

# Overview of Training

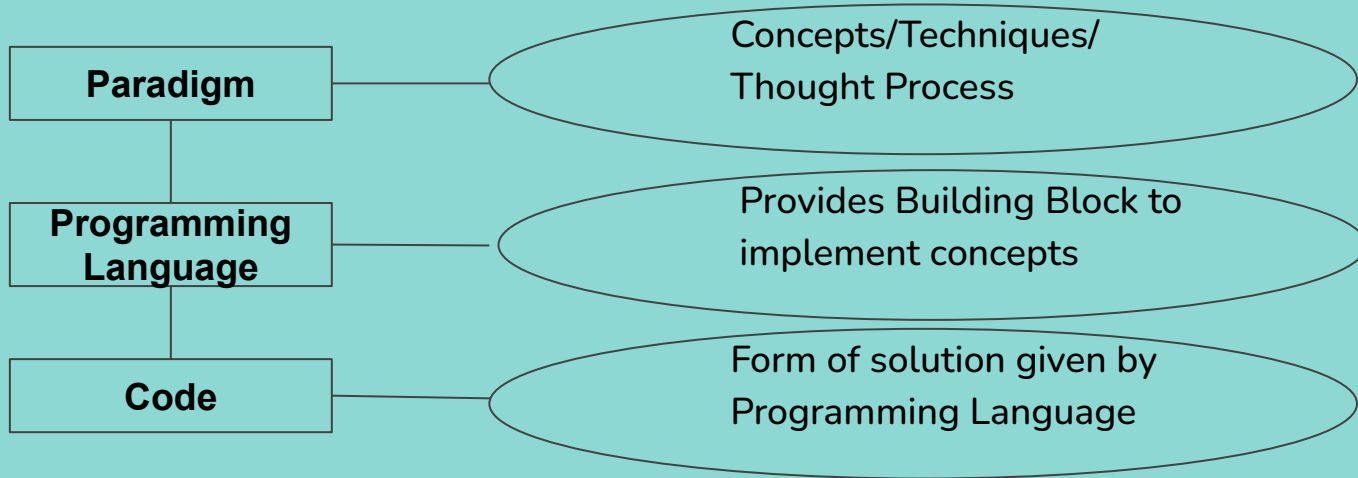


## Group Members:

Amulya Varshney  
Anupam Ujwane  
Shalini Lodhi  
Siddharth Singh  
Stuti  
Tushar Jain

# Programming Paradigm

- Paradigm is an approach or thought process which gives solution and concepts to problem statement.
- Select Paradigm-> Model Concepts -> Implementation.



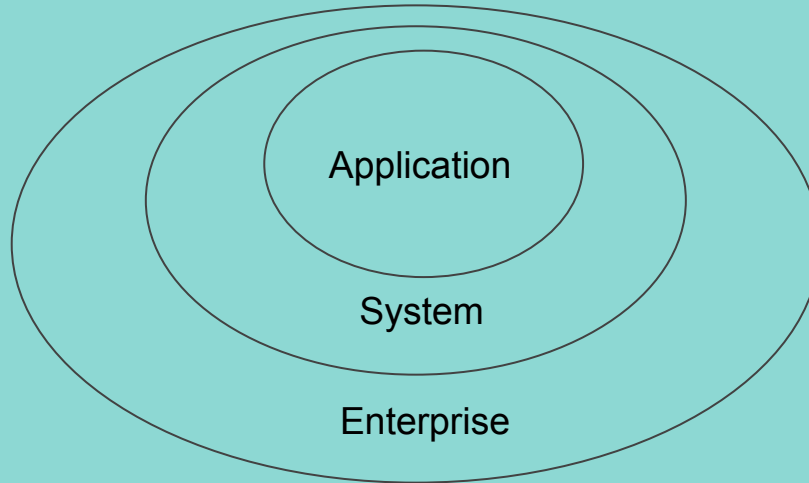


## Continued...

- **Functional Programming Paradigm** → To solve a problem, keep applying function to reduce the problem complexity. Here the output of the function is the input of the other function till we get solution.
  - No concept of object.
  - Good for parallel programming.
- **Concurrency**-> One thread is doing multiple tasks one after the other. There is switching of task during the wait time of the other task. Many tasks are done by one thread but only one task is done at a moment.
- **Parallelism**-> If more than one task is done by more than one threads at the same moment.

# Enterprise, System and Application

- **Enterprise:** Collection of systems which are integrated/collaborated.
- **System:** Cluster of applications.
- **Application:** It is a **Layered Architecture**.
  - Each layer has different functionalities.
  - Layers give logical bounding of functionalities which need to be separated.
  - Collection of components which belong to any of the layer.





# Data

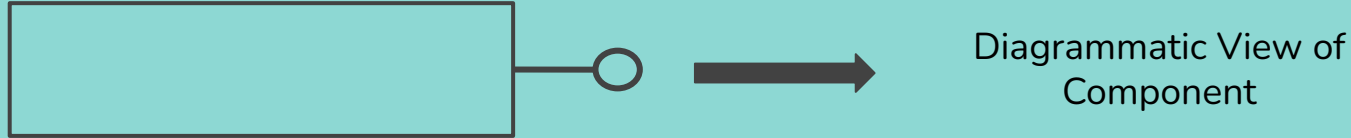
- An important part of any application is data.
- **Data:** There are three functionalities which deal with data.
  - **Persist:** Save Data in databases in:
    - **Transactional Relational Database Management System->** Structured Data is stored which follows ACID Properties.
    - **ACID->** Stands for Atomicity, Consistency, Isolation and Durability.
    - **No-SQL Database->** Unstructured Data is stored which follows doesn't follow ACID Properties it generally stores non-important data.
  - **Process:** Process Data or Perceive(Carried out by AI)
  - **Present:** Presenting Data Visualization



# Terminologies

- **Horizontal Partitioning:** By forming layers we partition application functionalities.
  - Number of layers depends on application requirements.
  - Even a smallest application would have Persist, Process and Present functionalities.
- Layers have functionalities which are coded; these codes of different layer are stored in different **Components**.
- **Monolithic Application:** When codes of every layer is kept in a single software component.
  - Its limitation is non-reusability.
  - Preferred by embedded systems due to space and communication constraints.
- **Application Execution:** An instance of application is created in a process block of operating system which bounds each instance.
- **Application Deployment:** Deliver components to machine (installed/installation).

# Components



**Component** is a reusable code/module which provides versionization.

- **Limitation of Component:** It could be reusable only in same runtime environment.
- **Architecture->** Systematic arrangement of components.

Components are Designed, Developed, Tested and Deployed.

**Low Level Design** → Component Design.

- To check the quality of code predefined workflows are built and these checks are automated.



# WorkFlow/Pipeline

Code passes through pipeline/workflow for checks. It's a collection of jobs like:

- Static Code Analysis
  - Unit Testing
- Pipeline contains of two steps.
  - **Build**-> Code goes first here and generates **artifacts** means it's ready to deliver.
  - **Release**-> Artifacts become input for this and release deploy code to respective deployment location.
- This is called **Continuous Integration Continuous Deployment(CI/CD)**
- Code is built twice:
  - In Local Machine
  - Build pipeline in repository
- An instance of application is created in a process block of operating system which bounds each instance.





