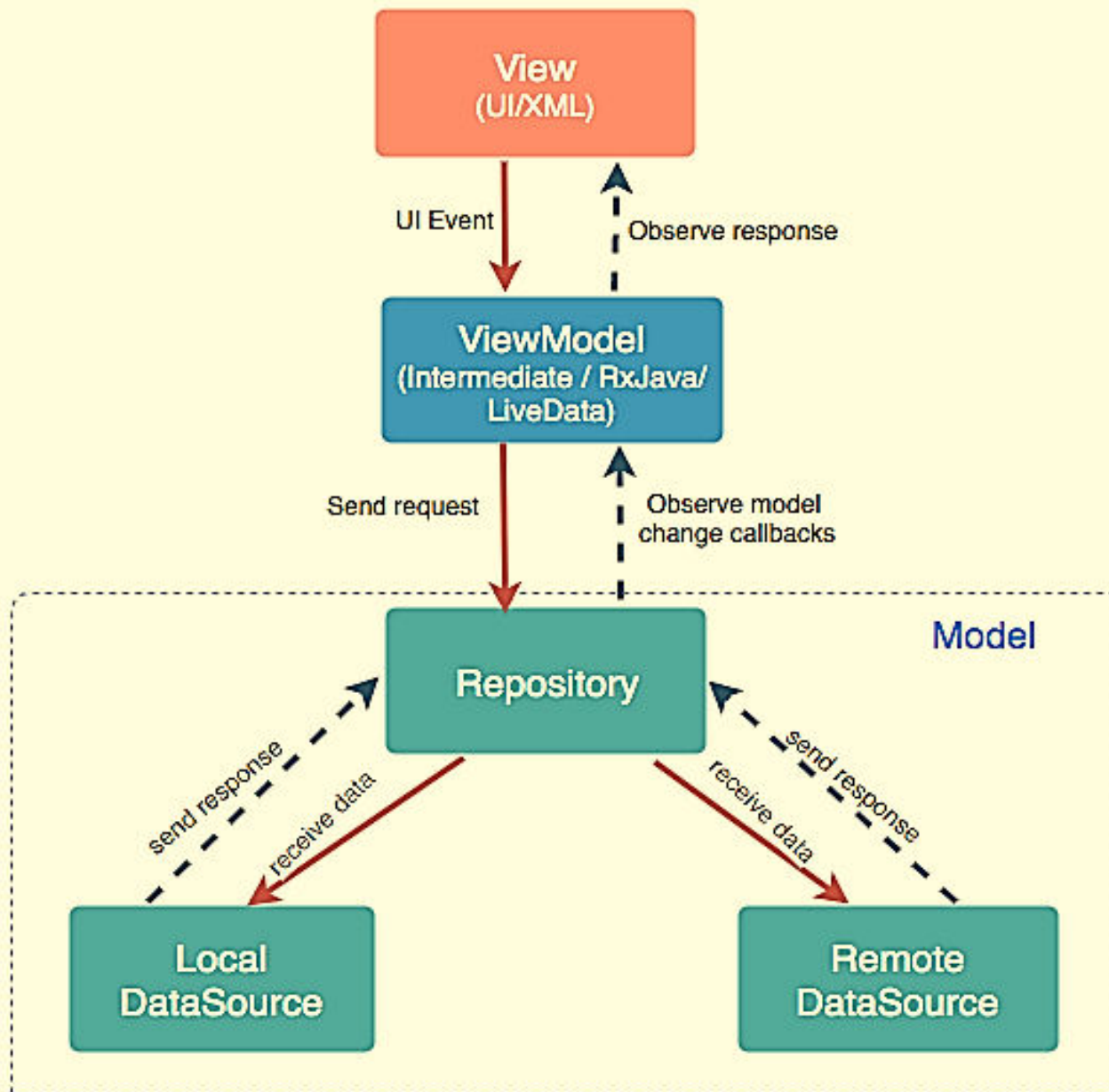
MVVM architecture facilitates a separation of development of the graphical user interface with the **help of mark-up language** or GUI code.

The full form of MVVM is **Model–View–ViewModel**.

MVVM removes the **tight coupling** between each component.

Most importantly, in this architecture, the children don't have the direct reference to the parent, they only have the reference by observables.

MVVM



Model: It represents the **data and the business logic** of the Application. It consists of the business logic - **local and remote data source, model classes, repository**.

View: It consists of the **UI Code** (Activity, Fragment), XML. It sends the user action to the ViewModel but does **not** get the response back directly. To get the response, it has to subscribe to the observables which ViewModel exposes to it.

ViewModel: It is a **bridge** between the View and Model(business logic). It does not have any clue which View has to use it as it does not have a direct reference to the View. So basically, the ViewModel should not be aware of the view who is interacting with. It interacts with the Model and exposes the observable that can be observed by the View.

Layered architecture

Have you ever wondered how Google makes **Gmail work** in different languages all over the world? Users can use Gmail every day in English, Spanish, French, Russian, and many more languages.

Did Google develop different Gmail applications for each country?
Of course not.

They developed **an internal version** that does all the message processing, and then developed different external user interfaces that work in many languages.

Layered architecture

Google developed the Gmail application in **different layers**:

1. There is an **internal layer** that does all the processing.
2. There is an **external layer** that communicates with the users in their language.
3. There is also **another layer that interacts with a database** where user email messages are stored (millions or maybe billions).

Gmail is divided into at least three layers, every one of them has a mission, and they exist separately to handle different processes at different levels. It is an excellent example of **a layered architecture**.

Layered architecture

- 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.

A generic layered architecture

User interface

User interface management
Authentication and authorization

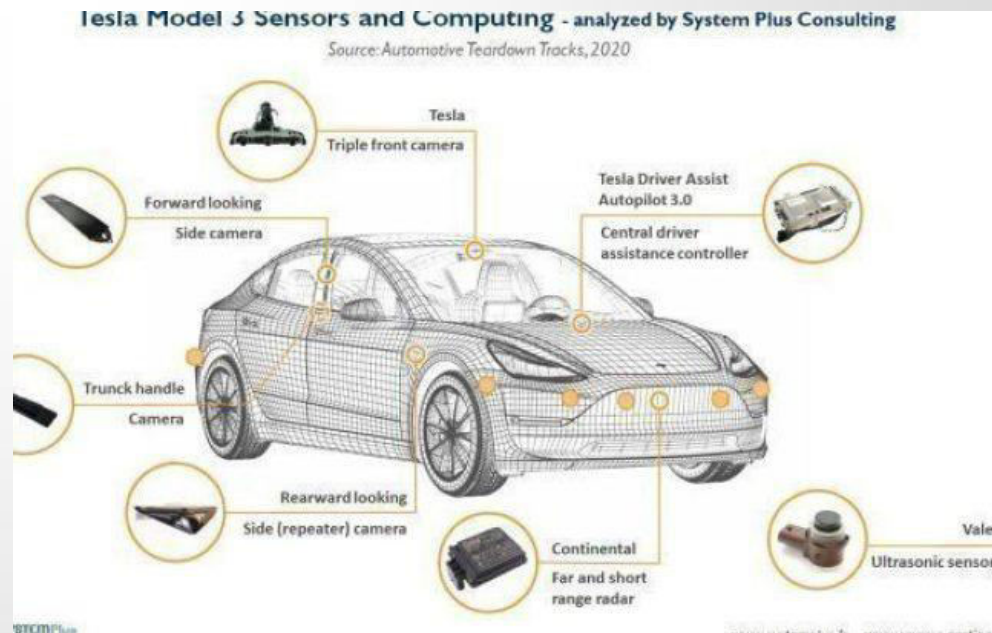
Core business logic/application functionality
System utilities

System support (OS, database etc.)

Layered architecture

Real-World Example

- Tesla Control Module (tesla.com): Control system for self-driving vehicles
- Waymo software (waymo.com)



Case-Study: Solving a Business Problem With Layered Architecture

Amaze is a project management software company.

Their product is sold globally with a monthly pay-per-user model and widely known among the project management community for being easy to use and able to operate on many different devices (PCs, Notebooks, laptops, tablets, iPhones, iPads, and Android phones).

Case-Study: Solving a Business Problem With Layered Architecture

What's the Business Problem?

The business problem is very straightforward: Amaze must work on any popular device on the market and be **able to support future devices**.

There must be **only one version** of the software for all devices. No special cases, no exceptions allowed.

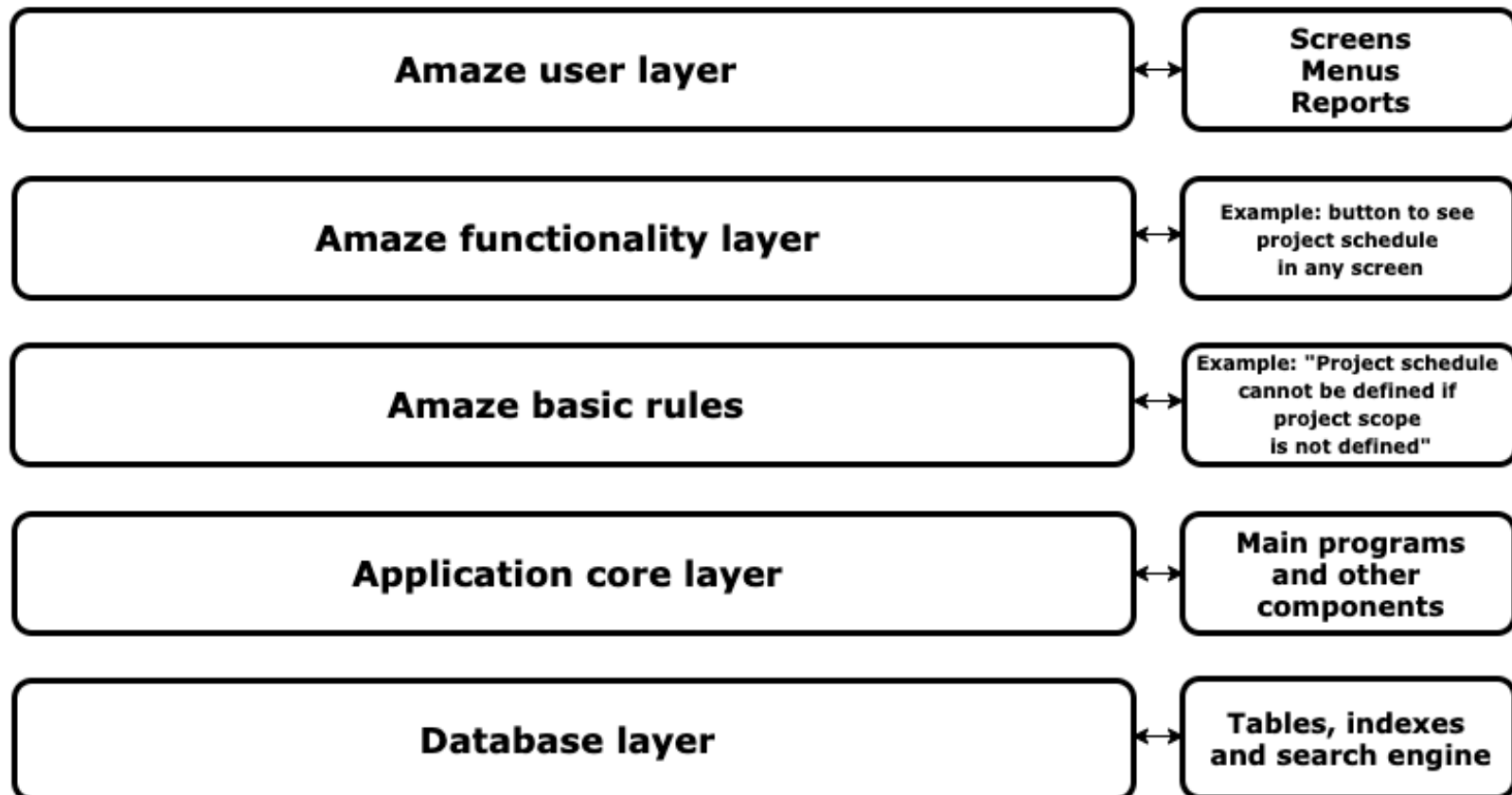
So to sum up:

- We know that **users have different devices**.
- There must be **only one software application** because the company wants to have low software maintenance costs.
- When **new device launches**, we do not want to change the whole software product.

Case-Study: Solving a Business Problem With Layered Architecture

What's the Solution?

