29/10/21

Marks Distribution

		Date	Weight
1	Quiz 1: UML / DB Design Quiz	13-Nov-21	15%
2	Quiz 2: SQL Quiz	27-Nov-21	15%
3	Quiz 3: Overall (other topics)	End-Term	35%
4	Mini Project Demo / Viva		35%

The MINI PROJECT has to be deployed to a server for the demo, running it from an IDE will just result in cutting of marks.

Intro

What are the different phases of the Waterfall Model?
Requirements
Design
Coding
Testing
Deployment
Maintenance

What are the phases of SDLC (Software Development Life Cycle)?
Trick question, it's the same as the one listed above.

Are Waterfall Model & SDLC the same?
NO!
The Waterfall Model is a form of SDLC.
So the different phases of SDLC can be stitched together in different ways, one of these ways is the Waterfall model.

What is the full form of UML?
Unified Modelling Language

What phases of SDLC does UML cater to?
Design!
Is what you would think of.
But in fact UML caters to all phases of SDLC except for Coding.
So for each phase you can find a UML diagram that will be useful for it.

What are the elements/diagrams of UML?
[There are 7 diagrams as follows]

1. Structural Diagrams
    - Class
    - Object
    - Component

2. Behavioural Diagrams
    - Use Case
    - Sequence
    - Activity
    - State

There are more but these are the broad classifications.

What diagrams of UML are useful for Requirements Analysis?
Use Case
Activity
Class
[atleast these three]

What is the most important UML diagram for design?
It depends but Sequence diagrams are pretty important.

Who are the "Three Amigos"?
UML is the combination (unification) of three different styles of diagrams developed by three different people (forgot what those names were, one was Jacobson).

"Gang of Four" is famous for what?
Similar to the "Three Amigos", the "Gang of Four" is the 4 people that came together to develop "Design Patterns".

Types of Software Systems

Enterprise Systems

- Large-scale systems used by **multiple people** collectively belonging to the **same organization**.

- Largeness of a system can be determined by lines of code, number of users, functionalities provided, size of data handled, etc.
- An enterprise system is a large system with a closed user group, for example. people of that organization itself are the users of the system. Because of this on the internet scale, enterprise systems typically have fewer users.
- Example: ERP = Enterprise Resource Platform

Consumer Systems

- Small-scale systems used by multiple **individuals**.
- These systems are typically smaller in comparison to enterprise systems but they have a huge user-base which doesn't even compare to the closed user groups of enterprise systems.
- Example: Mobile Apps

Engineering Systems

- Not a distinct category as such but important in the context of Industry 4.0.
- Large scale systems used by the engineering industry.
- Example: Manufacturing systems, like the operations and service management of telecom providers like Airtel, Jio, etc.

These systems are different from each other in ways that they'll require different software design practices.

One of the distinguishing features between consumer and enterprise systems is how the data is managed.
Consumer systems handle data at an individual user's level whereas enterprise systems collectively handle data as a whole (for an organization presumably).
This is useful because there can be enterprise systems that do not have a closed user group.

You can also think of a "Infrastructure Systems" category that consists of systems that are used to design systems, like Operating Systems.

Software Development Activities

Design

Specifying detailed component interaction is DESIGN.
How to break a large system into small manageable chunks.

Coding

Writing lines of code is PROGRAMMING.
Because of middleware and frameworks, actual lines of code that are written for the software dramatically reduces and automatic code generation is also a facility that's feasible now.

New buzzword that's going around nowadays, NC/LC => No Code/Low Code

This basically involves reassembling existing components and executing them together. No coding involved, just design required that can be executed directly.
Example, Given a class diagram, generate the Class SQL code for it.
So a class diagram is the input that's designed and no coding is involved, that is automatically done by the SQL generation framework.
Automatic testing frameworks where the only things that have to be written are the use-cases and actual testing is done automatically.

Testing
Writing test cases and executing them is Testing.
Non-functional testing => Testing aspects of a system which are not required for its functioning like performance, scalability, security, load, etc.

Regression testing => When a change happens to the system, ensure that it doesn't break the system and it still functions as it should.
So regression testing is linked with a change in the system.

Deployment
Implementing systems that run the software application is DEPLOYMENT.
Making the application available to the end-user.

Multi-Tier Systems

The software system is broken into multiple tiers each of which are their own independent system, so there is no overlap between them.
This breakdown can be done at development as well as deployment level. So multi-tier systems can have two meanings depending on whether tiers are at development or deployment stage.

MVC architecture is an example of this.
Model => Manages the data
View => Interaction with the users
Control => Business logic that connects the Model and View components, i.e. updates the data whenever required and updates the view accordingly when data changes.

This is a development multi-tier system so what does a deployment multi-tier system look like?
It basically means that we deploy parts of a single system onto multiple servers.

What is the benefit of this?
Reliability will be very high.
The physical workload is divided into multiple servers which is why scalability, performance, etc. goes up.

Multi-tier Deployment can only be done if Multi-tier Development has been already done for the system.

Full-Stack Development
The ability to work across all the tiers associated with a software.
Data, Frontend, Backend, OS, etc.
The entire SDLC is required, even the requirements analysis phase is part of this. So people skills involved with communicating with the customer/client and gathering requirements is also involved.

Deployment & Cloud

Data centers host the software system for its deployment.
These data centers can be inside the enterprise/company itself, so they own the servers for deploying their own systems. This is called **On-Prem** deployment.
These data centers can also be in the cloud, where third-party cloud providers will be responsible for providing services related to deployment of a system.