# Deployment Topology

- **Monolithic/Tier 1:** one process block of operating system is needed to execute one instance of application.
- **Tier 2:** Two process block of operating system is needed to execute one instance of application.

Here few components of application runs on one process block and rest on the other

- **Client Server Architecture:** Minimum two process blocks required.

The application shouldn't be monolithic. It is also called distributed computing as components execution are distributed.

Two process blocks communicate via RCP or IPC mechanisms.



# Microservice Architecture

- Splitting of application is done based on functionality point of view.
- Separate functions of application are stored in independent environment. Here the functionalities are delivered separately.
- Example: In Client Server Application there is one server and one client program.



# Server

Server is a program running in one operating system. It is a continuous running process which accept requests through socket using a standard protocol and an unique port number.

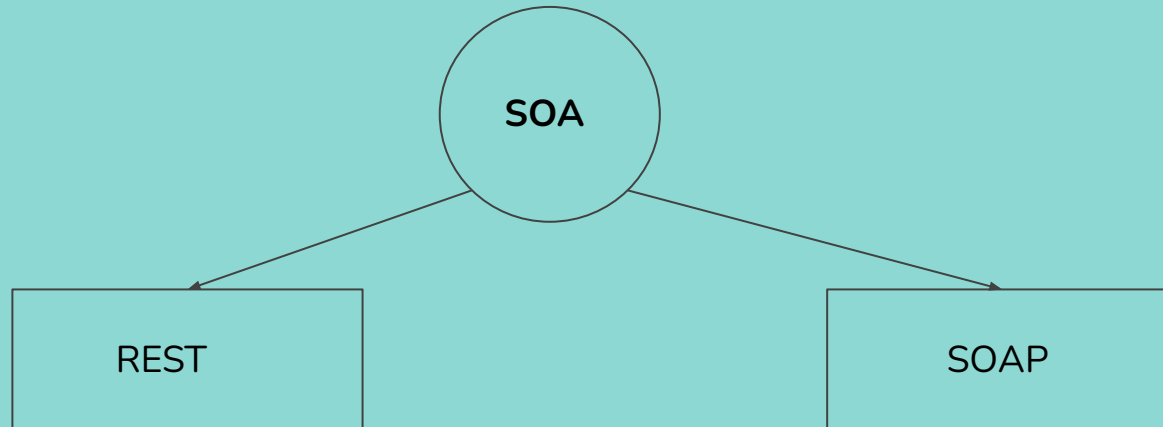
- **Protocol:** It defines in which format it receives input and in which format it gives output.
- Port number uniquely identifies the process.
- **Server Capacity:** At a time number of requests a server could accept.
- **Scalability:** Increasing the load a server can take at a time.
- **Vertical Scalability:** Increase server capacity but there is a limit to increase it.
- **Horizontal Scalability:** Run multiple instances of server using load balancer which sends incoming requests based on capacity of each instance. It is also called **cloning/replication**

## What if all the components of the system are not closely related and are written in different technological stack?

→ We opt an architecture which treats the components as services and is known as

### “Service Oriented Architecture(SOA)”

- SOA enables the components to become available for other runtimes within the Internet⇒ SOA is **runtime independent**
- All the microservice architecture adopts SOA.
- IPC/RPC is used for communication between two components.



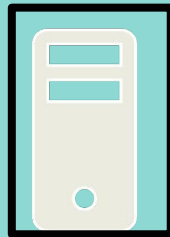
# REST Architecture

*"Representational State Transfer" is intended to evoke an image of how a well-designed Web application behaves: a network of web pages, where the user progresses by selecting links, resulting in the next page being transferred to the user and rendered for their use.*

# What does REST mean?



Client **GET** /users/2

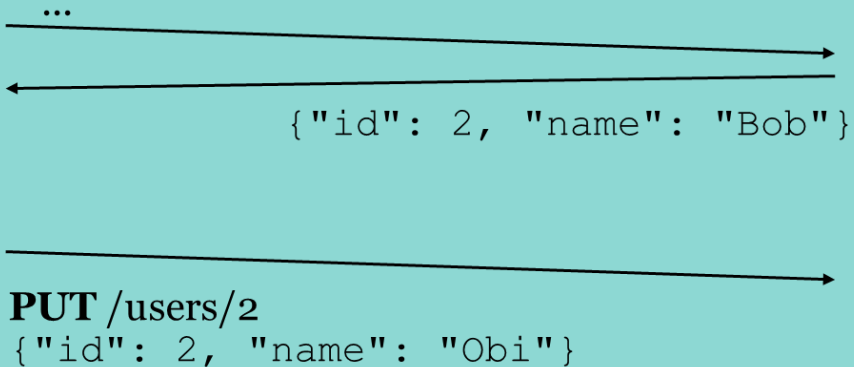


Server

Users	
Id	Name
1	Alice
2	Bob
3	Claire

Changes state.

```
{"id": 2,  
"name":  
"Obi"}
```



# Using HTTP as the uniform interface

- Use URIs to identify resources.
- Use HTTP methods to specify operation:
  - Create: POST (or PUT)
  - Retrieve: GET
  - Update: PUT (or PATCH)
  - Delete: DELETE
- Use HTTP headers

`Content-Type` and `Accept` to specify data format for the resources.

- Use HTTP status code to indicate success/failure.

# Continued...

REST is an architectural style, not a specification.

- In practice, it can be used in many different ways.
- But some are better than others.



# REST EXAMPLE

A server with information about users.

- The GET method is used to retrieve resources.

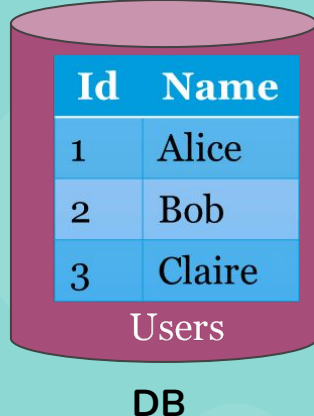
- GET /users
- GET /users/2
- GET /users/pages/1
- GET /users/gender/female
- GET /users/age/18
- GET /users/???
- GET /users/2/name
- GET /users/2/pets

GET /users?page=1

GET /users?gender=female

GET /users?age=18

GET /users?gender=female&age=18



Id	Name
1	Alice
2	Bob
3	Claire

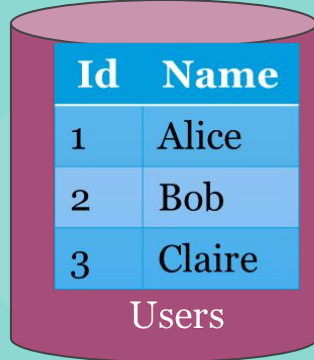
Users

DB

# REST EXAMPLE

A server with information about users.

- The POST method is used to create resources.
  - Which data format? Specified by the Accept and Content-Type header!



Id	Name
1	Alice
2	Bob
3	Claire

Users

```
POST /users HTTP/1.1
Host: the-website.com
Accept: application/json
Content-Type: application/xml
Content-Length: 49
```

```
<user>
  <name>Claire</name>
</user>
```

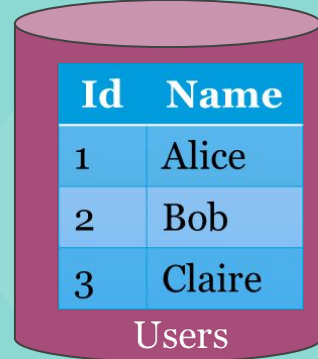
```
HTTP/1.1 201 Created
Location: /users/3
Content-Type: application/json
Content-Length: 28
```

```
{"id": 3, "name": "Claire"}
```

# REST EXAMPLE

A server with information about users.