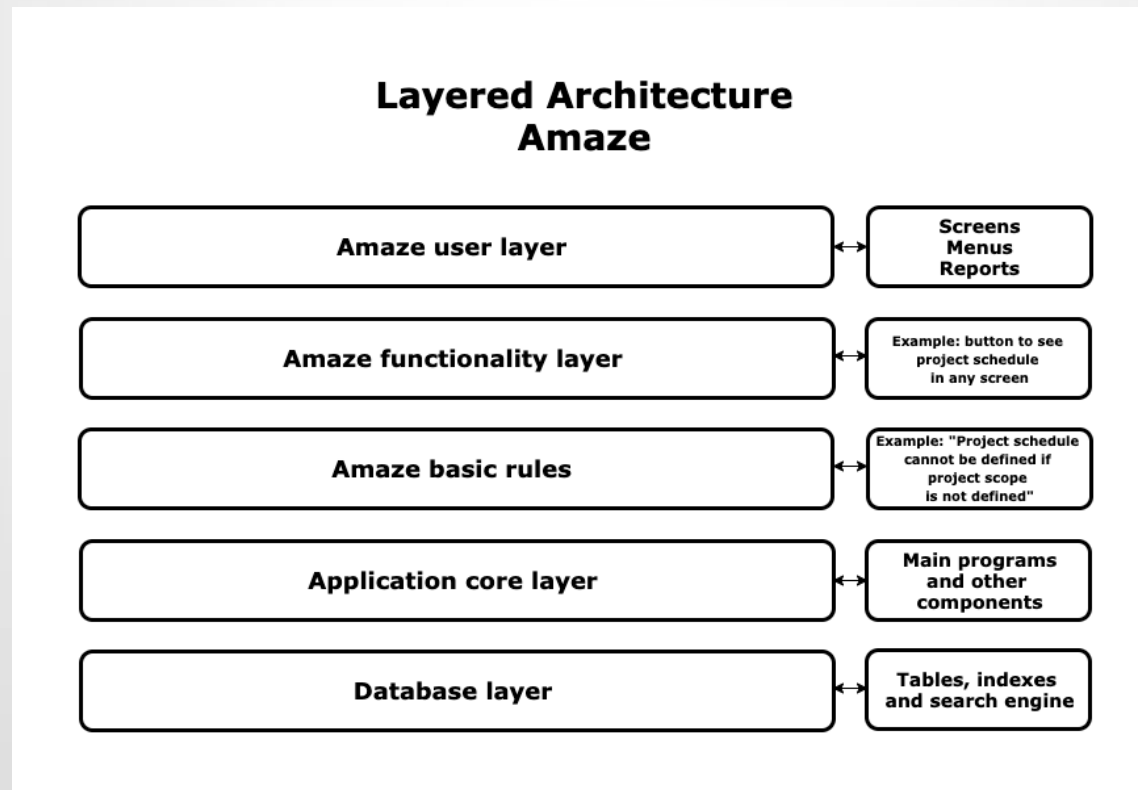
Layered Architecture Amaze



Case-Study: Solving a Business Problem With Layered Architecture

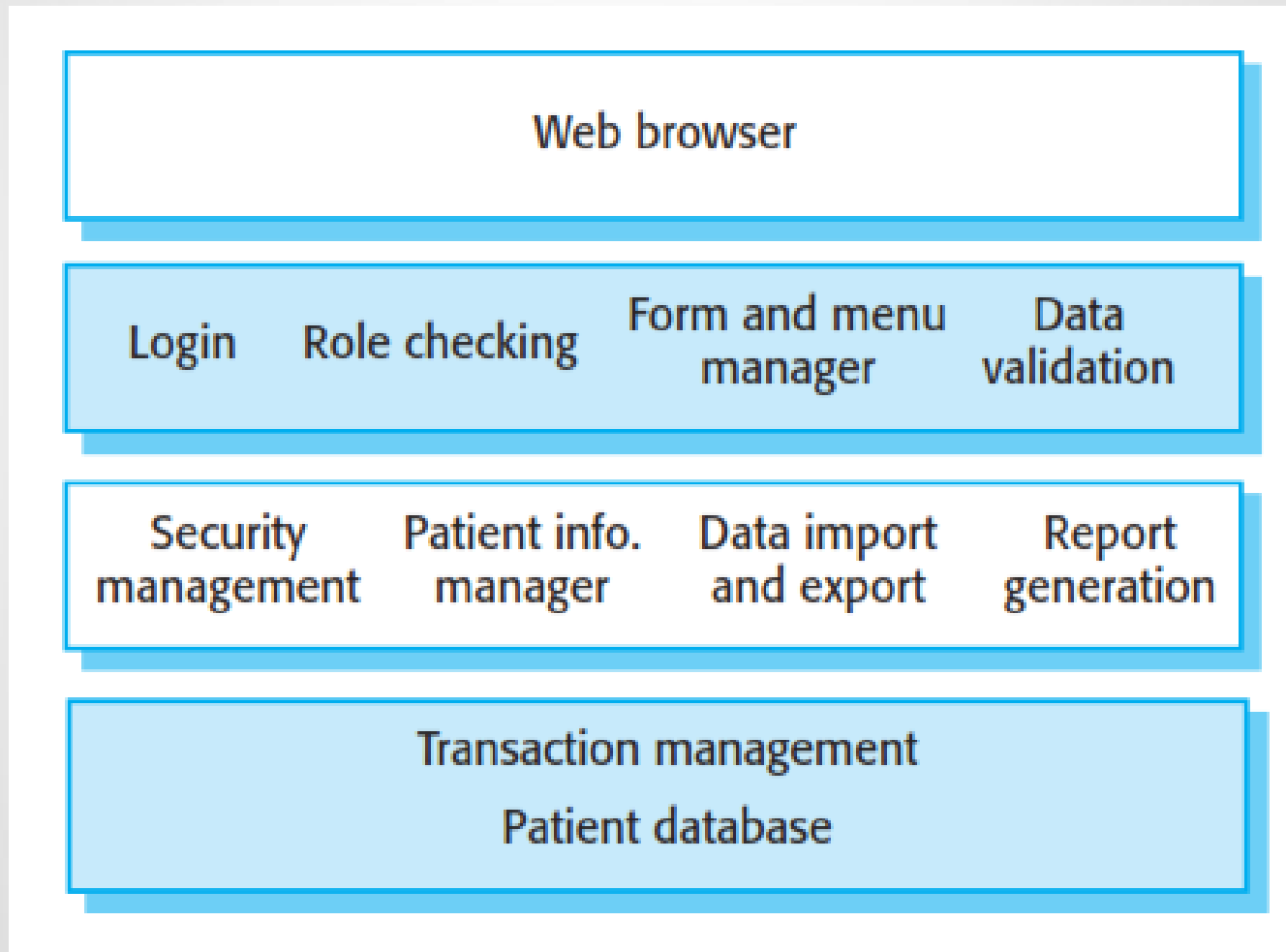
In our case, the **Amaze basic rules layer** is **critical**. This layer contains rules that determine the **behaviour of the whole application**, such as, "You can create a project schedule only if the project scope is defined."

The product's intelligence is in this layer: all special features that come from decades of experience in projects are developed there. The **application core layer** will be the most significant part of the application code.



The architecture of the Mentcare system

This system maintains and manages details of patients who are consulting specialist doctors about mental health problems.



The architecture of the Mentcare system

1. The top layer is a **browser-based user interface**.
2. The second layer provides the **user interface functionality** that is delivered through the web browser. It includes components to allow users to **log-in** to the system and checking components that ensure that the operations they use are allowed by their role. This layer includes form and menu management components that present information to users, and data validation components that check information consistency.
3. The third layer implements the functionality of the system and provides components that implement **system security**, patient information creation and updating, import and export of patient data from other databases, and report generators that create management reports.
4. Finally, the lowest layer, which is built using a **commercial database management system**, provides transaction management and persistent data storage.

The architecture of the iLearn system

Browser-based user interface

iLearn app

Configuration services

Group
management

Application
management

Identity
management

Application services

Email Messaging Video conferencing Newspaper archive
Word processing Simulation Video storage Resource finder
Spreadsheet Virtual learning environment History archive

Utility services

Authentication
User storage

Logging and monitoring
Application storage

Interfacing
Search

Layered architecture

There are several **advantages** to using layered architecture:

- **Layers are autonomous**: A group of changes in one layer does not affect the others. This is good because we can increase the functionality of a layer, for example, making an application that works only on PCs to work on phones and tablets, without having to rewrite the whole application.
- Layers allow **better system customization**.

There are also a few key **disadvantages**:

- Layers make an application more difficult to maintain. **Each change requires analysis**.
- Layers may affect application performance because they create overhead in execution: each layer in the upper levels must connect to those in the lower levels for each operation in the system.

Control styles

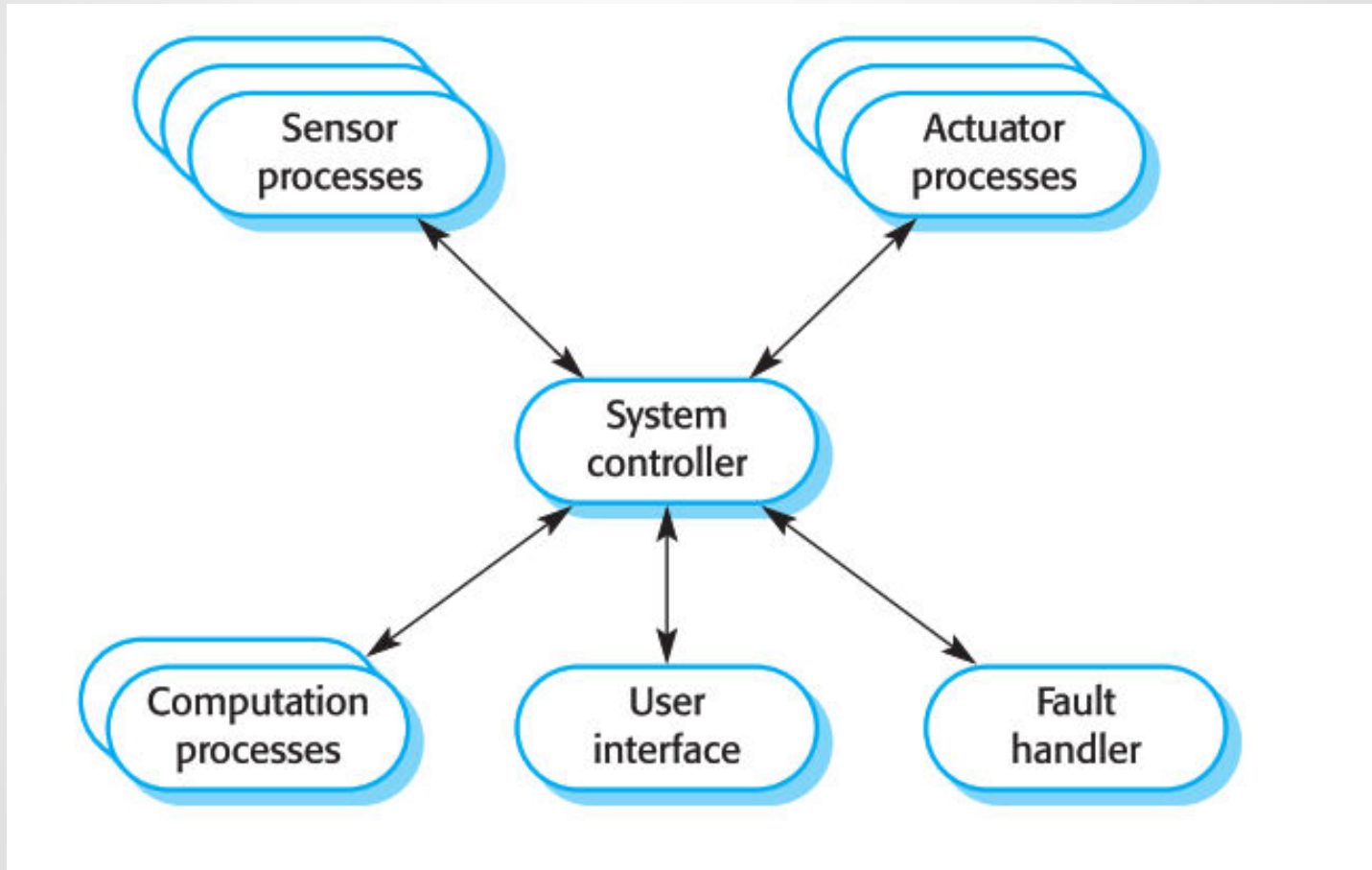
- Are concerned with the control flow between sub-systems. Distinct from the system decomposition model.
- Centralised control
 - One sub-system has overall responsibility for control and starts and stops other sub-systems.
- Event-based control
 - Each sub-system can respond to externally generated events from other sub-systems or the system's environment.

Centralized control

- A control sub-system takes responsibility for managing the execution of other sub-systems.
- **Call-return model**
 - **Top-down subroutine model** where control starts at the top of a subroutine hierarchy and moves downwards. Applicable to sequential systems.
- **Manager model**
 - Applicable to **concurrent systems**. One system component controls the stopping, starting and coordination of other system processes. Can be implemented in sequential systems as a case statement.

Centralised control

A centralized control model for a real-time system



The repository model

Sub-systems must **exchange data**. This may be done in two ways:

- Shared data is held in a **central database** or repository and may be accessed by all sub-systems;
- Each sub-system maintains its **own database** and passes data explicitly to other sub-systems.

When large amounts of data are to be shared, the repository model of sharing is most commonly used.

The repository model

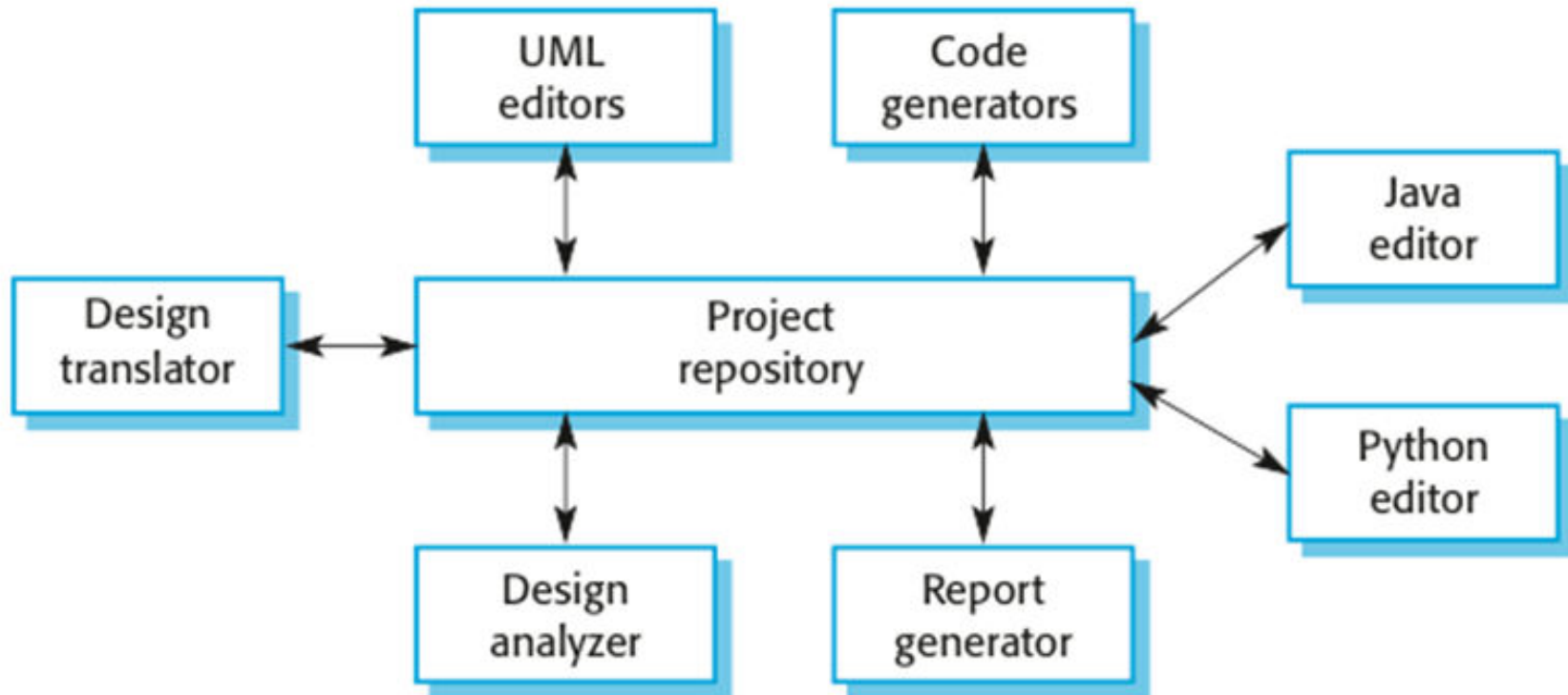
Advantages

- Efficient way to share **large amounts** of data;
- Sub-systems need not be concerned with how data is **produced**
- **Centralised** management e.g. backup, security, etc.
- **Sharing** model is published as the repository schema.

Disadvantages

- Sub-systems must agree on a repository data model. Inevitably a **compromise**;
- **Data evolution** is difficult and expensive;
- No scope for **specific** management policies;
- Difficult to **distribute** efficiently.