There are three paradigms related to Cloud deployment,

1. IaaS - Infrastructure as a Service
    - Entire server is provided by the cloud provider.
    - IaaS is the same as On-Prem deployment, the only difference is in who owns the servers.
    - Example, Amazon EC2

2. PaaS - Platform as a Service
    - Provides a platform for deployment and development on top of the infrastructure/servers.
    - IaaS + Platform = PaaS
    - Example, Heroku, Amazon Elastic Beanstalk

3. SaaS - Software as a Service
    - Provides an application system for usage/deployment/development on top of the platform on top of the infrastructure/servers.
    - PaaS + Application = SaaS
    - Example, Gmail, Google Drive, etc.
    - SaaS has to be designed such that the same software can be used by multiple enterprises end-to-end.
    - The problem with this big advantage is that if some business still wants to implement some custom business logic on top of it then they cannot do it.
    - For this reason, separate provisions for external APIs are made available that can be used by the respective enterprises to build on top of the existing software.
    - Salesforce Developers are an example of this.
    - This will take the application to more of the PaaS side rather than the SaaS side.

Agile Development

- An iterative model for development of software.
- You go through the entire SDLC stages in phases.
- At the end of each phase a working product is obtained after which the next phase starts in which the working product goes through more refinement to output an even more refined working product and this continues till the customer/client is satisfied.
- In a regular iterative model, deployment is not considered in the iteration phases, i.e. when and how many times deployment happens is not of concern to the model. So getting a working product as output of a phase is not a requirement in just a plain iterative model.
- However in the Agile model, deployment is also part of each phase.
- So the customer is able to see the development output in real time as a functioning system.

Maintenance however is done as the product is deployed, i.e. in the actual use-case itself.
When a phase is ended with the deployment of the product, maintenance phase starts as users and bug-testers can report issues with the new product and maintenance can happen parallelly as the next phase of development is also ongoing.

If bugs belonging to a previous phase are discovered at a much later date then you will have to go back and retrofit the bug fix into all phases, which could involve refactoring the entire application.
Note: Refactoring is a nice euphemistic way of saying to straight up rework the application.

And this refactoring has to be such that it doesn't break the existing application because the product is in live deployment already.
This is why Agile Development is so difficult and why multi-tier systems are so beneficial as fixing any problems can be contained to only one particular tier instead of the entire system if the tiers are designed with as minimal interlinking as possible.

Continuous Integration/Continuous Deployment (CI/CD)

Taking Agile development to the next level.
Agile brought the phase duration down from months to the span of weeks and this takes it even lower.
No longer working on phases of SDLC and continuous integration of functionalities are done as well designing, bug-fixing, testing, etc.
So basically all the phases are happening at the same time.
This is indeed very chaotic.

The CI/CD pipeline,

Code -> Test -> Integrate -> Deployment

This happens continuously, very continuously, almost on a daily basis.

For iterative development, project management methodologies and consistency are more than enough.
With Agile, all of the above + a good versioning system should be enough.

However with CI/CD, you can imagine how severe the testing chaos will be as regression testing will have to be done on a daily basis as well and it can be very easy to slip up.

Because of how difficult this is, a lot of tooling and automation has emerged and implemented which forms the new field of **Development Operations (DevOps)**.

It includes,
Versioning System (Git)
Automated Testing
Containers
Management Dashboards

02/11/21

Enterprise Softwares (Contd.)

What is it for, i.e. purpose of Enterprise software?
To automate a business's task or process
Example, for an educational institute, ERP is required as an enterprise software and it will need to automate tasks like course registrations, fee management, hostel management, canteen, etc. etc.
Requires lot of requirement analysis (used to have job title of System Analyst)

Who is it for, i.e. who is the user base?
Another way to frame this is, who are the stakeholders of the enterprise software?

1. Users
   Actual users of the software, e.g, clerks for booking systems, etc.
2. Organizations
   Sponsors => People that fund the development of the software.
3. External Organizations
   Compliance organizations, e.g. SEBI is a compliance organization for investment softwares/platforms, IRDA for insurance, etc.

Note: What is it for is heavily dependent on Who is it for and vice-versa

Where is it used?
Whether it is used inside the organization or outside the organization.
This involves geographic location, devices supported, etc.
(platform doesn't factor here because that's a technical aspect and these questions are a very high level overview of the system as a whole and doesn't focus on the technical details).

How is it used?
Real-time processing or batch processing.
Can also be phrased as, interactive or non-interactive.
That is, whether responses are expected in a given time or not.
Whether it is used by technical users or non-technical users.

When is it used?
Is it a 9 to 5 system, is it a 24x7 system, etc.
Any other combination of time for which system has to be up for.

An example of System Design ideology,
Say you have a system that has to be active 24x7 and another system that will only be active from 9 to 5 but across continents. So in each continent it will be active from 9 to 5 which will mean that the entire system has to be on, 24x7.
But… What are some system design aspects that will be different for both systems?

You can split the user load across servers which do not need to be on all the time depending on which region the server is located in.
You can do backups or maintenance in this system when the server is off.

Enterprise Software Deliverables

1. Requirements ("What", what functionalities does it provide)
    a. Functional
    b. Non-Functional
2. Architecture and Design ("How", how is it going to be implemented and designed)
    a. **Architecture = Collection of guiding principles that drives the design**
    Example, you want to build a room where light must fall on a particular area. This is a guiding principle that changes the design such that windows must be placed in the appropriate area for the light to enter.
    b. Upstream activities that determine other downstream activities, architecture is an upstream activity that determines the downstream activities like design, coding, etc.