- The PUT method is used to update an entire resource.



Id	Name
1	Alice
2	Bob
3	Claire

Users

```
PUT /users/3 HTTP/1.1
Host: the-website.com
Content-Type: application/xml
Content-Length: 52
```

```
<user>
  <id>3</id>
  <name>Cecilia</name>
</user>
```

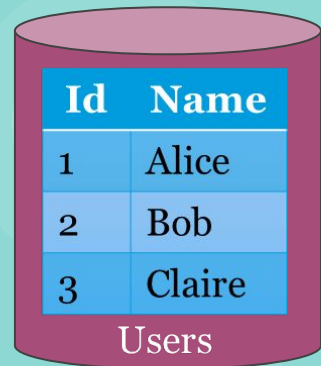
```
HTTP/1.1 204 No Content
```

PUT can also be used to create a resource if you know which URI it should have in advance.

# REST EXAMPLE

A server with information about users.

- The DELETE method is used to delete a resource.



Id	Name
1	Alice
2	Bob
3	Claire

Users

```
DELETE /users/2 HTTP/1.1
```

```
Host: the-website.com
```

```
HTTP/1.1 204 No Content
```

# REST EXAMPLE

A server with information about users.

- The PATCH method is used to update parts of a resource.



Id	Name
1	Alice
2	Bob
3	Claire

Users

```
PATCH /users/1 HTTP/1.1
Host: the-website.com
Content-Type: application/xml
Content-Length: 37
```

```
<user>
  <name>Amanda</human>
</user>
```

```
HTTP/1.1 204 No Content
```



The PATCH  
method is only a  
proposed standard.

# REST EXAMPLE

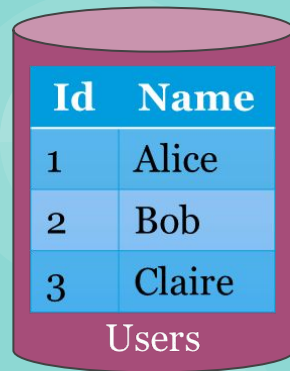
A server with information about users.

- What if something goes wrong?
  - Use the HTTP status codes to indicate success/failure.

```
GET /users/999 HTTP/1.1  
Host: the-website.com  
Accept: application/json
```

```
HTTP/1.1 404 Not Found
```

- Optionally include error messages in the response body.





Id	Name
1	Alice
2	Bob
3	Claire

Users

**HTTP uses request reply channel. It is half duplex.**

**For duplex communication we use web sockets!**

 HTTP	 WebSocket
Duplex	
Half	Full
Messaging Pattern	
Request-reponse	Bi-directional
Service Push	
Not natively supported. Client polling or streaming download techniques used.	Core feature

# SOAP Architecture

- XML based architecture, Transport independent (HTTP,TCP, etc.)
- Contains envelope which consists values of arguments of method and tells which method to call.
- Client sends request as a message and receives response as a message in an envelope.

**<envelope>**

**<header> “SOAP Action” </header>**

**<body>**

**(Parameters of the method)**

**</body>**

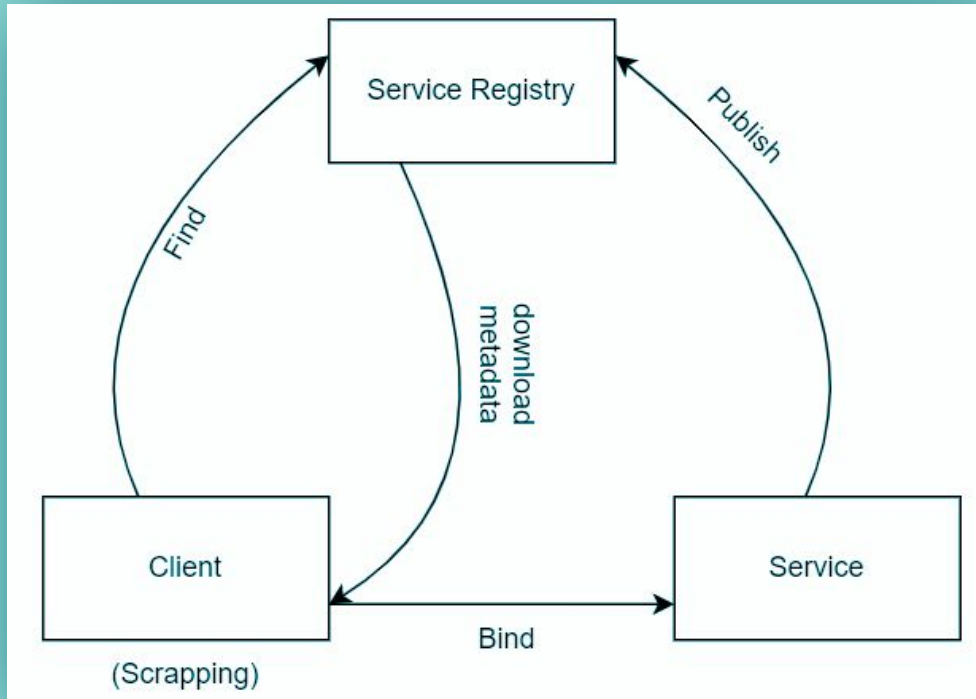
**</envelope>**



**SOAP  
message  
structure**



- SOAP Request and Response Message carries the payload(xml file).
- SOAP uses WSDL(Web services description Language) document.
- WSDL assists what message to exchange, how to exchange(transport) and address of delivery.