A repository architecture for an IDE



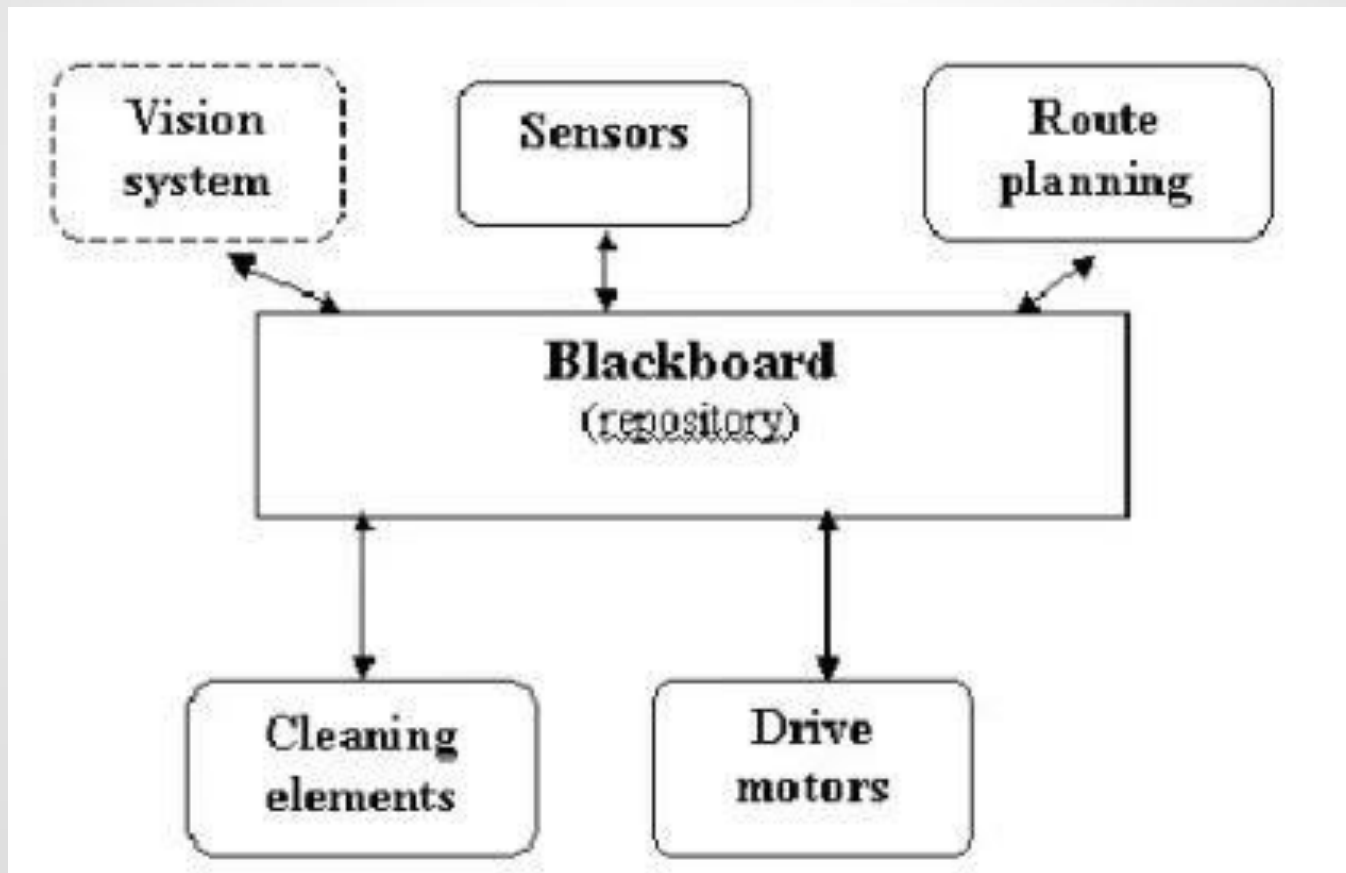
A repository architecture

Given reasons for your answer, suggest an appropriate structural model for the following systems:

- A **robot floor-cleaner** (standalone system) that is intended to clean relatively clear spaces such as corridors. The cleaner must be able to sense walls and other obstructions.

A repository architecture

- A robot floor-cleaner that is intended to clean relatively clear spaces such as corridors. The cleaner must be able to sense walls and other obstructions.



A repository architecture

- A robot floor-cleaner that is intended to clean relatively clear spaces such as corridors. The cleaner must be able to sense walls and other obstructions.
- The most appropriate model is a **repository model**, with each of the subsystems (wall and obstacle sensors, path planning, vision (perhaps), etc.) placing information in the repository for other subsystems to use.
- Robotic applications are in the realm of Artificial intelligence, and for the AI systems such as this, a special kind of repository called a **blackboard** (where the presence of data activates particular subsystems) is normally used

Plug-In Architecture

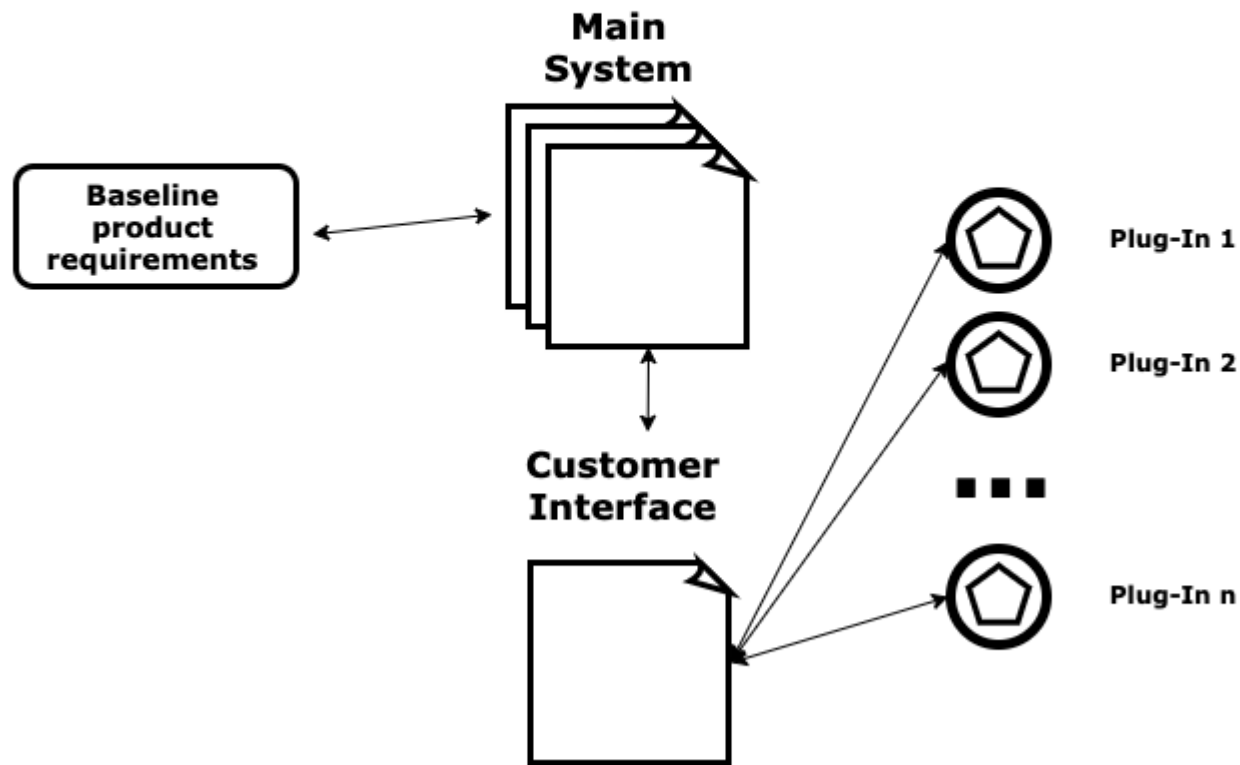
- ✧ Many people are worried about privacy, whether it's protecting their banking data, passwords, or making sure no one is reading their messages or emails. One way to ensure privacy protection is via encryption, and many software programs do so.
- ✧ Most encryption applications are installed in our browser (Chrome, Mozilla, Explorer, etc.) as an add-on: a module that works on top of the browser for any communication and encrypts it. This application is called a **plug-in**.

Plug-In Architecture

- ✧ A plugin architecture is an architecture that will **call external code at certain points without knowing all the details of that code in advance.**
- ✧ A plug-in is a **bundle** that **adds functionality** to an application, called the **host application**, through some well-defined architecture for extensibility. This **allows third-party developers** to add functionality to an application without having access to the source code.
- ✧ This also allows users to add new features to an application just by installing a new bundle in the appropriate folder.

Plug-In Architecture

Plug-In Architecture High Level Diagram



Plug-In Architecture

A standard plug-in architecture has four parts:

- **Baseline product requirements:** This is the **set of minimal requirements that define the application**, determined at the beginning of the development process when an initial set of features were included in the product.
- **Main system:** This is the application **we plug the plug-ins to**. The main system needs to provide a way to integrate plug-ins, and therefore will slightly vary the original baseline product to ensure compatibility.
- **Customer Interface:** This is the module that interacts with the customer, for example, a web browser (Chrome, Mozilla, etc.).
- **Plug-ins:** These are add-ons that enlarge the minimal requirements of the application and give it extra functionality.

Plug-In Architecture


There are several **advantages** to using plug-in architecture:

- The plug-in architecture is the **best way to add particular functionality** to a system that was not initially designed to support it.
- This architecture removes limits on the amount of functionality an application can have. We can add infinite plug-ins (The Chrome browser has hundreds of plug-ins, called extensions).
- No rewriting the system.

There are also a few key **disadvantages**:

- Plug-ins **can be a source of viruses and attacks from external players**.
- Having many plug-ins in an application may affect its performance.
- Plug-ins frequently **crash with each other** and produce malfunctions in the main system.

Real-World Example

- SendSafely (sendsafely.com)
 - Mailvelope (mailvelope.com)
- 
- Encryption software
- Rapportive (rapportive.com) connects LinkedIn to your browser.
 - Trello (trello.com) connects Trello to your browser.

Case-Study: Solving a Business Problem With Plug-In Architecture

Pacific is a retail website that is very successful in Southeast Asia. Here are some **quick facts**:

- The company sells more than 64,000 items on its website, mainly to Southeast Asian customers.
- About 12 items are sold every **second** on the website, 24 hours a day, 365 days a year.
- There are heavy users of the site: users that buy many items at once, and they need to do it quickly.

What's the Business Problem?

Each time a heavy user wants to buy many items at once, he or she must **open separate browser windows** for each item, causing confusion and frustration, which can stop the user from making a purchase.

For example, some users select ten items, but in the end they buy only six because they go back and forth from the shopping cart to the item pages.

How can this problem be solved?

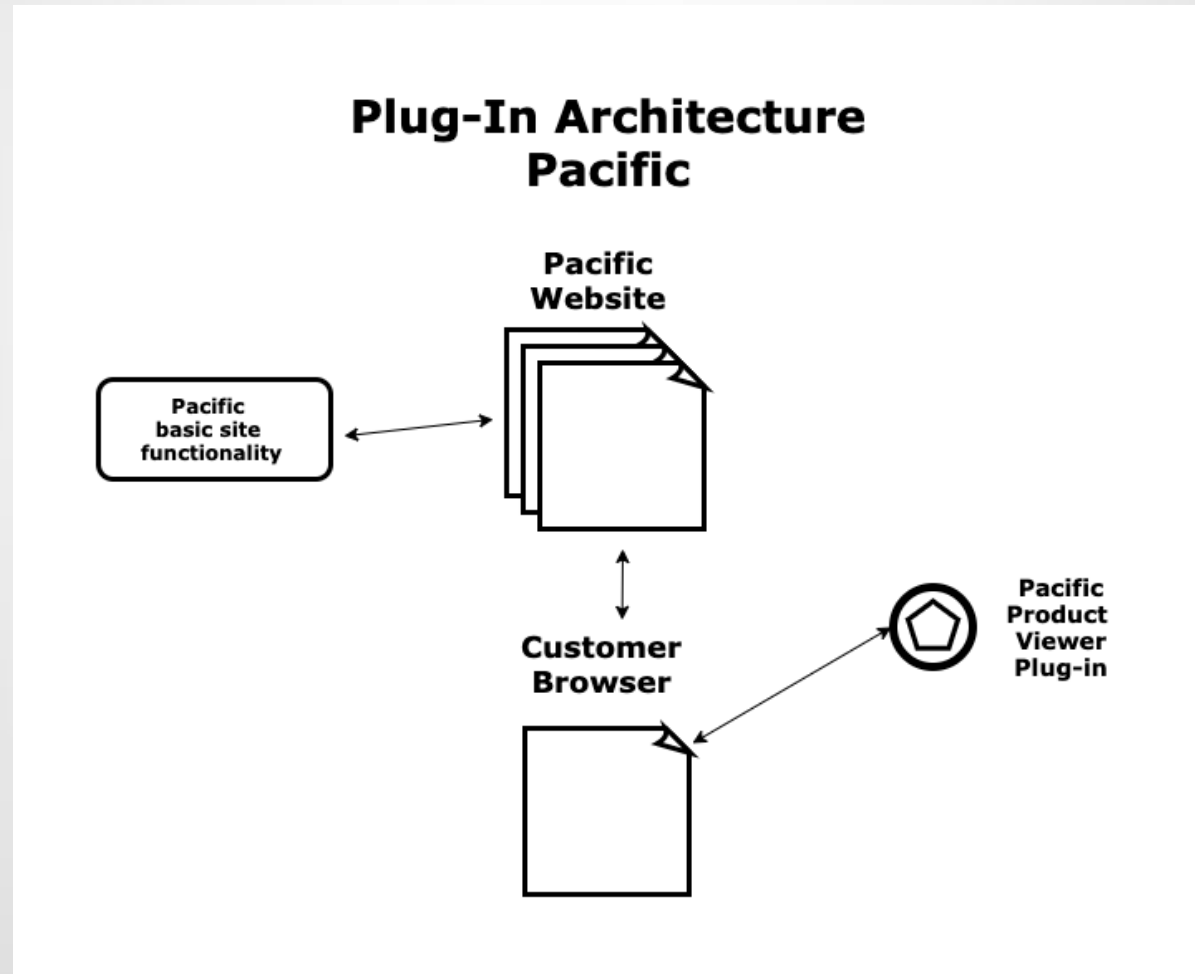
Pacific Product Viewer, a plug-in for any browser

Pacific has developed a plug-in for Chrome called Pacific Product Viewer: when the user installs the plug-in, a list of interesting items is shown in a pop-up window, and there is no need to open multiple tabs.

This window allows the user to see many items without having to switch tabs. It is usually located in the upper-right corner of the screen.