Sidenote: Types of Activities

Downstream => Activities that will have to be done after the current activity
Upstream => Activities that precede the current activity, already completed activities.

Development Process

Consists of the following components,

Environments

dev
test
prod

The test environment is much closer to the prod environment than the dev environment. The quantity will just be scaled down for the test environment.
Testing is done to ensure that there are no issues in production which is why the test environment will try to resemble the prod environment as much as possible ofcourse with additional tools for testing and such.

What should be common across the three environments?

- Operating System (this is debatable according to sir himself but his principles are such that all three environments must be running on the same OS no matter how platform independent the frameworks, libraries and systems are).
- Database Management System (DBMS) [Not the database itself]
- Libraries (can be split but there will be a lot of common libraries).

What should be different across the three environments?
Tools
This is a huge category and there are a lot of things that will be different between the three environments.
You'll have access to IDEs, debuggers, GUI applications in the dev environment whereas prod environment is most often restricted to just command line usage along with the system.

Types of testing (broadly)

1. Unit - Testing a particular component/unit of the system.

2. Integration - Integrating external systems into the system and testing whether the integration is successful or not.
   Example, GST is a compliance organization for the system and we have to upload GST reports at EOD, so we have to integrate the GST APIs with our system and testing how well this integration works is part of integration testing.

3. System Testing - Testing the entire system as a whole from within the system.
   Note that this also includes, integrating two or more units in an incremental manner until the entire system is covered (I know that this is what you thought was integration testing :P).
4. UAT (User Acceptance Testing)
   Testing done by end users (the company that gave the contract, the users that the system is for, etc.).
   Can happen that the development organization has no hand in performing UAT.
   Not as exhaustive as the other testing categories.

There are loads more, but they can be put under these 4 formal categories or a variant of them.

Configuration Management

SCM = Software Configuration Management
Involves version control and its management.
The entire dev team will be working on a shared repository which is managed using version control management software like Git.

The development cycle consists of,
Coding
Testing
Check-in
Check-out
And back to Coding again.

Check-in => Putting the code from the local repository into the version control repository.
Check-out => Pulling the code from the version control repository into the local repository.

Because we have CI/CD pipelines, i.e. dev-ops, this whole cycle has to be continuous even after deployment/release.

Quality Management

1. Process Quality => Are we building the product right?
2. Product Quality => Are we building the right product?

Deployment

Consists of three elements,
1. Compute
    a. Node
    b. CPU & RAM
    c. System Software

2. Storage
    a. DAS - Direct Attached Storage, e.g., HDD, RAM, etc.
        i. Advantages: Speed, no network connectivity required, standalone storage.
        ii. Disadvantages: No portability (cannot move the server around), scalability (limit posed by the server's physical storage capacity).
    b. SAN - Storage Area Networks
        i. Storage is completely outside the compute server.
        ii. Each SAN has a finite amount of storage but these SANs can be stacked on top of each other for virtually infinite storage for the compute server.
        iii. So the storage and compute servers are on different networks, so throughput is limited by the cables that are being used for communication among various other factors.
        iv. Typically the communication medium is fiber so that communication is as fast as possible.
    c. NAS - Network Attached Storage
        i. Also on the network but uses ethernet, so the local network is involved. It can be connected on a separate machine but it cannot be so far that it goes outside of a building.
        ii. Imagine it this way, you connect a NAS device to your router and now all devices connected to the WiFi can use the data stored in the NAS.
        iii. Limited by throughput as well as capacity because it is on the same local network.
3. Network
    a. Routers and Switches
    b. Firewall
    c. Load Balancer

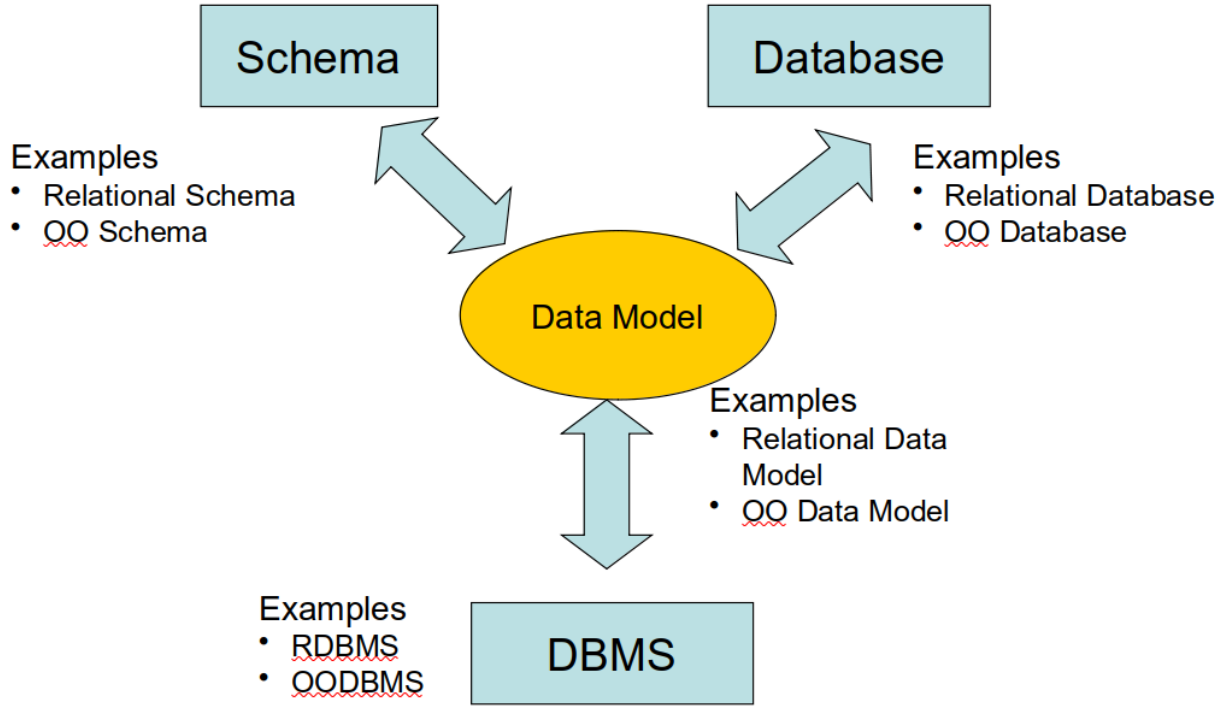
"There is more to enterprise software development than just coding or developer's role" is the point of emphasis here.