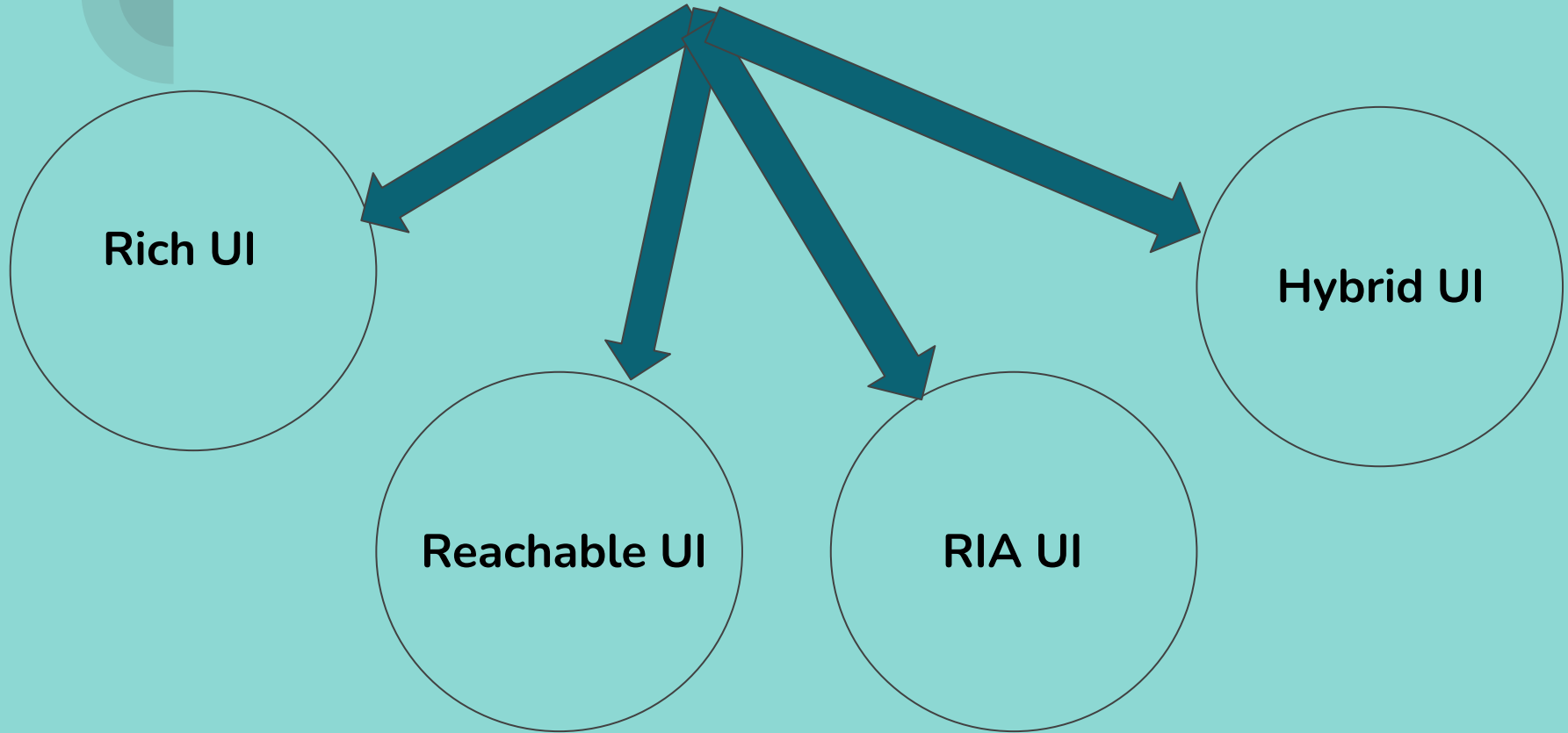
 **Publish-Find Bind Architecture**

# REST v/s SOAP

- Representational State Transfer
  - Transport Dependent(HTTP)
  - Have transport level security
  - Many formats supported :  
HTML, JSON, XML,etc.
  - Not suitable for distributed environment
- Simple Object Access Protocol
  - Transport Independent  
(HTTP,SMTP,UDP,etc.)
  - Have message level security
  - XML based
  - Supports distributed environment

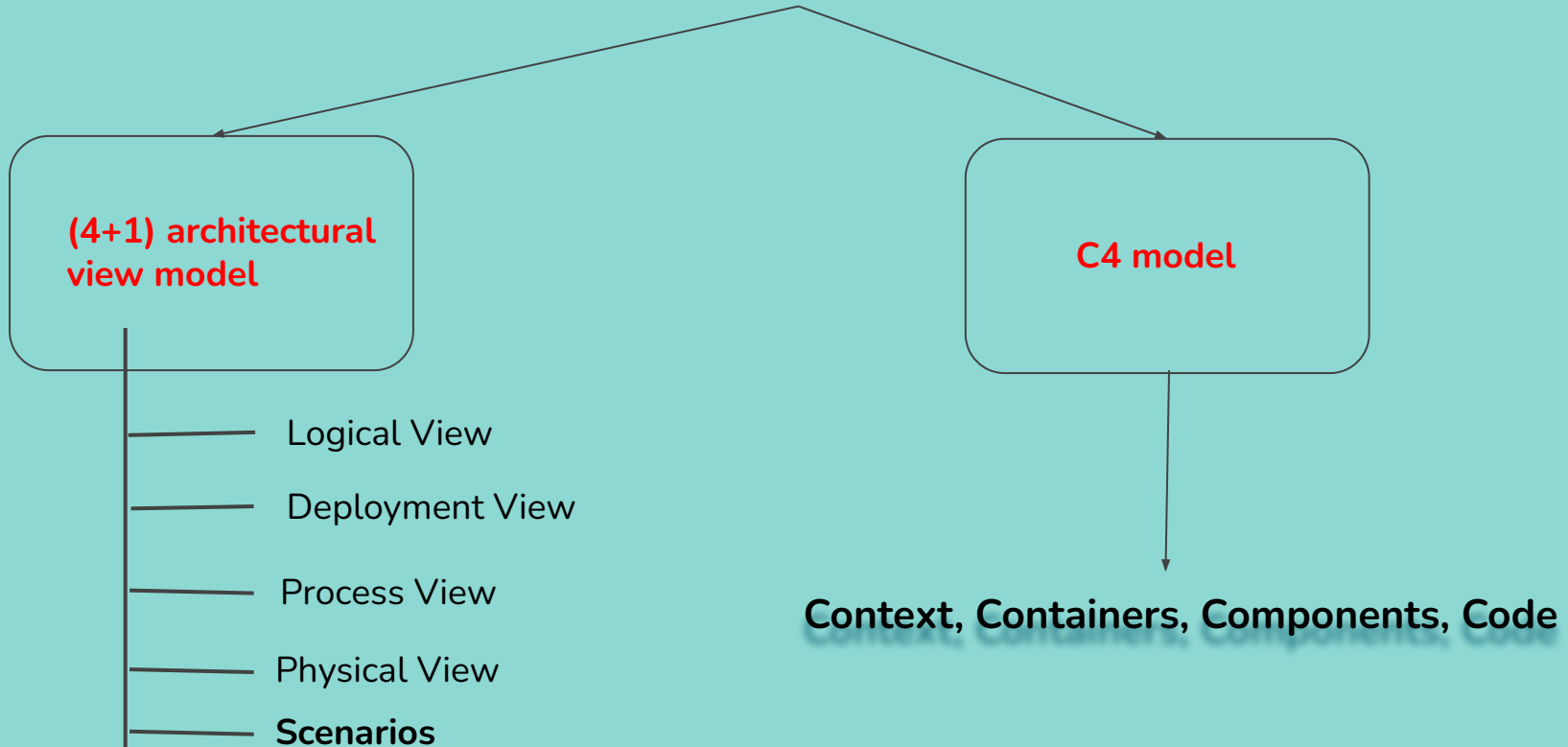
SOAP has demerits like it has heavy communication( as envelope is bulky), large bandwidth, delay or latency but it is more secure than REST architecture and is preferred for policy-based communication.

# Presentation Layer



# HOW TO PRESENT DECISIONS?

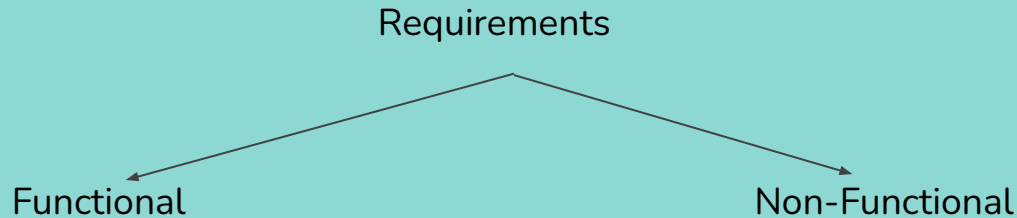
The answer is **Service Design Document (View Model Architecture)**





## (4+1) architectural view Model

- Using multiple concurrent views, architecture of a system is described.
- Every system has different viewpoints, thus creating different views is essential for different stakeholders.
- Classification of requirements is the major step in this type of model architecture.