Case-Study: Solving a Business Problem With Plug-In Architecture

- **Pacific website:** It is the main Pacific website, developed according to a minimal set of requirements.
- **Customer browser:** Chrome, Mozilla, Explorer, etc.
- **Pacific Product Viewer Plug-in:** The piece of software that was developed as an add-on for extra functionality.



Distributed Systems Architectures

✧ **Distributed Systems Architectures**

- ✧ **Client-Server Architecture**

- ✧ **Broker Architectural Style : CORBA**

- ✧ **Service-Oriented Architecture (SOA)**

Distributed Architecture

- A distributed system is "**a collection of independent computers that appears to the user as a single coherent system.**" Information processing is distributed over several computers rather than confined to a single machine.
- Virtually all large computer-based systems are now distributed systems.
- **Distributed software engineering** is therefore very important for enterprise computing systems.

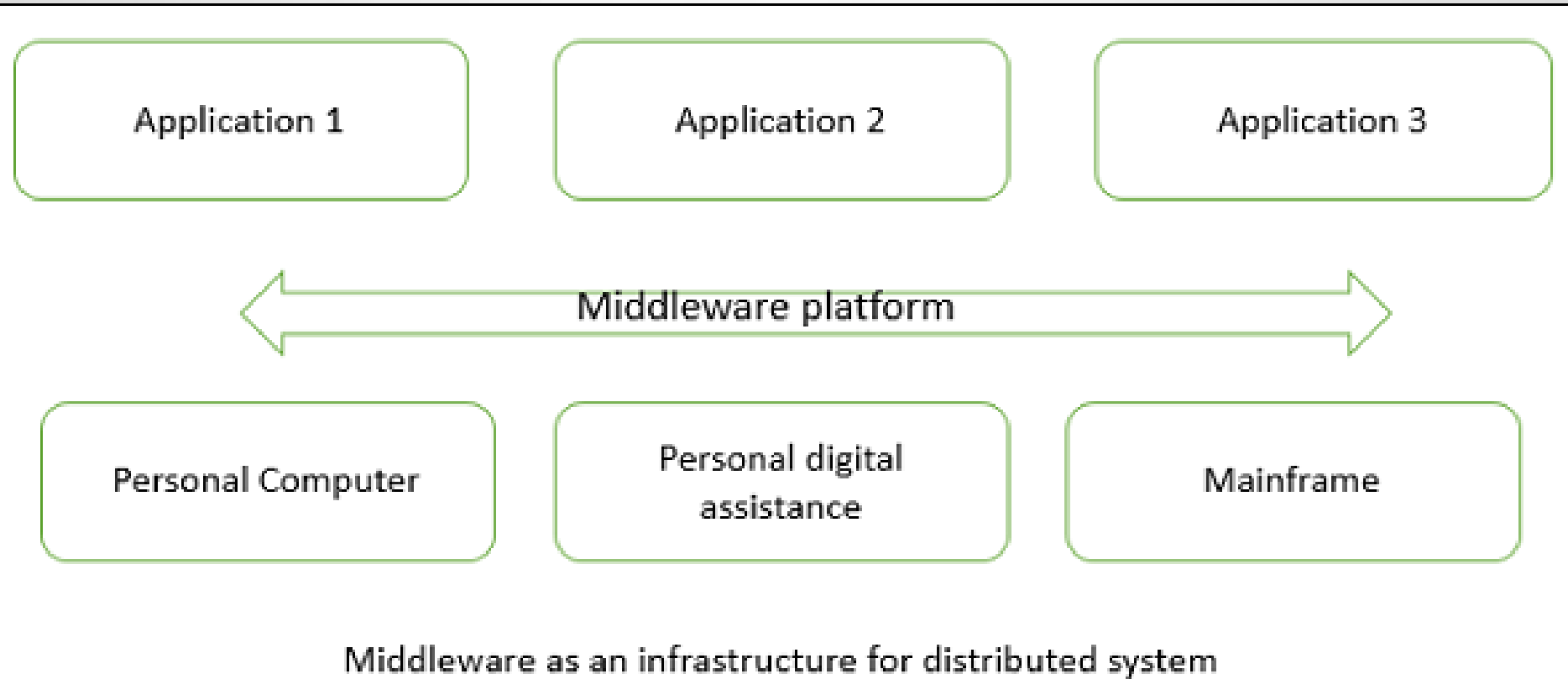
Distributed Architecture

- A distributed system can be demonstrated by the client-server architecture which forms the base for multi-tier architectures; alternatives are the broker architecture such as CORBA, and the Service-Oriented Architecture (SOA).
- There are several technology frameworks to support distributed architectures, including .NET, J2EE, CORBA, .NET Web services, AXIS Java Web services, and Globus Grid services.

Distributed Architecture

- **Middleware** is an infrastructure that appropriately supports the development and execution of distributed applications.
- It provides a **buffer** between the **applications and the network**.
- It sits in the middle of system and **manages or supports the different components** of a distributed system.
- Examples are transaction processing monitors, data convertors and communication controllers etc.

Distributed System



Distributed System

Advantages

- **Resource sharing** – Sharing of hardware and software resources.
- **Openness** – Flexibility of using hardware and software of different vendors.
- **Concurrency** – Concurrent processing to enhance performance.
- **Scalability** – Increased throughput by adding new resources.
- **Fault tolerance** – The ability to continue in operation after a fault has occurred.

Disadvantages

- **Complexity** – They are more complex than centralized systems.
- **Security** – More susceptible to external attack.
- **Manageability** – More effort required for system management.
- **Unpredictability** – Unpredictable responses depending on the system organization and network load.

Centralized System vs. Distributed System

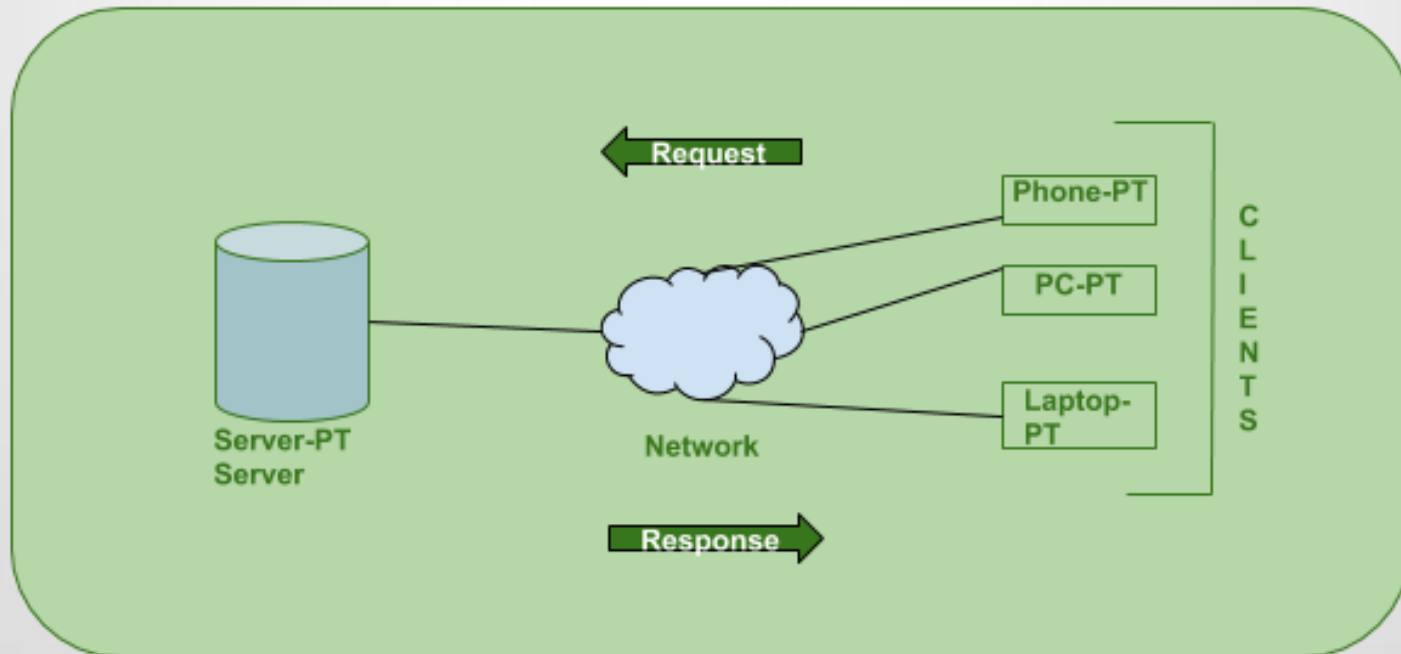
Criteria	Centralized system	Distributed System
Economics	Low	High
Availability	Low	High
Complexity	Low	High
Consistency	Simple	High
Scalability	Poor	Good
Technology	Homogeneous	Heterogeneous
Security	High	Low

Client-Server Architecture

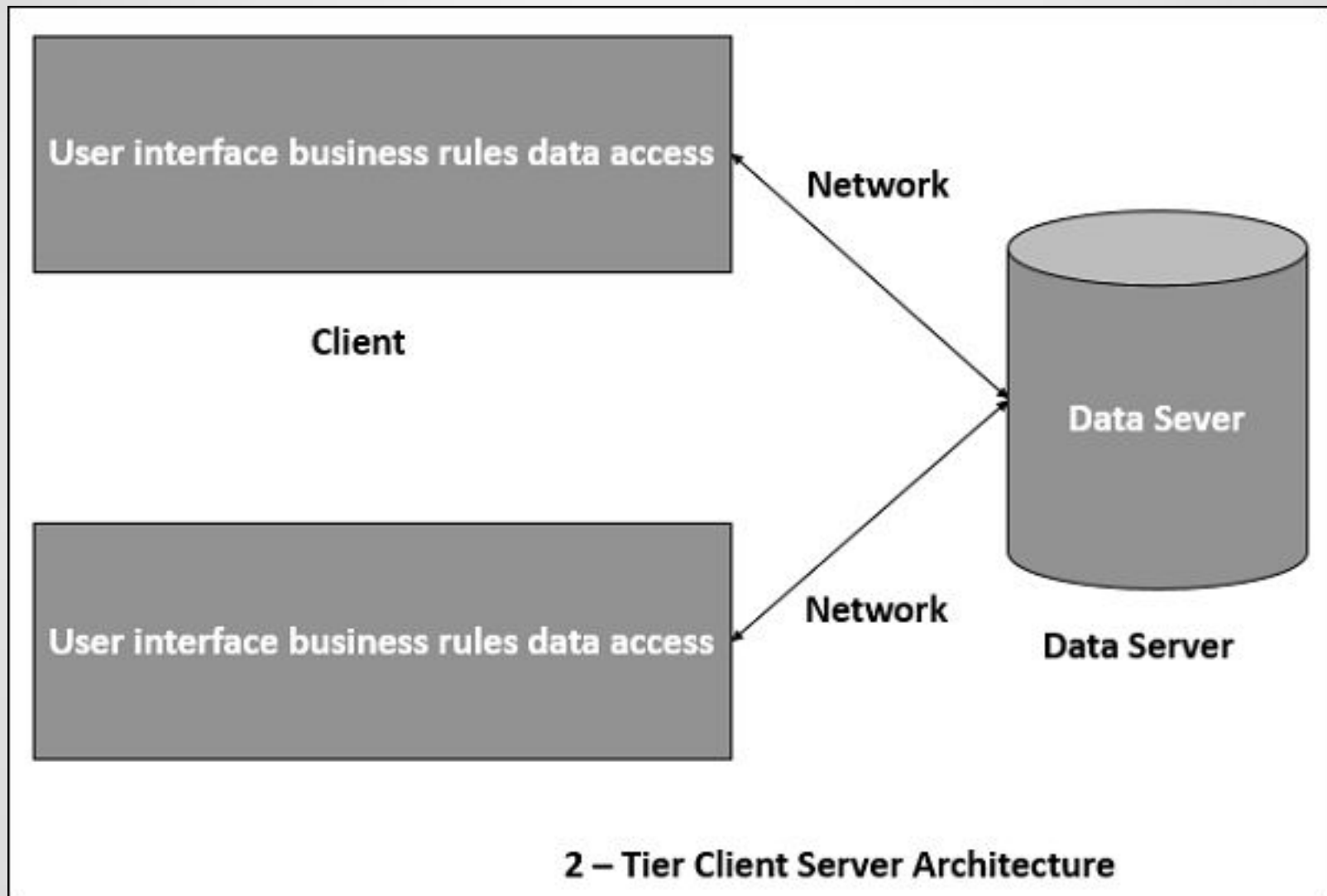
The client-server architecture is the most common distributed system architecture which decomposes the system into two major subsystems or logical processes –

Client – This is the first process that issues a **request to** the second process i.e. the server.

Server – This is the second process that **receives the request**, carries it out, and sends a reply to the client.



Client-Server Architecture

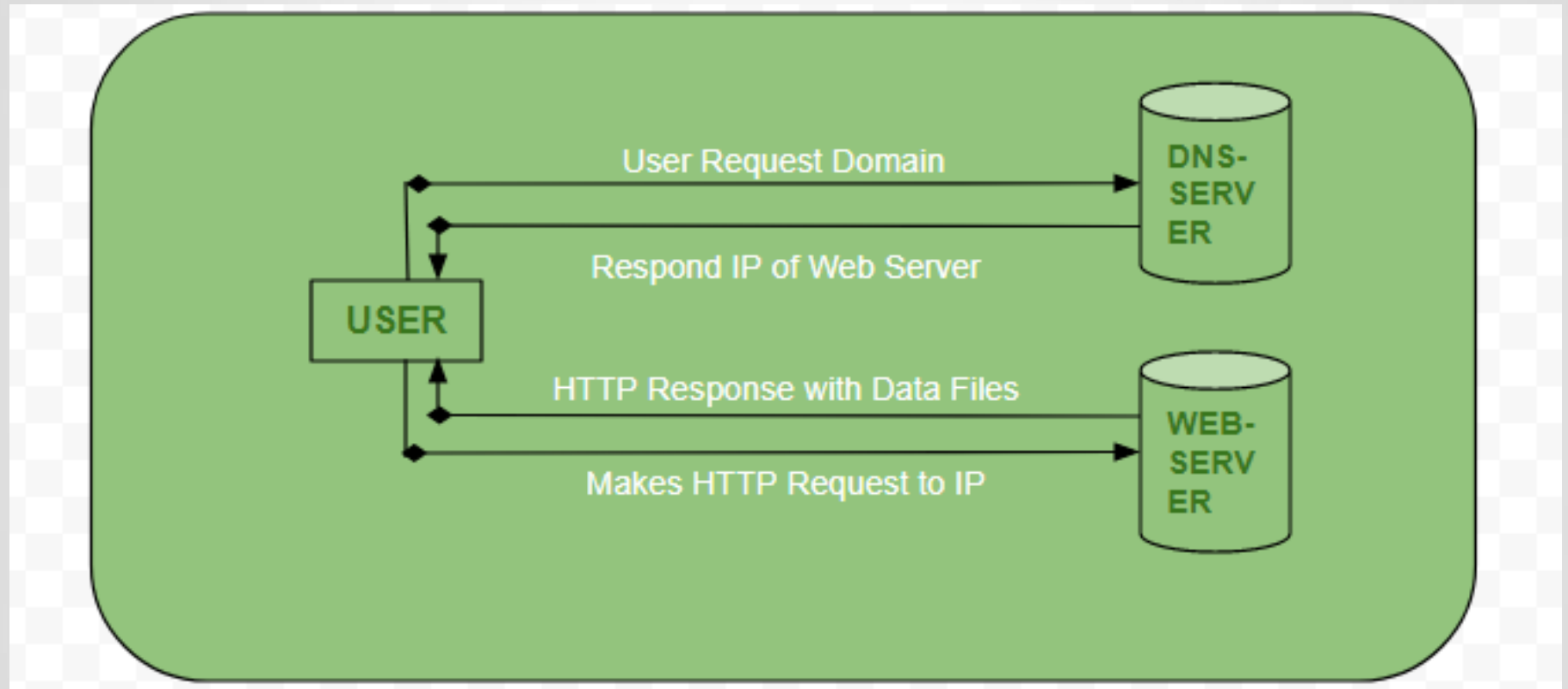


How the Client-Server Model (web browsers) works ?