09/11/21

Introduction to Database Design

Data Model
It represents the coupling between the schema, database and DBMS.

The database design process can be illustrated with the following image,



The database stores the actual data but it is the schema that determines in exactly what format the data is to be stored and it is the DBMS that actually manages the databases so that handling the data becomes easy for the user.

So the Database Design process gives us the data model which is the binding factor for all three aspects shown in the image.

What this tells us is that there is no general purpose DBMS or Database or Schema, which type of these components are used depends on which Data Model that we are using.
That is, if we are using the Relational Data Model then it implies that we are using the Relational Schema, Relational Databases and also RDBMS.
Same goes for Object Oriented Data Model.
This is why the Data Model is said to be the binding factor to these three components.
[Quite often, Schema and Model are used interchangeably but that is not the case, Model determines the Schema]

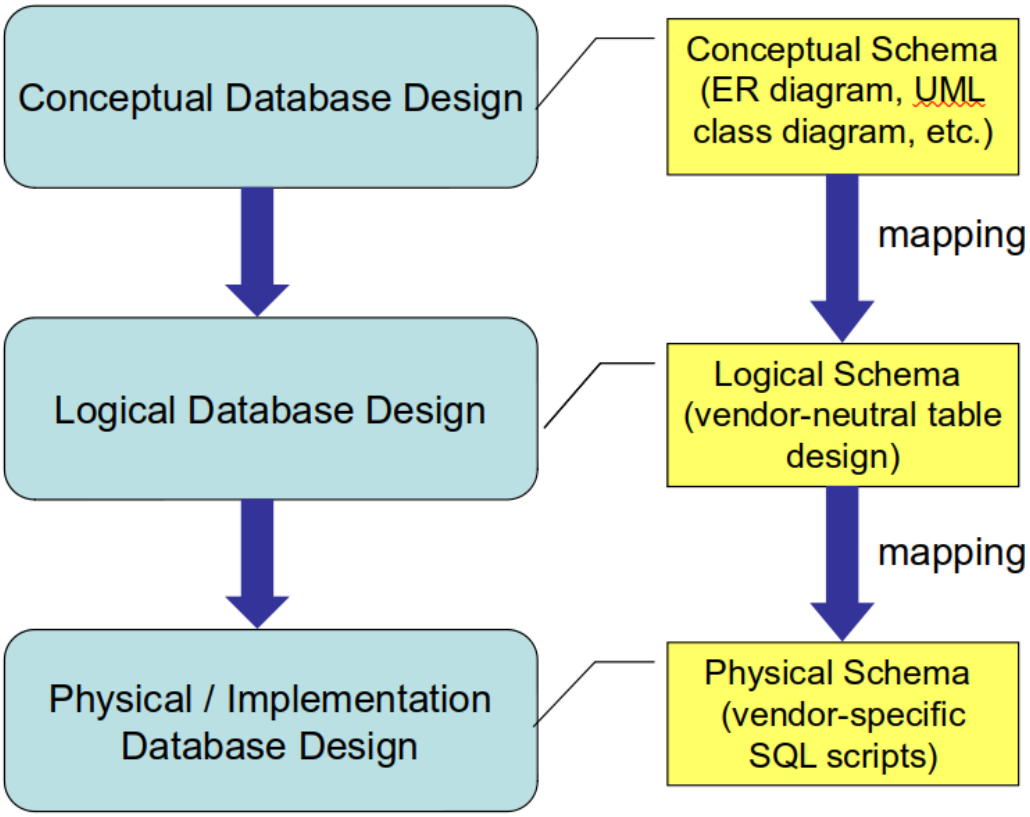
Relational Database Design

Goal is to create a design that can be implemented using an RDBMS.
The design is a collection of tables (which is a collection of rows and columns).
There are two ways in which the set of tables for a database can be identified,

- Design through conceptual modelling
- Design through normalization
  [Note: Sir said that nobody uses this in industry, so that's nice.

The reason why it's not used is because from a practical point-of-view it is much more complicated to implement than conceptual modelling but normalization is no doubt a useful method of database design.]

Design through Conceptual Modelling



Conceptual Data Model involves pictorial representations and/or notations to describe the schema of the database.
But the key point is that we don't start the design directly on a sheet of paper or anything like that, first we iron all the phases out conceptually and only then move onto implementation.

Once we have the Conceptual Schema ready, we can map it to the Logical Schema via a process called OR Mapping and then proceed to map to the Physical Schema which actually generates vendor-specific SQL scripts.

[Vendor just refers to the DBMS provider like Oracle, MySQL, PostgreSQL, etc.]