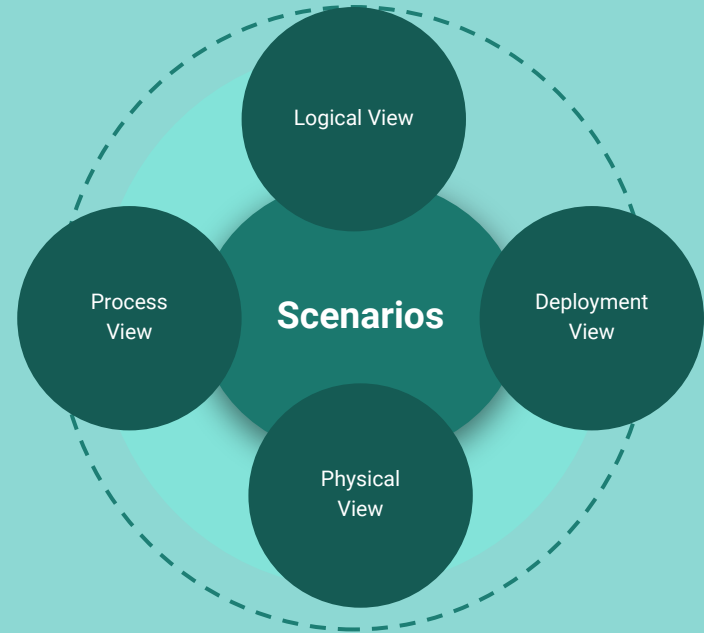
# Different views in (4+1) Model

**Logical View:** Performs horizontal(make layers of subsystem) and vertical(split systems to subsystems) partitioning of the system.

**Physical View:** Express how system is deployed.

**Process View:** Express non functional requirements.

**Deployment View :** Focuses on software management and presents a system from the viewpoint of a programmer.



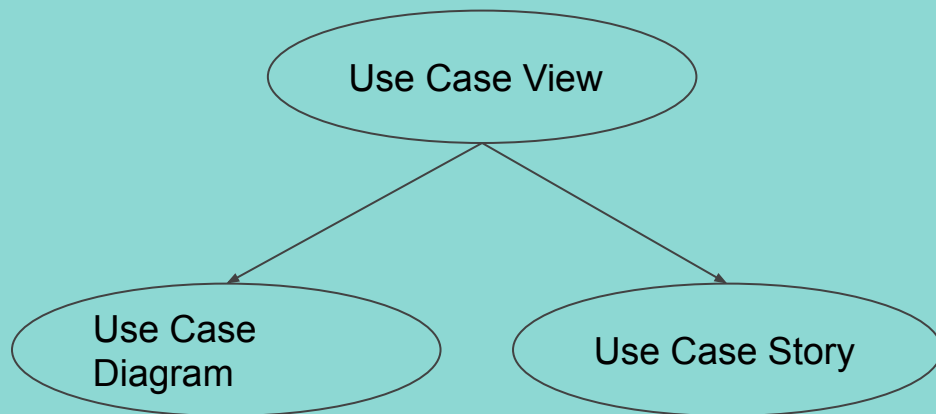


# Use Case View

**Use Case View** outlines the steps that go into interactions between processes and objects.

Documents all the functional requirements.

- Every system has its **Boundary** and end users(**actor**) interacts with the system.
- What actors are allowed to do with the system uses “include” keyword and all the use cases start with a verb.





# Object Oriented Paradigm

When more than one object collaborate to solve a problem.

Key features of Object Oriented Paradigm are:

- Used to reduce the complexity of code.
  - Easily maintainable.
  - Not good for Parallel Programming.
  - Security is not high.
- 
- Object is a real world entity which exists in a business domain carrying few behaviors of its own.
  - Classification of an object is called encapsulation. Encapsulation could be done on Object, Function and Component.





# Encapsulation

**SRP( Single Responsibility Principle):** It is a principle on how to correctly classify an object. It says “**Define a role and assign a single responsibility**” if SRP is violated it leads to god object. There must be uniform distribution of responsibilities.

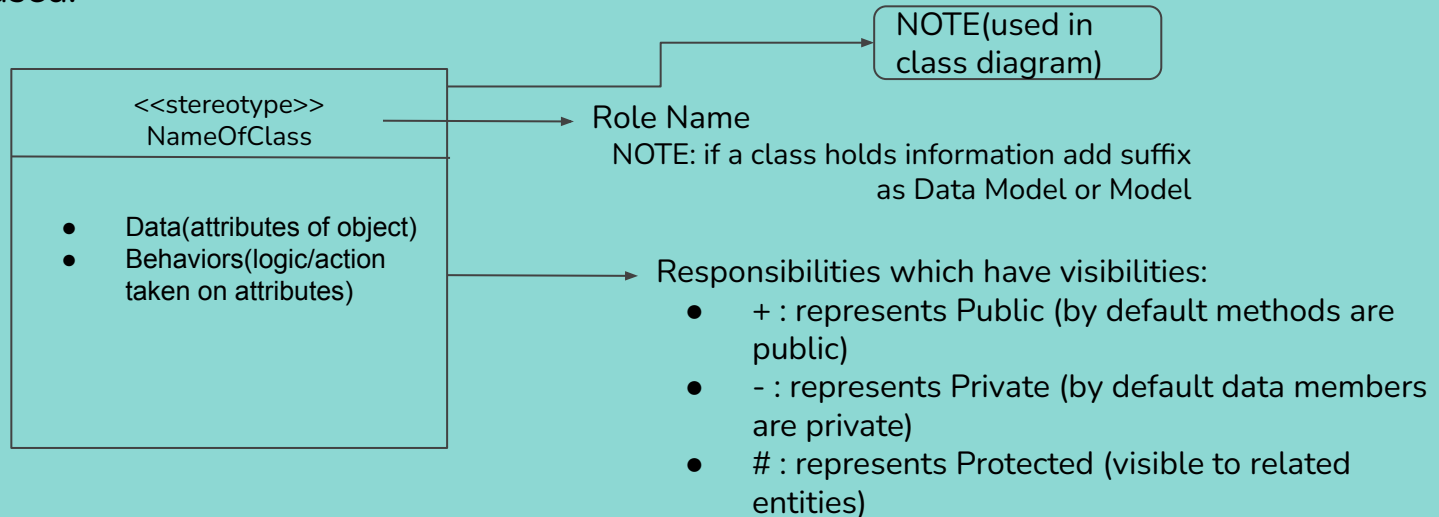
- Any method, class or object must have one reason to change if its more than one then its violation of SRP.
- **God Object:** An object which reference large number of distinct types and has too many unrelated or uncategorized methods.
- A software quality is measured on two things which are how adaptable it is and how bug free it is.
- In the case of god object it is neither easy to change nor it is bug free as it is not unit testable.



# Class Diagram

**Note** used in class diagram for a particular class provides description about data implementation(how function is implemented). It has comments, text and code.

While designing the class diagram to know the flow of code an **Activity Diagram** is used.





# Key Terms

**Facade Pattern** is the front face which interacts to other objects.

**Unit testing** is a software development process in which the smallest testable parts of an application, called units, are individually and independently scrutinized for proper operation.