- User enters the **URL**(Uniform Resource Locator) of the website or file. The Browser then requests the **DNS**(DOMAIN NAME SYSTEM) Server.
- **DNS Server** lookup for the address of the **WEB Server**.
- **DNS Server** responds with the **IP address** of the **WEB Server**.
- Browser sends over an **HTTP/HTTPS** request to **WEB Server's IP** (provided by **DNS server**).
- Server sends over the necessary files of the website.
- Browser then renders the files and the website is displayed. This rendering is done with the help of **DOM** (Document Object Model) interpreter, **CSS** interpreter and **JS Engine** collectively known as the **JIT** or (Just in Time) Compilers.
-

Client-Server Architecture

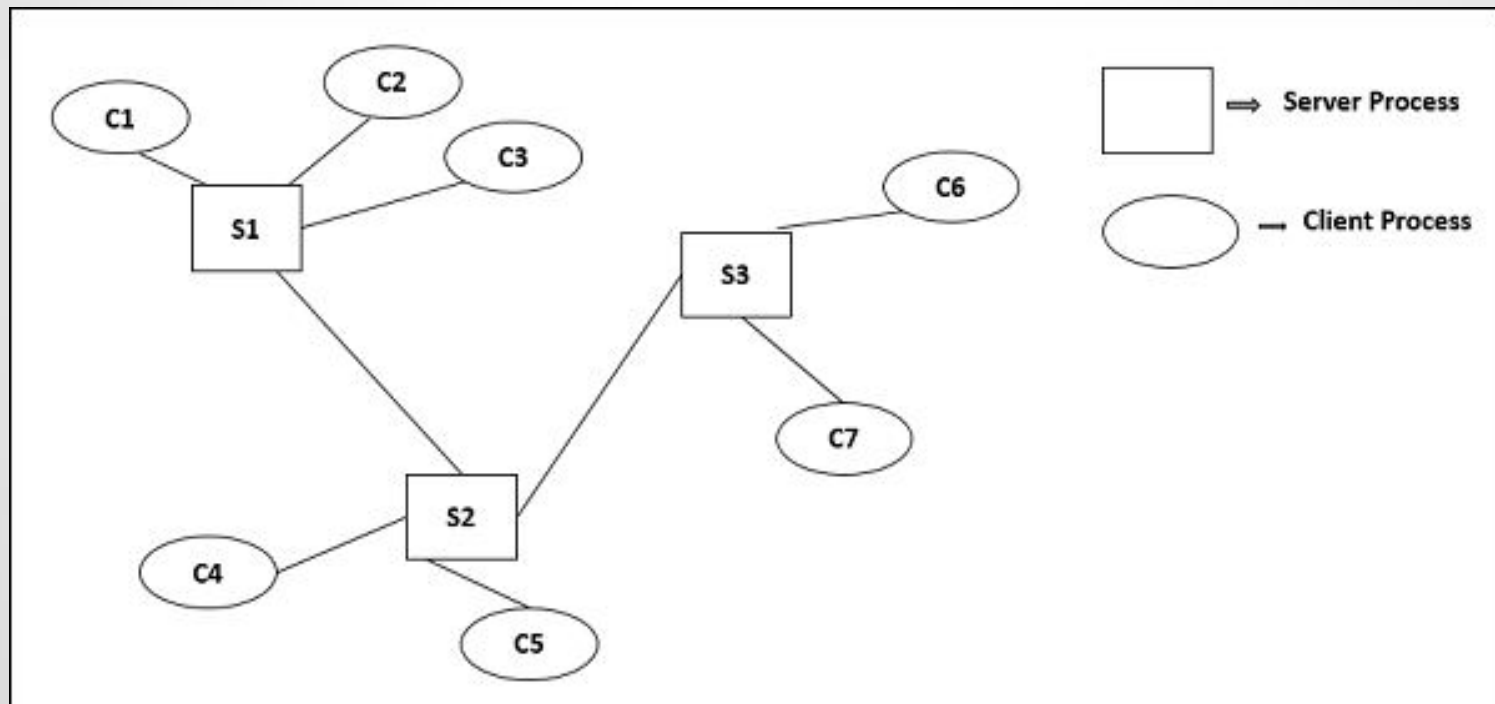
How the Client-Server Model (web browsers) works ?



Multi-Tier Architecture (n-tier Architecture)

Multi-tier architecture is a client–server architecture in which the functions such as **presentation**, **application** processing (Business Logic, Logic Tier, or Middle Tier), and **data** management are physically separated.

** 3-Tier Architecture, 2-Tier Architecture



Client-Server Architecture

Advantages

- Distribution** of data is straightforward;
- Makes effective use of **networked** systems. May require cheaper hardware;
- Easy to add **new servers or upgrade existing servers**.

Disadvantages

- No shared data model so sub-systems use **different data organisation**. **Data interchange** may be **inefficient**;
- Redundant management** in each server;
- No central register of names and services** - it may be hard to find out what servers and services are available.

Here are some real-life situations that this would be useful for:

- Enterprise resource planning (ERP) system
- SAP (sap.com).
- Oracle Business Suite (oracle.com).
- Microsoft Dynamics (microsoft.com).
- Infor (infor.com).
- Epicor (epicor.com).

Case-Study: Solving a Business Problem With Client-Server Architecture

IrisGold is a gold mining company. Here are some **quick facts**:

- IrisGold operates on **three continents**, with more than **21,000 employees**.
- The company's mines are mostly **located in remote places** like the Amazonas in Brazil, the Andes mountain range, the Ural mountains in Russia, and eastern South Africa.
- The company is selecting an Enterprise Resource Planning (ERP) system package. How does this inform your decision?

What's the Business Problem?

IrisGold wants to deploy and operate its new ERP package securely. But they have **two main constraints**:

- Its users are in **remote places** in the world with different kinds of devices (laptops, notebooks, phones, tablets).
- **Client devices must be light**: they must be a simple notebook computer, tablet, or phone with few processing and storage capabilities, **able to communicate to a remote server**.

Case-Study: Solving a Business Problem With Client-Server Architecture

Let's start with the facts!

1. We know that users are scattered over the world and use different devices that need to be lightweight.
2. We know that users have deficient infrastructure capabilities and work in remote places.
3. Clients need only to be able to connect with a central server.
4. The ERP system installed in the central server must cover all the business rules. In essence, it must manage all modules for all countries internally, and it must answer client requests by communicating with a central database.

Case-Study: Solving a Business Problem With Client-Server Architecture

So to sum up:

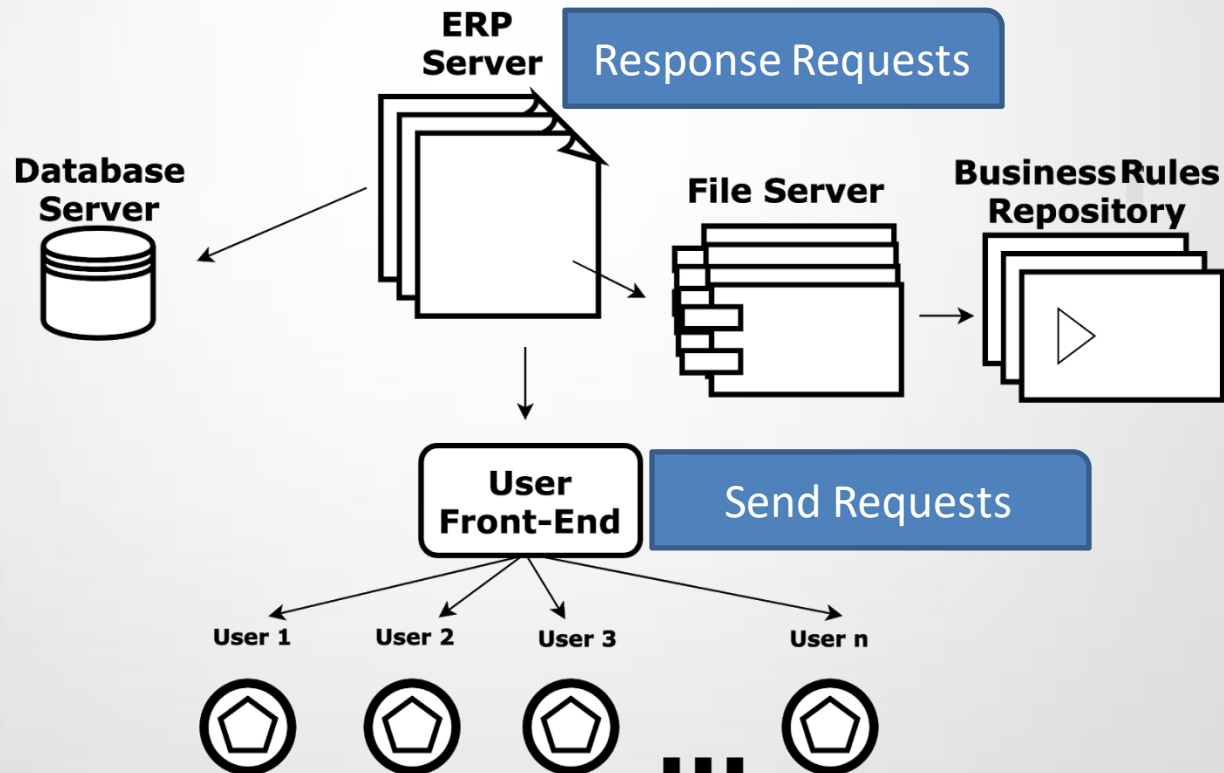
1. We know that users are scattered over the world and use different devices that need to be lightweight.
2. We know that users have deficient infrastructure capabilities and work in remote places.
3. Clients need only to be able to connect with a central server.
4. The ERP system installed in the central server must cover all the business rules. In essence, it must manage all modules for all countries internally, and it must answer client requests by communicating with a central database.

Case-Study: Solving a Business Problem With Client-Server Architecture

What's the Solution?

All heavy processing is done at **IrisGold's headquarters**. Clients are thin; that's why they are called "**thin clients**." They do not process or store large amounts of data. They just speak with the server.

Client-Server Architecture IrisGold



Case-Study: Solving a Business Problem With Client-Server Architecture

Front-End: This is the piece of software that interacts with ERP users, even if they are in different countries.

ERP server: This is the server where the ERP software is installed.

File server: The ERP server requests files from the file server to fulfill user requests. Examples: an invoice printed in PDF format, a report, a data file that the user needs. It is good practice to install a file server when the application has many users that read, update, or write files frequently.

Business rules repository: A repository of business procedures, methods, and regulations for every country (all different), to make the ERP work according to each country's needs. This repository is often separated from the software to make customizations more agile and secure. It is a good practice introduced by ERPs in the '90s that has spread over many kinds of non-ERP applications.

Database server: This server contains the tables, indexes, and data managed by the ERP system. Examples: customer table, provider table, invoice list, stock tables, product IDs, etc.

Broker Architectural Style

Broker Architectural Style

Broker Architectural Style is a middleware architecture used in distributed computing to **coordinate and enable** the communication between **registered servers and clients**. Here, object communication takes place through a middleware system called an **object request broker(ORB)**.

- Client and the server **do not interact** with each other **directly**. Client and server have a direct connection to **its proxy** which communicates with the mediator-broker.
- A server provides services by registering and publishing their interfaces with the broker and clients can request the services from the broker statically or dynamically by look-up.
- **CORBA** (Common Object Request Broker Architecture) is a good implementation example of the broker architecture.

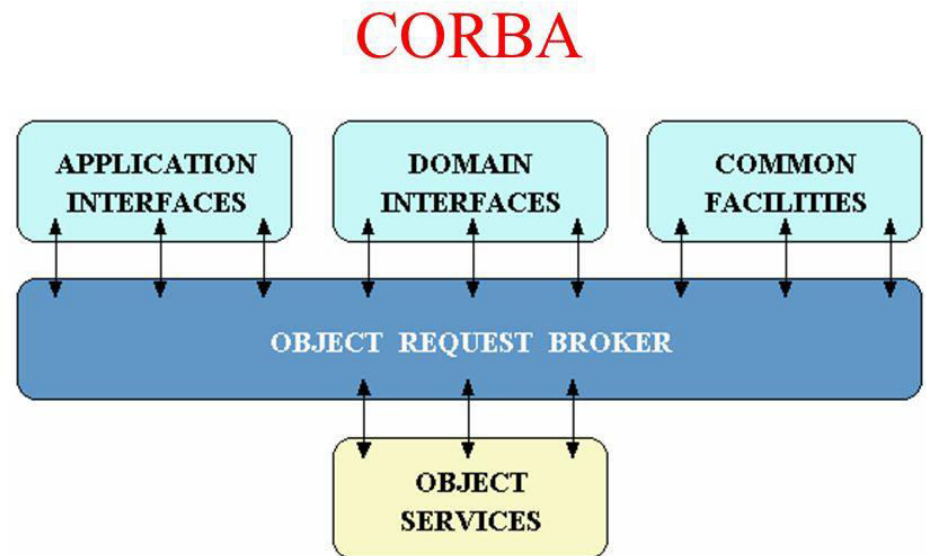
Broker implementation in CORBA

CORBA, or **Common Object Request Broker Architecture**, is a standard architecture for **distributed object systems**. It allows a distributed, heterogeneous **collection of objects to interoperate**.

CORBA is a standard **defined by the OMG** (Object Management Group). It describes an architecture, interfaces, and protocols that distributed objects can use to interact with each other.

Part of the CORBA standard is the **Interface Definition Language (IDL)**, which is an implementation-independent language for **describing the interfaces of remote objects**.