We're going to be using UML Class Diagram as our Conceptual Data Model as it's more powerful than ER Diagram.

Why do we need Conceptual Design?

- Relational Database Designs can be very hard to communicate (just pure Databases, if you talk about things like ER diagrams then you are talking wrt Conceptual Design only).
- Errors can be hard to detect.
- Conceptual designs are usually expressed in pictures (and pictures are worth a hundred words here).

Are there any errors in this design?

Diagram 1:
[These can be .sql files or .c files or anything really, .java is just to illustrate that its a code file]
There are no syntax errors in these files but there is a logical error somewhere.

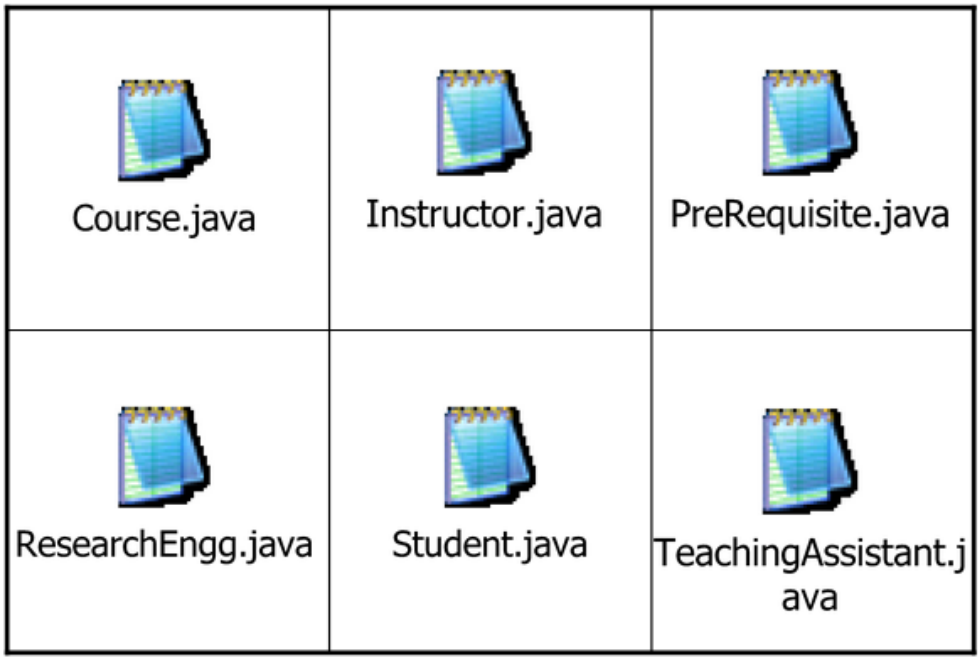
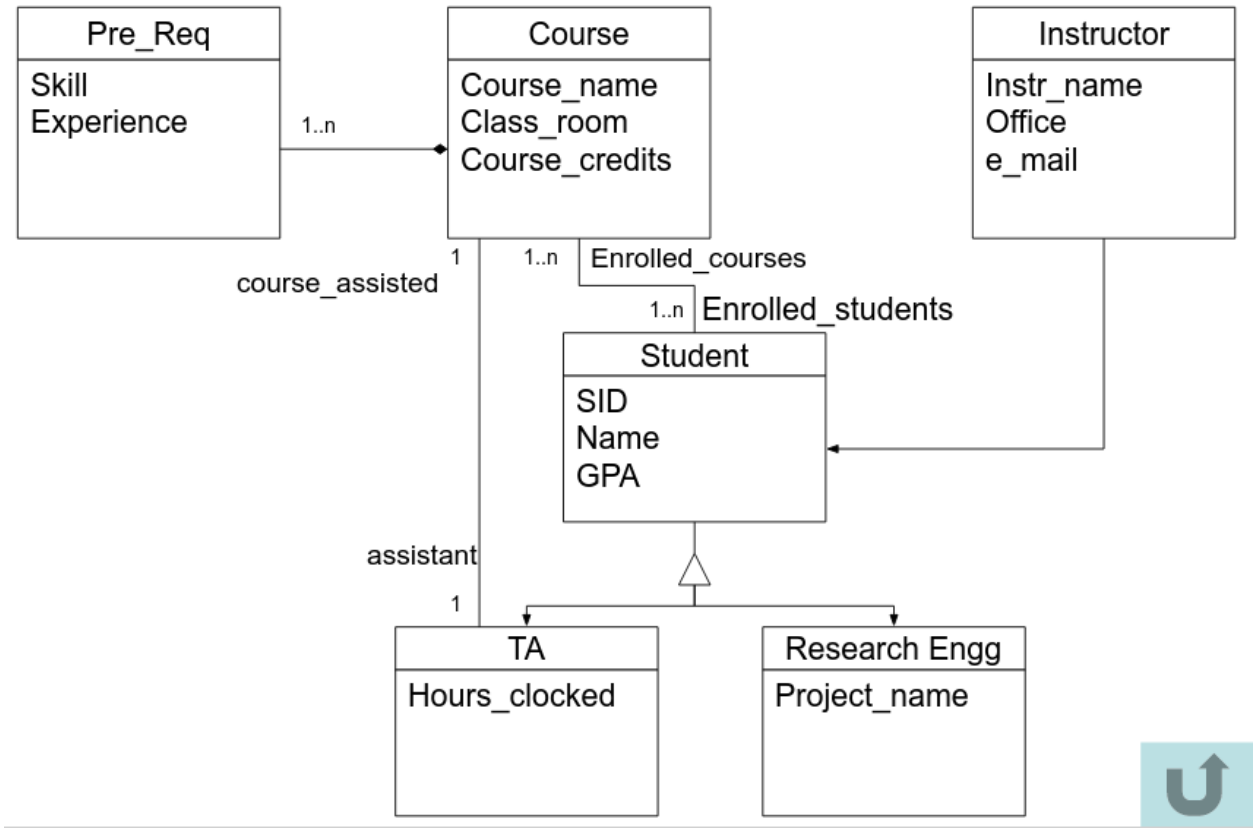
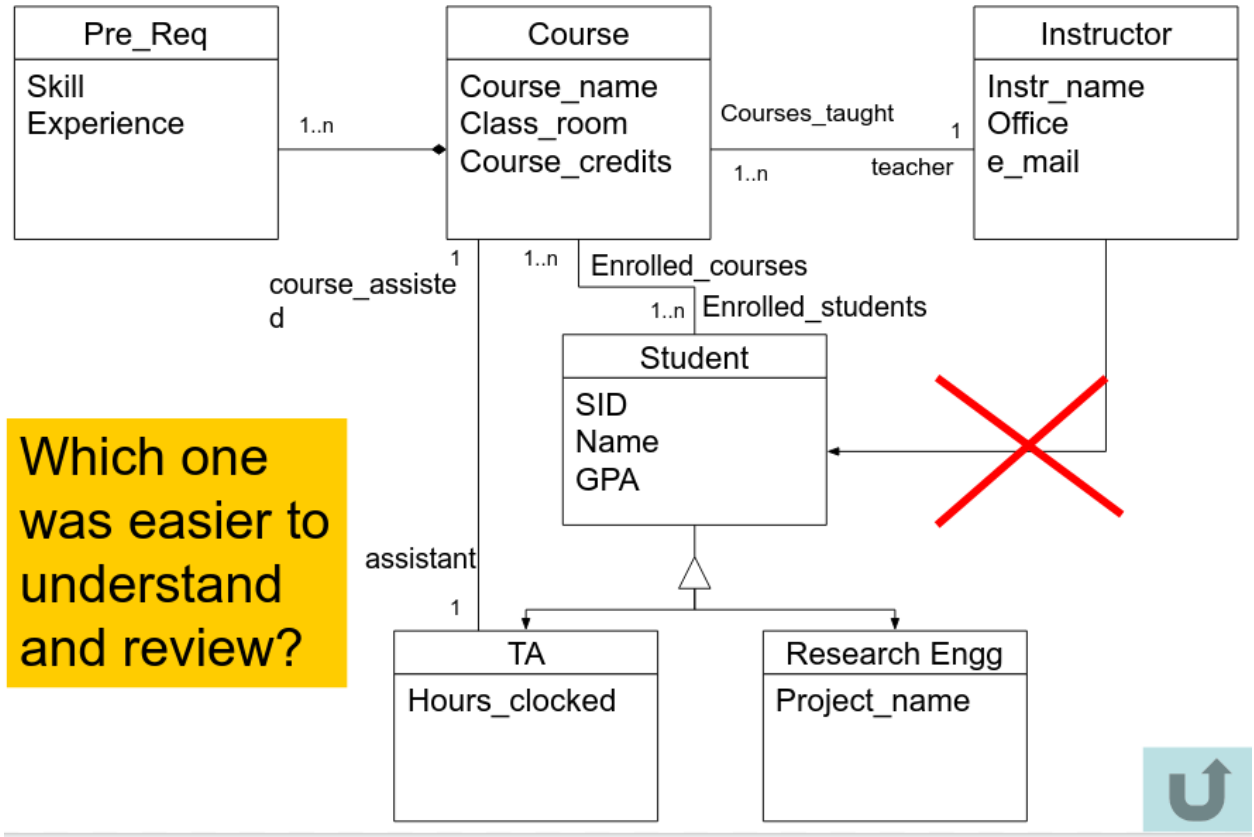