**Refactor** is a way to find god classes/objects.



# Understanding OOP

Object Creation : space allocated for object's data member by constructor.

Class: Template of memory.

Object: Instance of class.

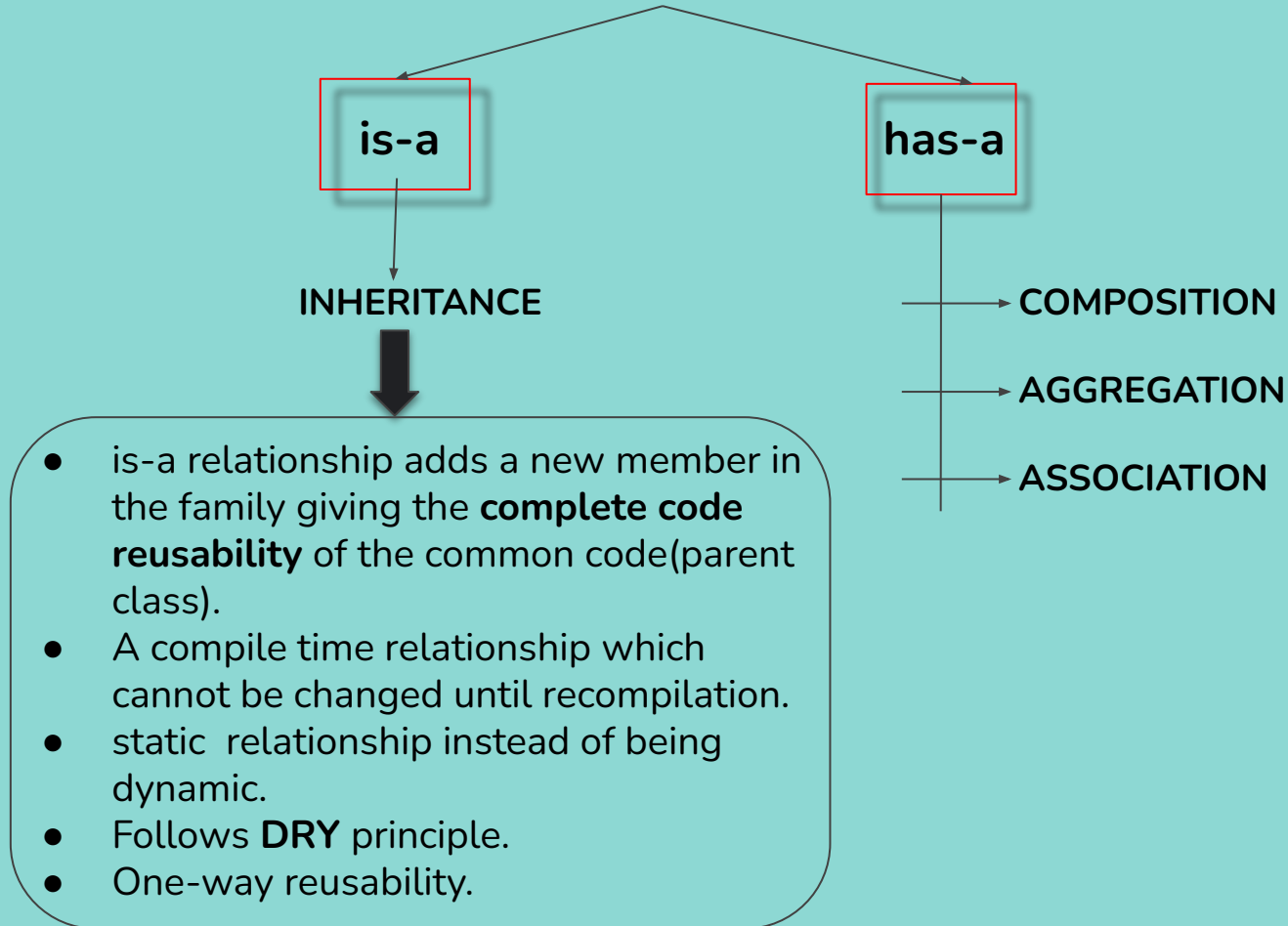
- Each object has a copy of its data members but its methods are shared with other object of same class.
- To execute a program thread is required which runs code function by function thus following procedural paradigm(runtime based).
- Every thread has its own stack.
- Thread is stateless so it needs bookmarks of function(at visited) which are ordered in stack => stack trace.
- At least one thread is required for running a program.



# Current Thread Stack Memory

- Contains local variables, return value, arguments of methods.
- Copies of arguments and local variables are thread safe means they are only shared in threads.
- No. of threads present is degree of parallelism.
- Every program starts with one thread(main thread) and then threads add other threads based on concurrency and parallelism.
- Every data in program is associated with lifetime - static (specific time) or dynamic (no specific time)

# Relationships in OOPs





# Classification of has-a relationship

**COMPOSITION** (contains) → Lifetime Dependent / Death Relationship

**AGGREGATION** (holds) → Lifetime Independent

**ASSOCIATION** (belongs to) → Using Referential Attributes, Related through 3rd Party Classes.



# Lifetime of Data

## Short lived

Once thread finish executing method, it is removed from the stack

## Long lived

Till that program ends, global and static variables (raw memory)

## Dynamic

Allocated in Heap, we control the lifetime of data

- Runtime creates thread. Every thread has its own stack, local variables must be initialized else thread fills garbage value.
- Object orientation in top of threading:
- As thread only knows stack but object is kept in heap, so with help of pointers.





# Parallelism in OOP

- If class method is used by many threads, only one thread can access same heap resource it leads to concurrency issues so a lock must be added which is called thread synchronization but it reduces throughput.
- Objects are shared between threads
- This is why OOP is not parallel.
- In object orientation instance, function are not pure and that is a reason they are not good for concurrency and parallelism.



# Functional Programming

## Pure Functions:

- Return values are identical for identical arguments and function has no side effects.
- Immutable (function never changes arguments)
- No side effects as it always depends on its arguments and they are predictable.
- If same argument given again, the output can be in cache without executing the method → highly performance oriented making it uni-testable.
- 
- In OOP every function has an argument “this” if we don’t want that argument mark the method as “static”



# Coupling

How independent different part of system are.

**Tightly Coupling** : classes and object are dependent on one another.

- It reduces flexibility and reusability of code
- Not test friendly

**Loose Coupling** : reducing dependencies of class that uses different class directly

**Dependency Injection** : injection of required dependency in fake environment.

**Runtime Polymorphism** : dependency exists in many forms & during run time program decides which form of dependency to choose

# Module

**High Level Module** : class/object depending on other object.

**Low Level Module** : class/object which is being used.

Dependency Inversion Principle

high level module should not depend on low level module but on it's abstracts.