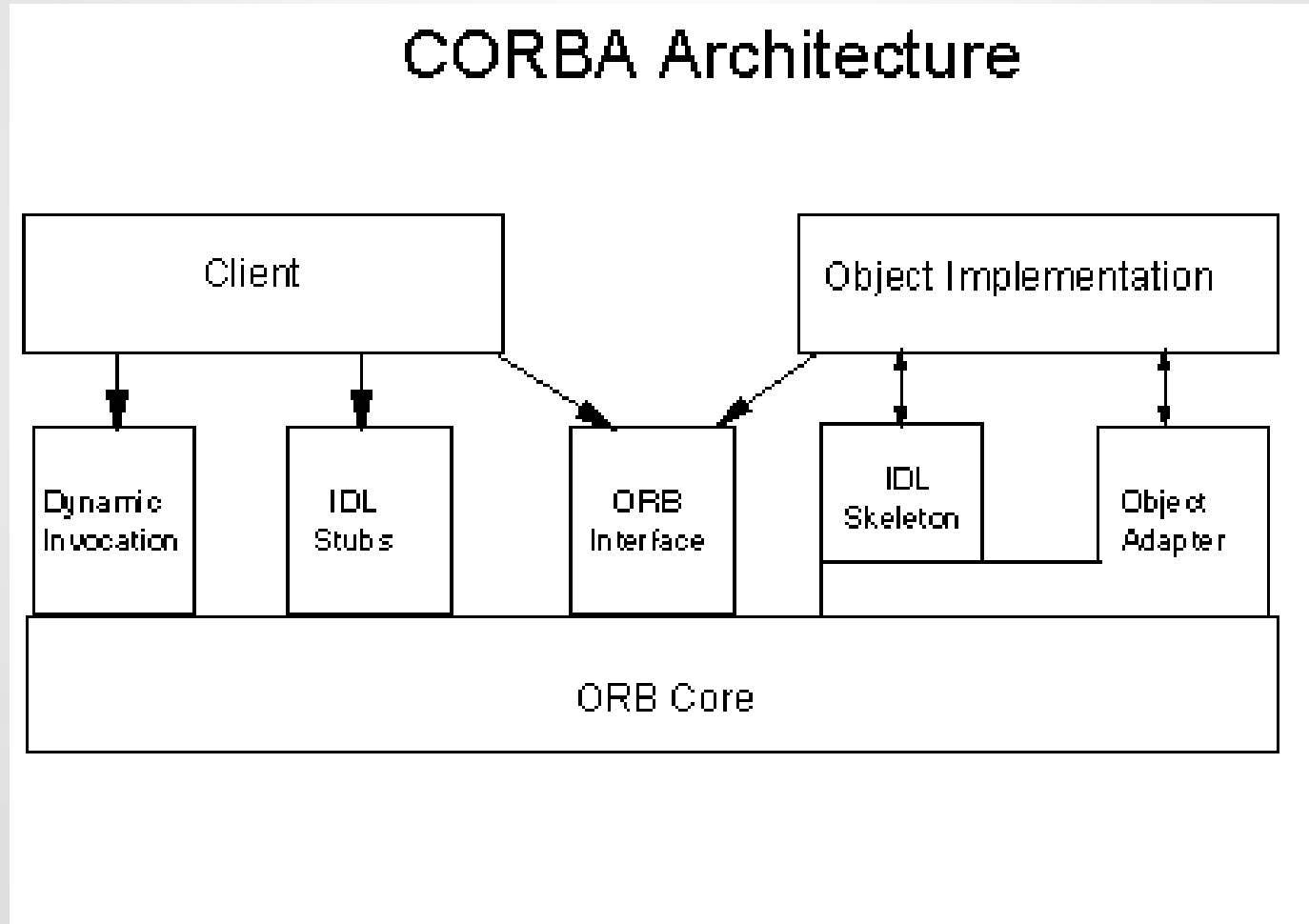
CORBA **describes a messaging mechanism** by which objects distributed over a network can communicate with each other irrespective of the platform and language used to develop those objects.



OMG Reference Model architecture

Broker implementation in CORBA

The ORB manages the interactions between clients and object implementations (Server).



Broker implementation in CORBA

Client Side Architecture

The client side architecture provides clients with interfaces to the ORB and object implementations. It consists of the following interfaces:

Dynamic Invocation - This interface allows for the **specification of requests at runtime**. This is necessary when **object interface is not known at run-time**. Dynamic Invocation **works in conjunction with the interface repository**.

IDL Stub - This component consists of **functions generated by the IDL interface definitions** and linked into the program. The functions are a **mapping between the client and the ORB implementation**. Therefore ORB capabilities can be made available for any client implementation for which there is a language mapping. Functions are called just as if it was a **local object**.

ORB Interface - The ORB interface may be called by either the client or the object implementation. The interface **provides functions of the ORB** which may be directly accessed by the client (such as retrieving a reference to an object.) or by the object implementations. This interface is **mapped to the host programming language**. The ORB interface must be supported by any ORB.

ORB core - Underlying mechanism used as **the transport level**. It provides **basic communication of requests to other sub-components**.

Broker implementation in CORBA

Implementation Side Architecture

The implementation side interface consists of the ORB Interface, ORB core and IDL Skeleton Interface defined below:

IDL Skeleton Interface - The ORB calls method **skeletons** to **invoke the methods** that were requested from clients. Object Adapters (OA) - Provide the means by which object implementations **access most ORB services**. This includes the generation and interpretation of **object references, method invocation, security and activation**. The object adapter actually exports three different interfaces: **a private interface to skeletons, a private interface to the ORB core and a public interface used by implementations**. The OA **isolates the object implementation from the ORB core**.

Requests

The client **requests a service** from the object implementation. The **ORB transports the request**, which invokes the method using object adapters and the IDL skeleton.

The client has an object reference, an operation name and a set of parameters for the object and activates operations on this object. The Object Management Group / Object Model defines each operation to be associated with a controlling parameter, implemented in CORBA as an object reference. **The client does not know the location of the object or any of the implementation details**. The request is handled by the ORB, which must locate the target object and route the request to that object. It is also responsible for getting results back to the client.

Broker implementation in CORBA

Object Adapters

Object Adapters (OA) are the primary ORB service providers to object implementations. OA have a public interface which is used by the object implementation and a private interface that is used by the IDL skeleton.

Example services provided by OA's are:

- **Method invocation** (in conjunction with skeleton),
- Object implementation **activation and deactivation**,
- Mapping of object reference to object implementations,
- **Generation of object references**, and
- Registering object implementations, used in locating object implementations when a request comes in.

ORB Interface Operations

The ORB provides operations on **references and strings**. Two operations for converting an object reference to strings and back again. An **is_nil operation** for testing whether an object reference is **referencing no object**. A duplicate operation allows for a duplicate reference to the same object to be created. A release operation is provided to reclaim the memory used by an object reference.

Broker implementation in CORBA

Object Services

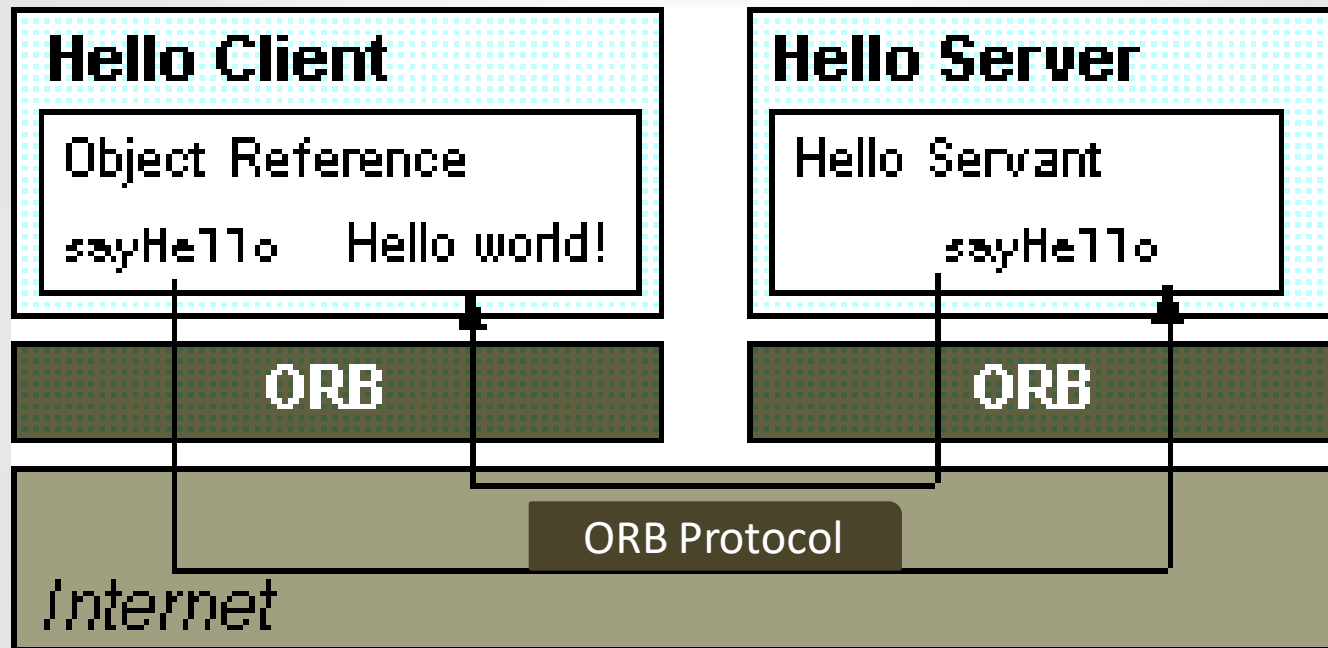
Object services refer to fundamental services provided by their own objects that support common interactions between other objects. The services follow the OMG Common Object Services Specification (COSS) guidelines.

Current object services include:

- **Event Management services** which refer to the asynchronous communication of different CORBA objects with one another.
- **Naming Object services** that maps a human-readable name (string) to an object relative to its context.
- **Persistent services** assure that an object (CORBA developed) outlives its creator. It allows an objects state to be retained at a given point in time. This feature is used when a host crashes or is rebooted.
- **Externalization services** collect objects and transport them as serial objects.
- **Life-cycle services** determine the methods for an object creation and termination.
- **Concurrency services** provide for distributed locks on a given object.
- **Relationship Objects** are used for CORBA object modeling and graphing.
- **Transaction object services** allow the sharing of a transaction among multiple objects.

Broker implementation in CORBA

Describe this architecture based on ORB style and mention its all objects services?



Service-Oriented Architecture

Cloud computing

The term **Cloud** refers to a **Network or Internet**. In other words, we can say that Cloud is something, which is present at **remote location**. Cloud can provide services over public and private networks, WAN, LAN or VPN.

Cloud computing is the **on-demand availability** of computer system resources, especially data storage and computing power, **without direct active management** by the user.

Cloud computing (also called simply, the cloud) describes the act of storing, managing and processing data online - as opposed to on your own physical computer or network.

Cloud computing is the delivery of **different services through the Internet**. These resources include tools and applications like data storage, servers, databases, networking, and software.

Cloud computing is the delivery of **on-demand computing services over the internet** on a **pay-as-you-go basis**.

Cloud computing

Types of Cloud Computing

Public Cloud: Multi-tenant environment with pay-as-you-grow scalability

A public cloud is built over the Internet and can be **accessed by any user** who **has paid for the service**. Public clouds are owned by service providers and are accessible through a subscription.

Many public clouds are available, including **Google App Engine (GAE)**, **Amazon Web Services (AWS)**, **Microsoft Azure**, IBM Blue Cloud, and Salesforce.com's Force.com.

Private Cloud: Scalability plus the enhanced security and control of a single-tenant environment

A private cloud is built within the domain of an intranet **owned by a single organization**. Therefore, it is client owned and managed, and its access is limited to the owning clients and their partners. Its deployment was not meant to sell capacity over the Internet through publicly accessible interfaces.

Types of Cloud Computing

Hybrid Cloud: Connect the public cloud to your private cloud or dedicated servers - even in your own data center.

Private clouds can also support a hybrid cloud model by supplementing local infrastructure with computing capacity from an external public cloud.

For example, the Research Compute Cloud (RC2) is a private cloud, built by IBM, that interconnects the computing and IT resources at eight IBM Research Centers scattered throughout the United States, Europe, and Asia. A hybrid cloud provides access to clients, the partner network, and third parties.