Diagram 2:
Now what is the logical error in the given UML class diagram representing the same schema as the .java files above?



Instructor is someone that teaches a course. So it makes no sense that an instructor is linked to a student directly (plus it's not even defined what kind of relation the Student & Instructor classes have).
Instead an instructor should be associated with the courses that they teach and students can thus be related to an instructor indirectly via the courses that they have selected.

This image shows what changes have to be made,



Conceptual Design Summary

- It uses diagrams to communicate information about data and relationships.
- Helps make “implicit” information “explicit” [as we saw in the diagram above].
- Recall that data models are vocabularies, i.e. the type of data model defines what kind of rules that the schema, database and DBMS have to follow.
- Conceptual data models help create conceptual database designs.
- ER and UML are examples of conceptual data models.

Relational Data Model Revision

Each relation is a table.
Each attribute of a relation is a column.
Business rules/constraints which are given by the client are mapped into Database constraints. **But remember that not all Business constraints can be translated into Database constraints.** In such cases, the constraints are implemented via the application itself before being given to the database.

Why do we prefer to implement constraints at the Database level (if it’s possible) instead of the Application level (because we can implement any constraint at the Application level)?

- The most important reason is that business rules are data constraints, so it is better to implement them at the database level as it means that we can even use the same database for multiple applications without having to add constraint checking for each application separately which would be redundant/duplicated effort.
- And another supplementary reason is that the actual effort required to implement constraints at the database level (if it is possible) is way simpler (one-liner even) as compared to implementing the same constraint at the application level.