# Abstraction

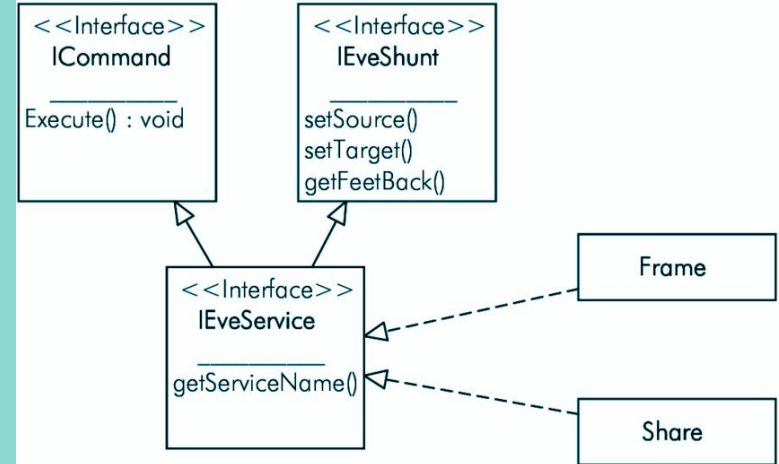
Knowing the essential details; need of runtime polymorphism (OOPs)

**Runtime Polymorphism** (Object Oriented Programming Paradigm)

**Function Pointer** (Function Oriented Programming Paradigm)

## Interface

- Collection of method signature; whole dependencies belongs to drive class.
- Members required by high level module from low level module.
- One object implement multiple interface/views.
- Can't be instantiated
- Interface only has essential methods which are accepted by High Level Module



## Types of Dependency Injection

```
graph LR; A[Types of Dependency Injection] --> B[Interface (Type 1)]; A --> C[Setter (Type 2)]; A --> D[Constructor (Type 3)];
```

### Interface (Type 1)

the dependency provides an injector method that will inject the dependency into any client passed to it

### Setter (Type 2)

should use setter injection in case the dependency is truly optional

### Constructor (Type 3)

process of using the constructor to pass in the dependencies of a class

## Dependency Inversion Principle

- *High-level modules/classes should not depend on low-level modules/classes. Both should depend upon abstractions.*
- *Abstractions should not depend upon details. Details should depend upon abstractions.*

## Single Responsibility Principle

*a class should have only one reason to change*

## Open/Closed Principle

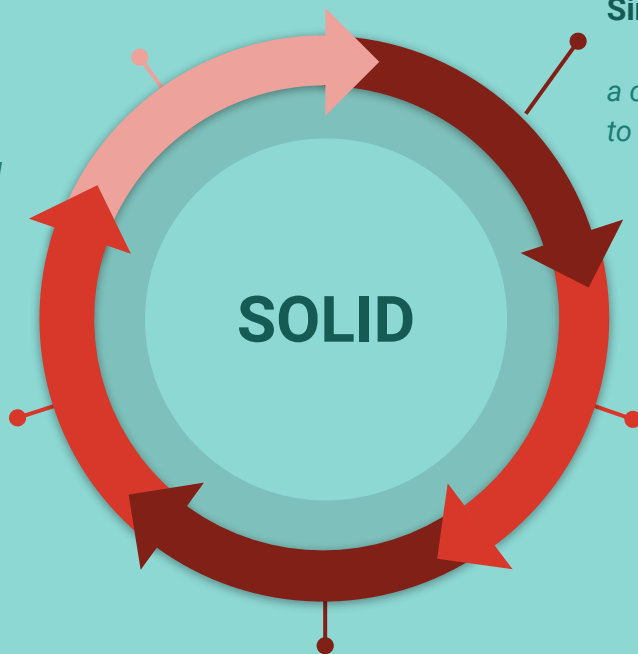
*software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.*

## Liskov Substitution Principle (LSP)

*Derived or child classes must be substitutable for their base or parent classes*

## Interface Segregation Principle

*do not force any client to implement an interface which is irrelevant to them. Default Interface is splitted into multiple views*



# Abstract Class & Interface

- contains only abstract methods.
- Only final and static variables are used.
- The interface can be declared with the interface keyword.
- It has class members public by default.

**Abstract  
class**

- contain both abstract and non-abstract methods.
- can have all four; static, non-static and final, non-final variables.
- To declare abstract class abstract keywords are used.
- It has class members like private and protected, etc

**Interface**

# FUNCTION

Chaining

Output of one function is input of another function.

Pointer

Assign function to variable as that it could be passed as function argument or returned by another function.

Closures

Concept which explains runtime we want local arguments and variable of function to be preserved even after execution of function.





# Asynchronous Programming



*a technique that enables your program to start a potentially long-running task and still be able to be responsive to other events while that task runs, rather than having to wait until that task has finished.*

- UI thread create new thread (worker thread) which interacts and get in wait but UI thread is still interactive so UI doesn't freeze.
- When worker thread completes task it remembers required object (lifetime till function executes) which is callback address and the callback method is invoked.

## UI Thread

UI is always single threaded; one thread in UI can interact with user, if thread busy UI will freeze.