Cloud computing

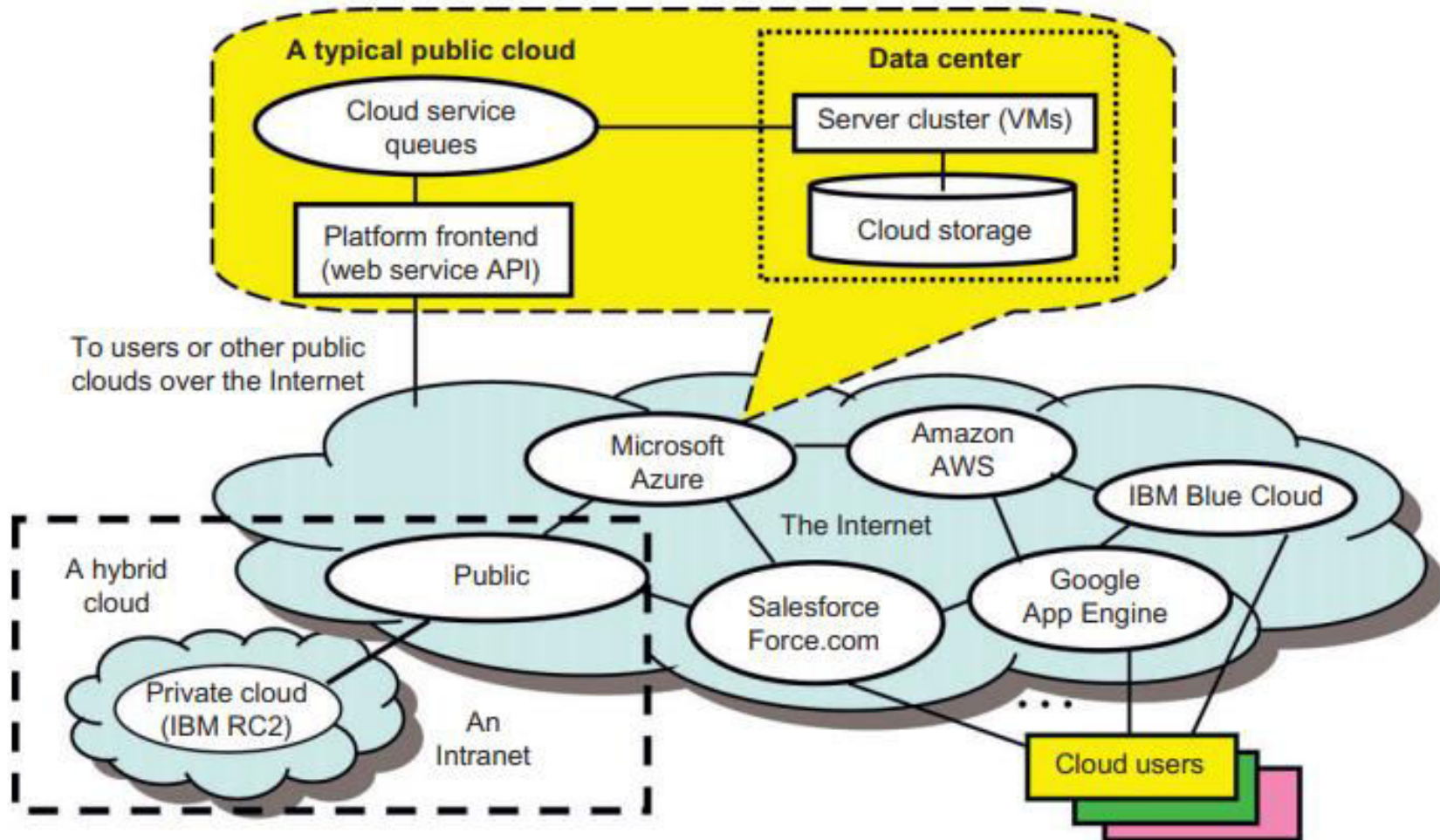
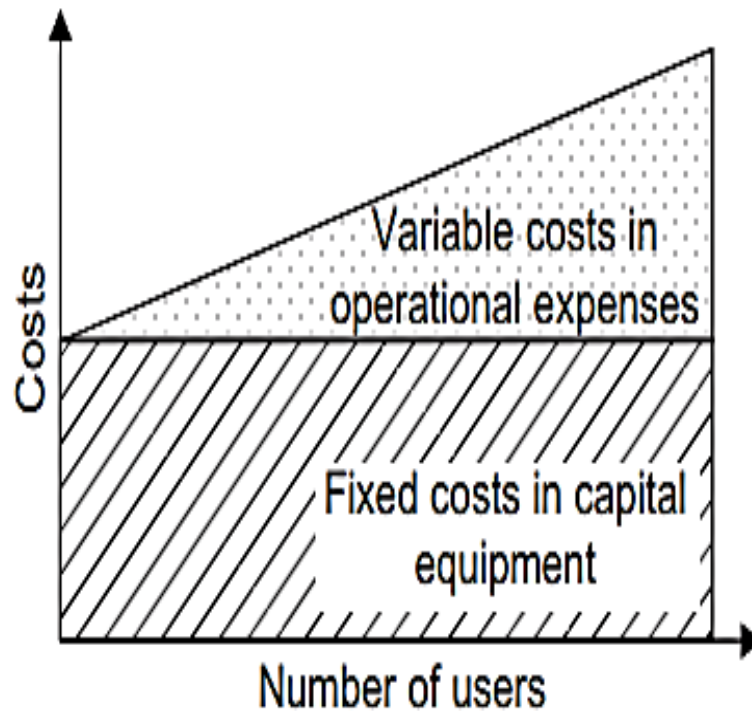


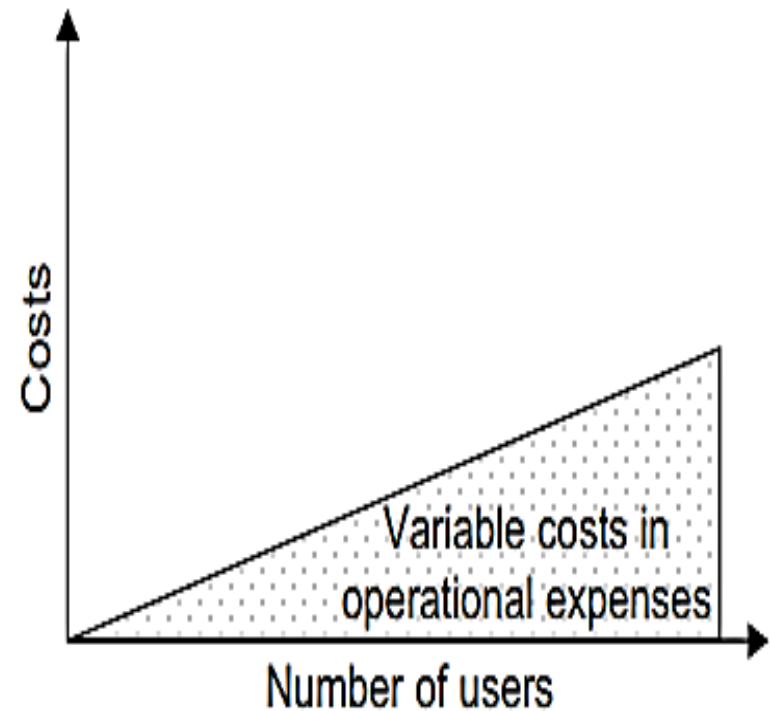
FIGURE 4.1

Public, private, and hybrid clouds illustrated by functional architecture and connectivity of representative clouds available by 2011.

Cloud computing : Cost



(a) Traditional IT cost model



(b) Cloud computing cost model

FIGURE 4.3

Computing economics between traditional IT users and cloud users.

Service Models

Cloud computing is based on service models. These are categorized into three basic service models which are –

- ❖ Infrastructure-as-a-Service (IaaS)
- ❖ Platform-as-a-Service (PaaS)
- ❖ Software-as-a-Service (SaaS)

Cloud computing

Infrastructure as a service (IaaS)

- Most basic cloud service model
- Cloud providers offer computers, as **physical** or more often as virtual machines, and other resources. This model allows users to use **virtualized IT resources** for computing, storage, and networking.
- Virtual machines are run as guests by a hypervisor, such as Xen or KVM.
- Cloud users deploy their applications by then installing operating system images on the machines as well as their application software.
- Cloud providers typically bill IaaS services on a utility computing basis, that is, **cost will reflect the amount of resources allocated and consumed**.

Examples of IaaS include: Amazon Cloud Formation (and underlying services such as Amazon **EC2**), Rackspace Cloud, Terremark, and **Google Compute Engine**

***A hypervisor is computer software, firmware or hardware that creates and runs virtual machines*

Cloud computing

Platform as a service (PaaS)

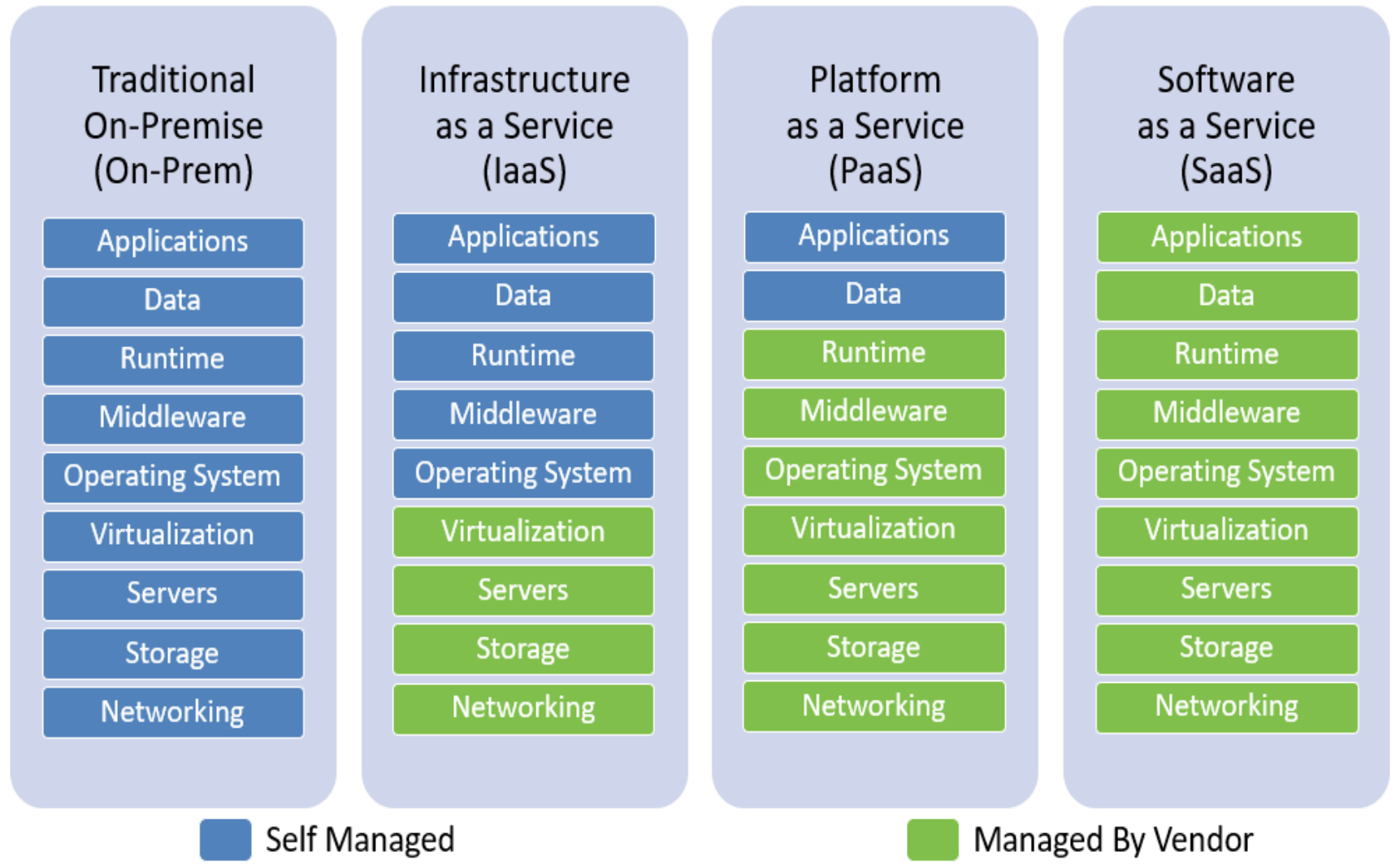
- Cloud providers deliver **a computing platform** typically including operating system, programming language execution environment, database, and web server.
- Application developers **develop and run their software** on a cloud platform without the cost and complexity of buying and managing the underlying hardware and software layers.
- Examples of PaaS include: Amazon Elastic Beanstalk, Cloud Foundry, Heroku, Force.com, EngineYard, Mendix, **Google App Engine**, Microsoft Azure and OrangeScape.

Software as a service (SaaS)

- Cloud providers install and operate application software in the cloud and cloud users access the software from cloud clients.
- The pricing model for SaaS applications is typically a monthly or yearly flat fee per user, so price is scalable and adjustable if users are added or removed at any point.
- **Examples of SaaS include:** Google Apps, innkeypos, Quickbooks Online, Limelight Video Platform, Salesforce.com, and Microsoft Office 365.

Cloud computing Services

Cloud Services



Service-Oriented Architecture (SOA)

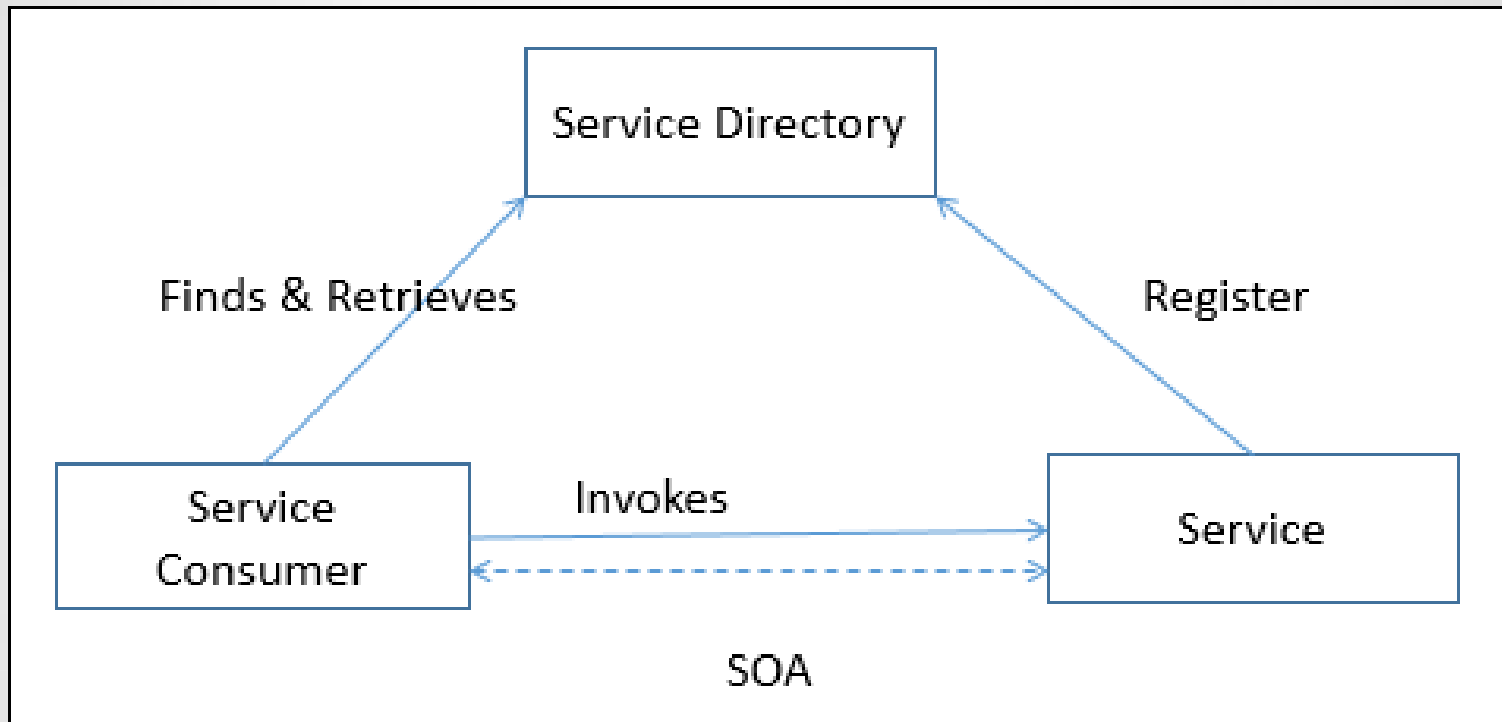
Service-Oriented Architecture (SOA) is a **style of software design** where **services are provided to the other components** by application components, through a **communication protocol over a network**.

There are two major roles within Service-oriented Architecture:

Service provider: The service provider is **the maintainer of the service** and the organization that makes available one or more services for others to use. To advertise services, the provider can publish them in a registry, together with a service contract that specifies the nature of the service, how to use it, the requirements for the service, and the fees charged.

Service consumer: The service consumer can locate the service metadata in the registry and develop the required client components to bind and use the service.

Service-Oriented Architecture (SOA)



Service-Oriented Architecture (SOA)

Why to use SOA?