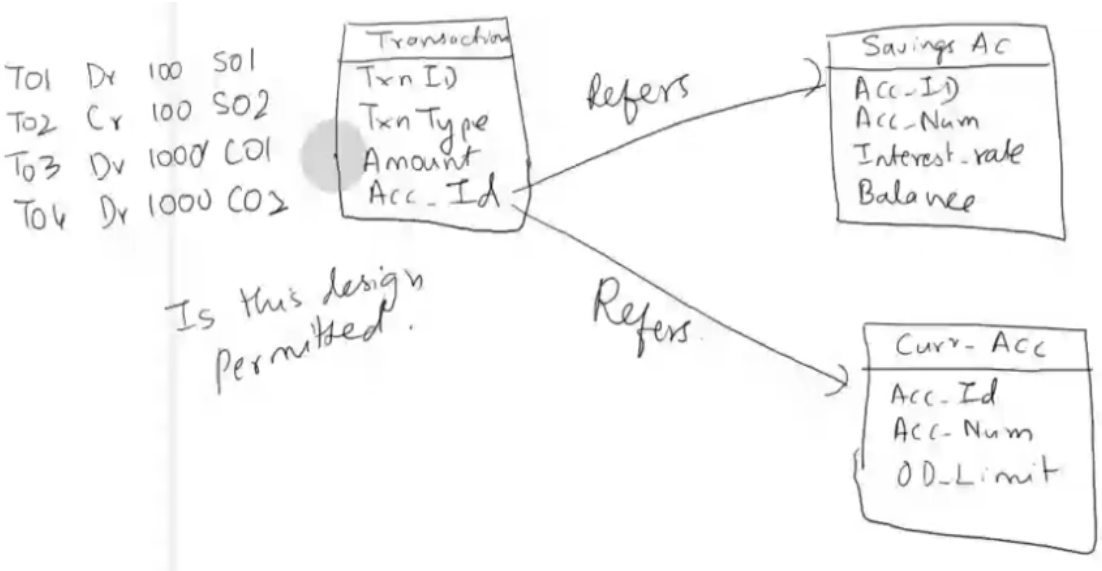
If we implement constraints at the database level then do we not implement those same constraints at the application level?

- Answer is that it depends but it is quite often implemented at both levels. Popular example is form data validation at the front-end as well as the database.
- We implement validation at the front-end to make the software more usable and faster and we implement validation at the back-end to protect our database and prevent suspicious/malicious activities of users.
- This adds duplication of code to the application but the trade-off is that we're making the overall application far more usable for the end-user which is highly beneficial.

Database Constraints include,

- Entity Integrity/Key Constraint
    - Set of attributes that uniquely identifies each row of the table.
    - Every table must have atleast one key.
    - Primary key cannot contain NULL values.
    - Types:
        - Superkey => Set of attributes that uniquely identifies each row of the table.
        - Key / Candidate Key => Minimal, irreducible superkey.
        - Simple => Key that consists of a single attribute
        - Composite => Key that consists of multiple attributes
            - Is a composite key minimal?
              YES!
              Because by definition if you remove any attribute from a composite key then it would no longer be a key. All composite keys are candidate keys and if a superkey is minimal then it would be a candidate key as well as composite key.
        - **Surrogate Key / Artificial Key** => A key that is **generated by the system** and is **immutable**, i.e. can never be changed throughout its lifetime.
            - A key that is not an artificial key is called a Natural Key.
            - Examples:
                - Our college roll number is a surrogate key because it is generated by the college system and it never changes.
                - A SIM card's mobile number is also a surrogate key.

        - Primary Key => Arbitrarily chosen candidate key that will be used for identifying each row in the table in SQL statements by the DBMS. This ensures that tables are properly maintained and any conflicts can be directly detected.
        - Unique Key => A key that can take 1 NULL value for 1 row but other than that it has to uniquely identify each row in the table. There can be any number of unique keys in a table but only one primary key.
- Referential Entity Integrity Constraint
    - Foreign Key
        - Is it a key in the table? It depends. It can be a key or not a key (detailed example after this section).
        - However, the attribute that the foreign key is referring to has to be a key of the referenced table (by SQL standards it has to be the primary or unique key).
        - The referential integrity constraint **refers** to the fact that the foreign key actually refers to a column in another table (referenced table) and can only take values that are actually taken by this column in the referenced table.
        - Example: Employee is in a department, so each employee in the Employee table has an associated “d_id” which refers to “d_id” in the Department table. However Employee(“d_id”) cannot take a value which is not present in Department(“d_id”), i.e. an employee cannot be mapped to a non-existent department and that is the referential entity integrity constraint.
        - There can be any number of foreign keys in a table.
        - Consider a banking system and a table called Transaction with this schema,



        - Basically we are asking, can we have a foreign key that is declared like this in its definition?

        - acc_id INT REFERENCES Savings_AC(acc_id), Curr_AC(acc_id)
        - NO!!!
        - No reason other than the fact that a foreign key can reference only one column in one table.
        - A column can be referenced as a foreign key by multiple tables but vice versa is no no.
        - Similarly the following statement is also not allowed,
            - acc_id INT REFERENCES Savings_AC(acc_id, amount)
- Semantic Constraints
    - Anything that's not Entity Integrity or Referential Entity Integrity constraints fall under the category of Semantic Constraints and typically includes business logic. These constraints are not standardized as such and what type of semantic constraints are available depends heavily on the DBMS provider.
    - Things like **data type enforcement for column values**, value constraints such as ensuring values for a particular column falls in a particular range. which can be implemented via **CHECK constraint** or belongs to a certain set of values which can be implemented using **the ENUM constraint**.
    - Also includes the **Unique Key, NOT NULL** (whether field is mandatory or not) and **Default** (field has default value) constraints as well.