# Financial Risk System

Customer Theory



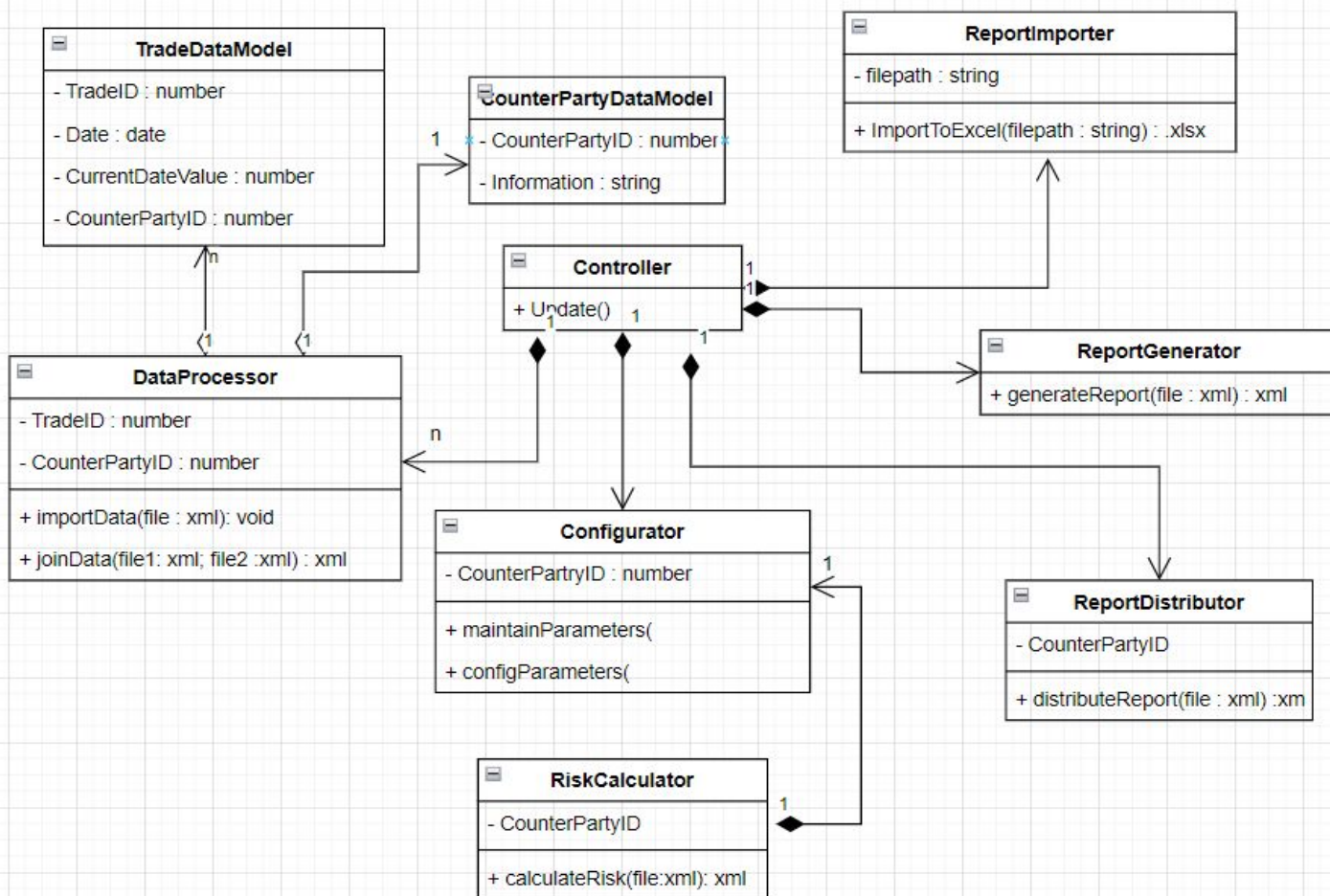
# Functional Requirements

- Import trade data from the TDS in file-based XML format
- Import counterparty data from the RDS in file-based XML format
- Join the trade data and counterparty data
- Enrich trade data with counterparty information
- Calculate risk for each counterparty
- Generate a report in Microsoft Excel format containing risk figures for all counterparties
- Distribute the report before 9am of the next trading day in Singapore
- Provide a way for business users to configure and maintain the external parameters used by the risk calculations

# Non-Functional

- The system is able to handle large volumes of data
- The system is able to process data quickly and efficiently
- The system is able to generate the report before the start of the next trading day in Singapore (Timezone 13 hours ahead of New York)
- highly secure to protect sensitive financial data
- user-friendly and easy to use for business users

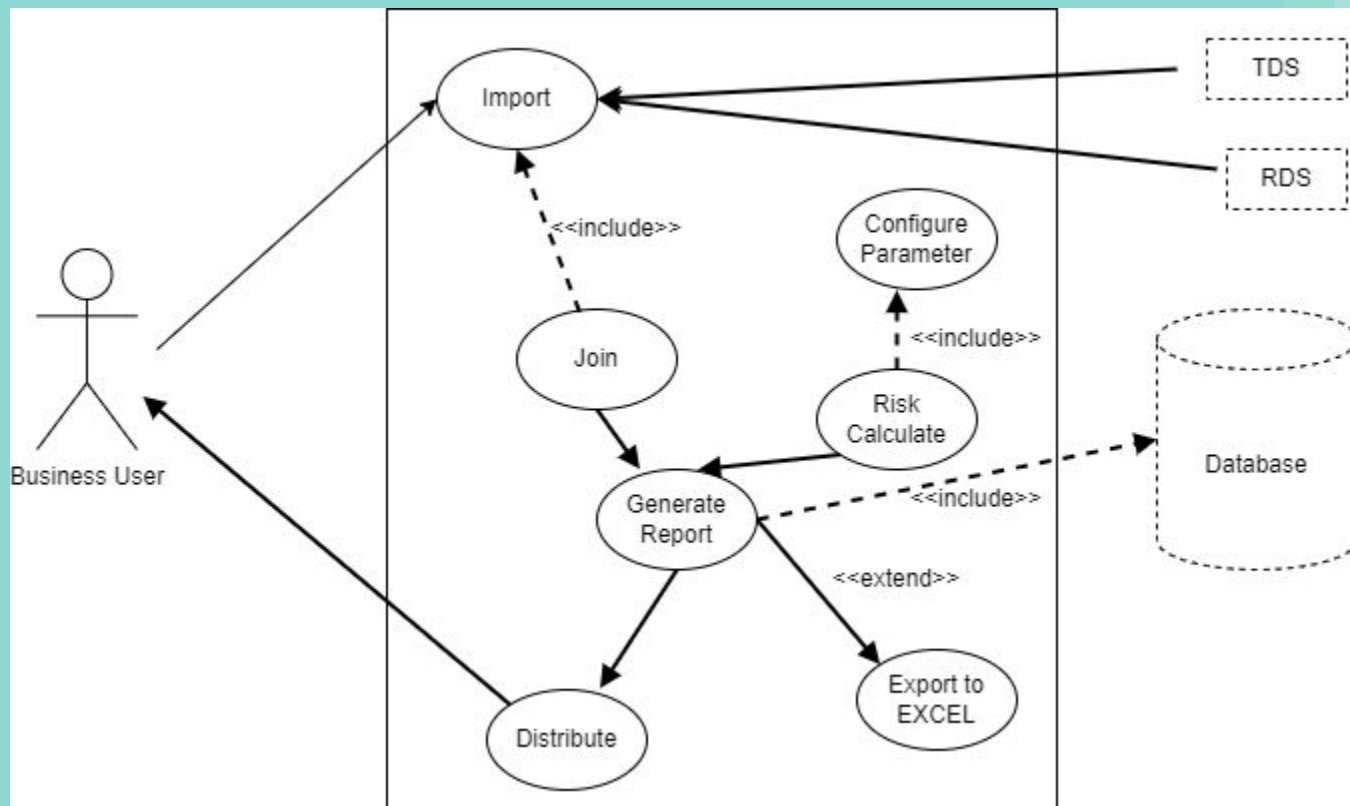
# Class Diagram



# Use Cases

- A business user imports trade data from the Trade Data System
- A business user imports counterparty data from the Reference Data System
- The system joins the trade data and counterparty data
- The system enriches trade data with counterparty information
- The system calculates risk for each counterparty
- The system generates a report in Microsoft Excel format containing risk figures for all counterparties
- A business user distributes the report before the start of the next trading day in Singapore
- A business user configures and maintains the external parameters used by the risk calculations

# Use Case Diagram



01

## Import trade data

- **Given** the Trade Data System is configured to generate a file-based XML export at the close of business
- **When** a business user imports the trade data
- **Then** the system should successfully import the trade data in file-based XML format

02

## Import counterparty data

- **Given** the Reference Data System is configured to generate a file-based XML export
- **When** a business user imports the counterparty data
- **Then** the system should successfully import the counterparty data in file-based XML format

03

## Join trade data and counterparty data

- **Given** the trade data and counterparty data have been imported
- **When** the business user initiates the join operation
- **Then** the system should join the trade data and counterparty data

04

## Enrich trade data with counterparty information

- **Given** the trade data and counterparty data have been joined
- **When** the business user initiates the enrichment operation
- **Then** the system should enrich the trade data with counterparty information



05

### Calculate risk for each counterparty

- **Given** the trade data has been enriched with counterparty information
- **When** the business user initiates the calculation of risk
- **Then** the system should calculate the risk for each counterparty

06

### Generate report

- **Given** the risk has been calculated for each counterparty
- **When** the business user initiates the report generation
- **Then** the system should generate a report in Microsoft Excel format containing risk figures for all counterparties

07

### Distribute report

- **Given** the report has been generated
- **When** the business user distributes the report
- **Then** the report should be distributed before the start of the next trading day in Singapore

08

### Configure and maintain external parameters

- **Given** the business user has access to the configuration settings
- **When** the business user initiates the configuration of external parameters
- **Then** the system should allow the business user to configure and maintain the external parameters used by the risk calculations.

**Thank You**