- SOA is widely used in market which responds quickly and makes effective changes according to market situations.
- The SOA keep **secret the implementation details** of the subsystems.
- It allows interaction of new channels with customers, partners and suppliers.
- It authorizes the companies to select software or hardware of their choice as it acts as platform independence.

Features

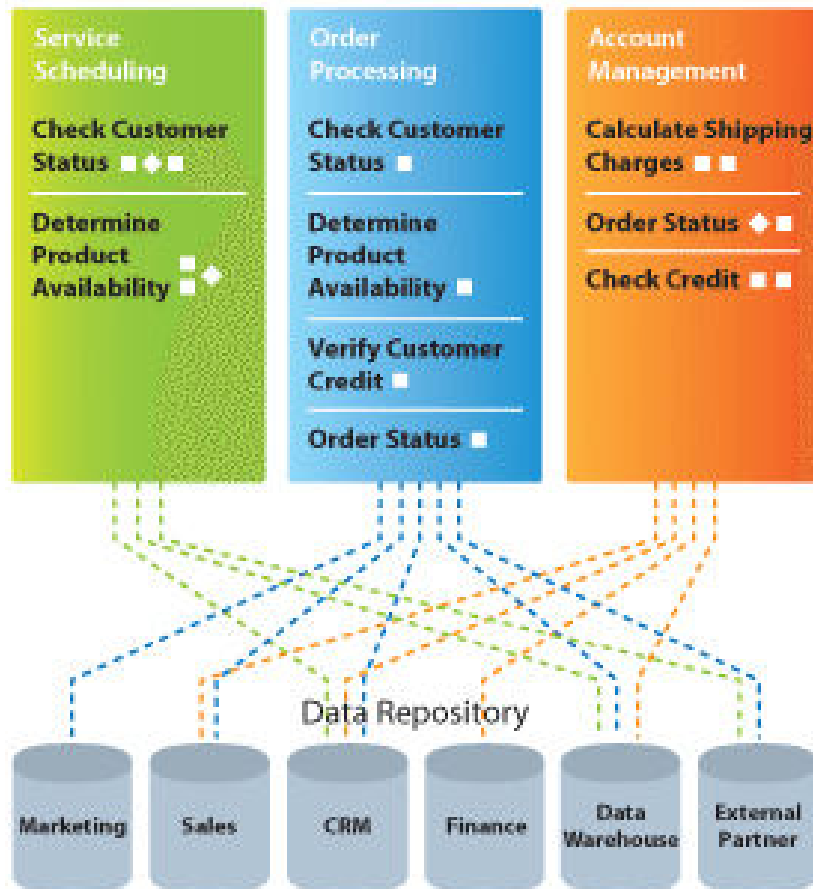
- SOA uses interfaces which **solves the difficult integration** problems in large systems.
- SOA communicates customers, providers and suppliers with messages by using the **XML schema**.
- It uses the message monitoring to improve the performance measurement and detects the security attacks.
- As it reuses the service, there will be lower software development and management costs.

Service-Oriented Architecture (SOA)

Before SOA

Closed - Monolithic - Brittle

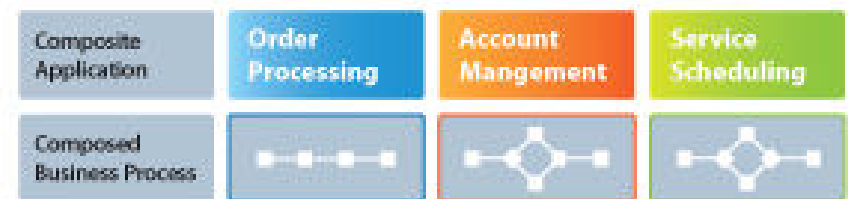
Application Dependent Business Functions



After SOA

Shared services - Collaborative - Interoperable - Integrated

Composite Applications



Reusable Business Services

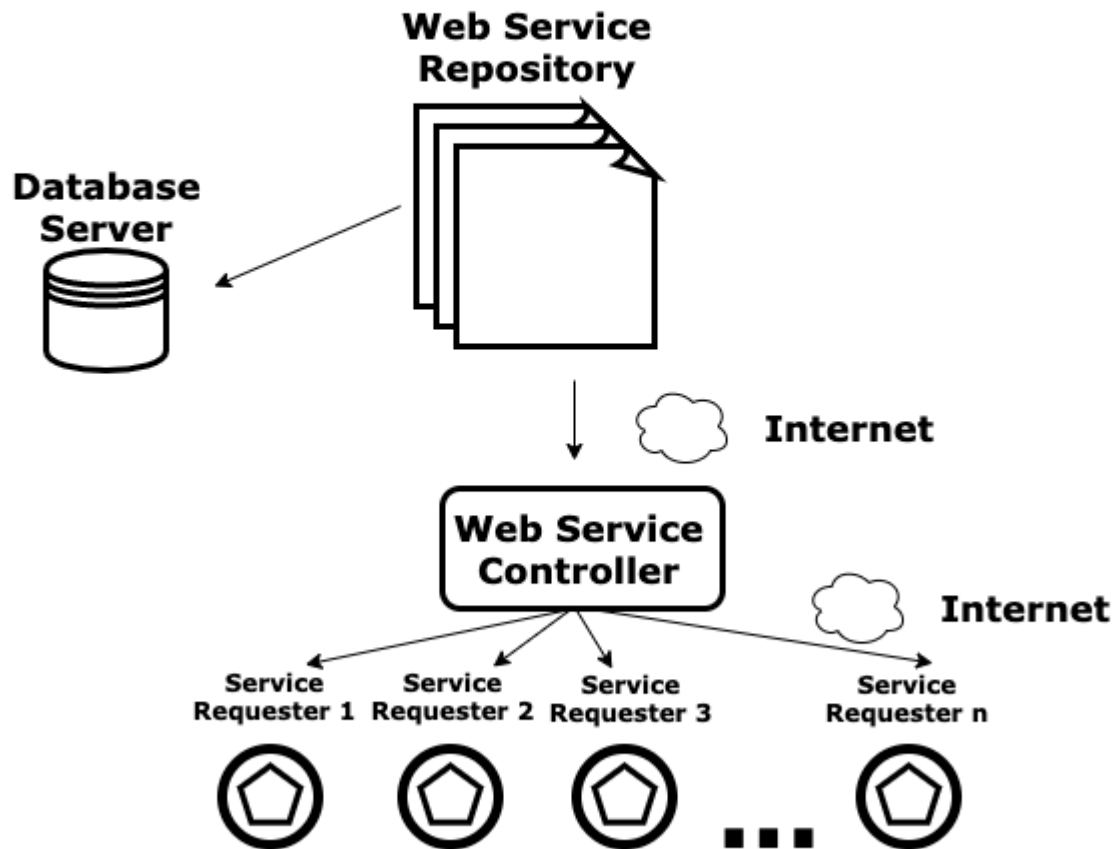


Data Repository



Service-Oriented Architecture (SOA)

Service Oriented Architecture High Level Diagram



Service-Oriented Architecture (SOA)

A standard client-server architecture has these parts:

Web service repository: This is a **library of web services** built to **serve external requests for information**. The served information is usually a little piece of information, like a number, a word, some variables, etc.

Web service controller: This module communicates the information in the web service repository with the **service requesters**. When an **external service requester calls** a certain function from the web service repository, the web service controller **interprets the call** and looks for the function in the web server repository. Then it executes the function and **returns a value to the requester**.

Service-Oriented Architecture (SOA)

Database server: This server contains the tables, indexes, and data managed by the core application. Searches and insert/delete/update operations are executed here.

Service requesters: These are **external applications that request services** from the web service repository through the internet, such as an organization requesting flight information from an airline or another company asking the package carrier for the location of a package at a given moment.

Note that service-oriented architecture is **created by the companies providing the service** - not the ones consuming it, and this is done to simplify the connections to possible clients.

Service-Oriented Architecture (SOA)

Advantages of SOA:

- **Service reusability:** In SOA, applications are made from existing services. Thus, services can be reused to make many applications.
- **Easy maintenance:** As services are independent of each other they can be updated and modified easily without affecting other services.
- **Platform independent:** SOA allows making a complex application by combining services picked from different sources, independent of the platform.
- **Availability:** SOA facilities are easily available to anyone on request.
- **Reliability:** SOA applications are more reliable because it is easy to debug small services rather than huge codes
- **Scalability:** Services can run on different servers within an environment, this increases scalability

Service-Oriented Architecture (SOA)

Disadvantages of SOA:

- **High overhead:** A validation of input parameters of services is done whenever services interact this **decreases performance as it increases load and response time.**
- **High investment:** A huge initial investment is required for SOA.
- **Complex service management:** When services interact they exchange messages to tasks. the number of messages may go in millions. It becomes a cumbersome task to handle a large number of messages.

Case-study: Solving a Business Problem With Service- Oriented Architecture

GlobalWeather is a global company that sells information about weather conditions. Here are some important facts:

- GlobalWeather is based in the U.S. and has more than 700 employees.
- GlobalWeather sells weather information to companies about countries, regions, cities, or even precise locations defined by latitude and longitude.
- GlobalWeather clients are all over the world and usually consume information using the internet. This is especially useful for transport companies: airlines, bus services, and logistics companies that have a truck fleet to distribute merchandise over a region. Additionally, any companies that coordinate outdoor events.
- Clients need information in real-time.

Case-study: Solving a Business Problem With Service- Oriented Architecture

What's the Business Problem?

GlobalWeather sells information to clients **in real-time**, but it cannot let all clients **access their internal systems** because of security constraints.

How can we solve this problem?

Let's look at the facts:

GlobalWeather has **thousands of clients**. To buy its services, each client needs to connect to GlobalWeather systems and get information manually. It is very **complex and time- consuming**, especially since the information requested is usually a little piece of information that can be transmitted via the internet.

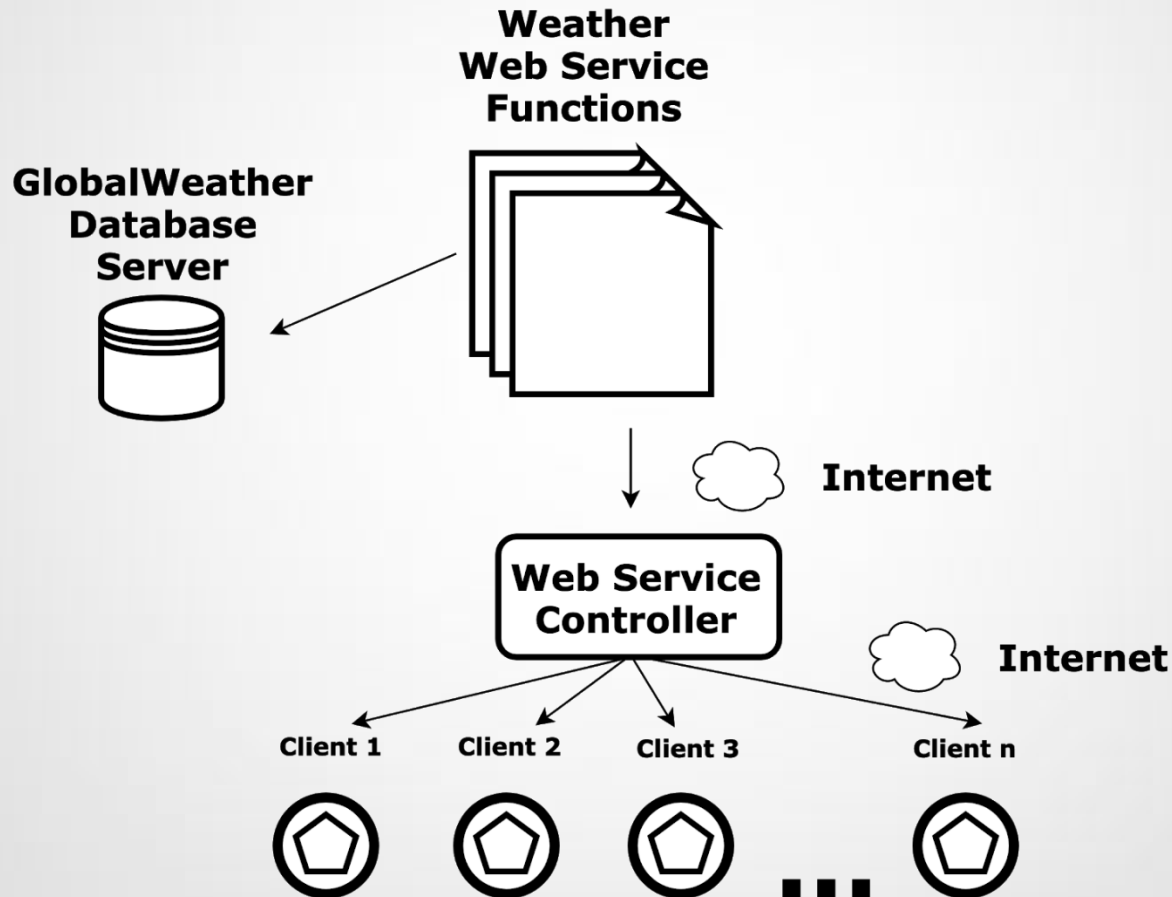
Additionally, many clients are **websites that need to connect** to the information feed and quickly display weather information in real-time. For example, an airline that is selling a ticket from city A to city B wants to display the weather in city B once the client buys that ticket.

This situation is the right situation for a **service-oriented architecture**: GlobalWeather can build a **library of functions** and implement it as web services that their clients consume once they pay. This ensures a speedy, real-time service without allowing clients to access internal servers.

Case-study: Solving a Business Problem With Service- Oriented Architecture

What's the Solution?

Service-Oriented Architecture GlobalWeather



Case-study: Solving a Business Problem With Service- Oriented Architecture

Component of the system:

- Weather web service functions:** This is a library of web services built to serve external websites with weather information such as rain, wind direction and speed, atmospheric pressure, etc.
- Web service controller:** This module communicates the web service repository with the service requesters. When an external service requester calls a certain function (for instance, flight information, weather information, package status) from the web service repository, the web service controller interprets the call and looks for the function in the web server repository. Then it executes the function and returns a value to the requester.
- Clients:** These are external applications that request services from the web service repository through the internet. These requests are coded into the requester's application according to a certain syntax published by the service provider.
- GobalWeather database server:** This server contains the tables, indexes, and data managed by the system.

Case-study: Home Work

You work at a major airport administration department in your country. You have been asked to develop a software system to give the status of a certain flight that is in the air to some possible requesters: airlines, travel sites, hotel sites, etc.

There is a central system at the airport control area that works in real-time: it receives information for the radars and aircrafts as the flights progress.

Now, it's your job to create an architecture for them!

Here are some **key questions** you can ask yourself to get started:

- I. Many flights are in the air at a certain moment. How can you get their flight status from the central system?
- II. How are you going to pass this information to the external requesters?
- III. Why does the airline need this system? Who is going to consume this information?

Can you produce an architecture diagram for this business setting?