Can foreign key be a key in its own table?

- YES!
- A straightforward example for this is whenever the concept of inheritance is used (will learn more about this in the UML class diagram modelling section later).
- So for example, in a banking system, there is an Account table and there are separate tables for different types of Accounts like CurrentAccount, SavingsAccount, etc.
- Each of these account type tables inherit the Account table and any account has an account_id irrespective of their type of account, so each account type table has a foreign key that refers to the account_id of Account table.
- And this foreign key is naturally the key of the Account type tables as well.
- So you can have foreign keys that are primary keys in their own tables.
- Just remember that the direct example for this is inheritance.

Surrogate Key vs Natural Key

A case study on why Surrogate keys should be chosen as Primary Keys over Natural Keys,
Consider a Telecom provider's Database where mobile numbers are selected as the Primary Key.
This is a natural key because the provider themselves assigns the mobile numbers.
So this mobile number is used as a foreign key in several tables (potentially hundreds).

Earlier there used to be 9 digit mobile numbers but eventually this was updated to 10 digit to accommodate increased demand for mobile phones and SIMs.

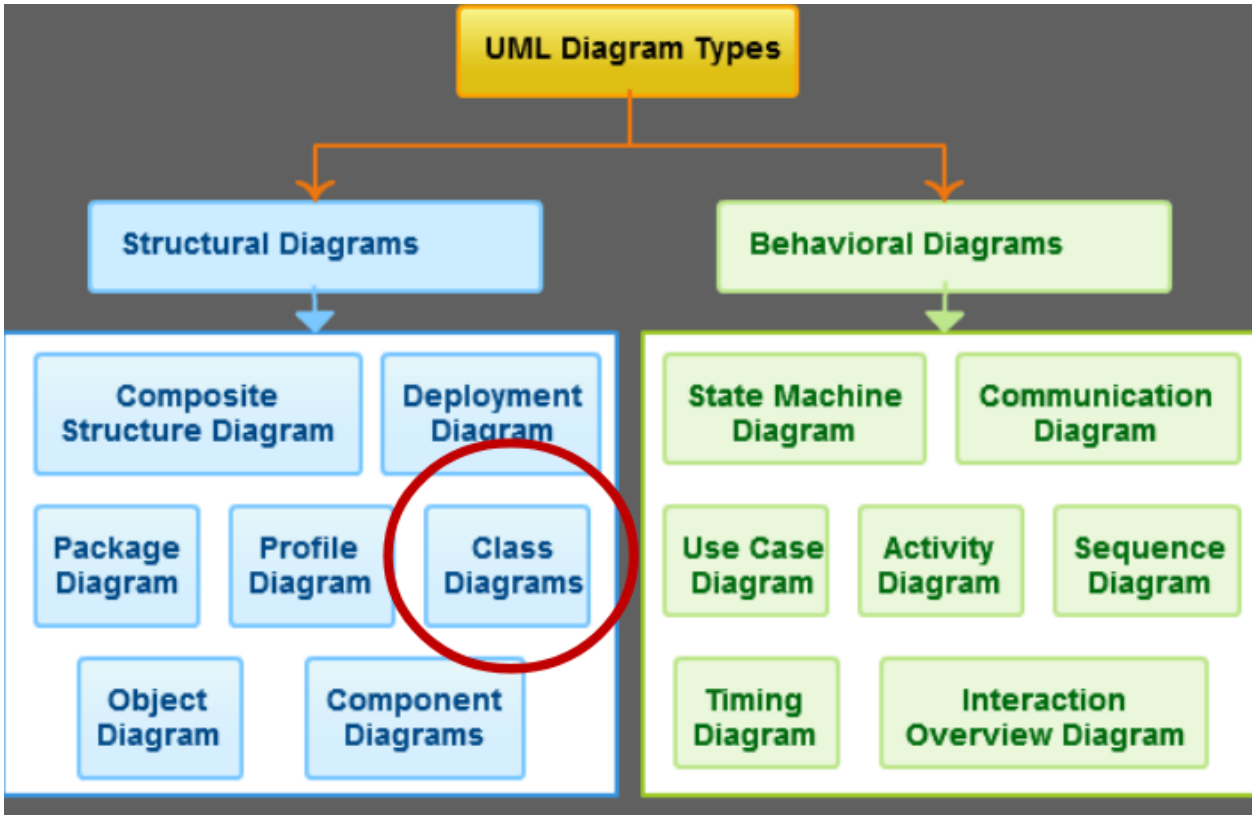
Now you can't update these 9 digit mobile numbers directly because it results in referential integrity violations, so all of the referenced tables must be analyzed and updated properly (you can see why this is so tricky) before finally the mobile numbers are updated in the table that is being used for reference.

The lesson to learn from this is that Natural Keys are subject to change.
This is exactly why you should prefer Surrogate Keys over Natural Keys as your primary key as a problem like this would have never occurred in this case.

You could have used a surrogate primary key like customerID that is system generated and now you can directly update mobile numbers in all the tables without violating any constraint or you can just avoid using mobile numbers in any tables and this problem would be resolved in like 5 minutes for even millions of rows [mobileNumber would now be a Unique Key + Not NULL column].

Note that this would add extra requirements of space for storing one extra column and all operations involving mobileNumber would now require join operations which can be time consuming.
So there is a tradeoff between space and flexibility.
But we consider that the flexibility the surrogate key approach provides is so invaluable that we shouldn't be stingy about storage.

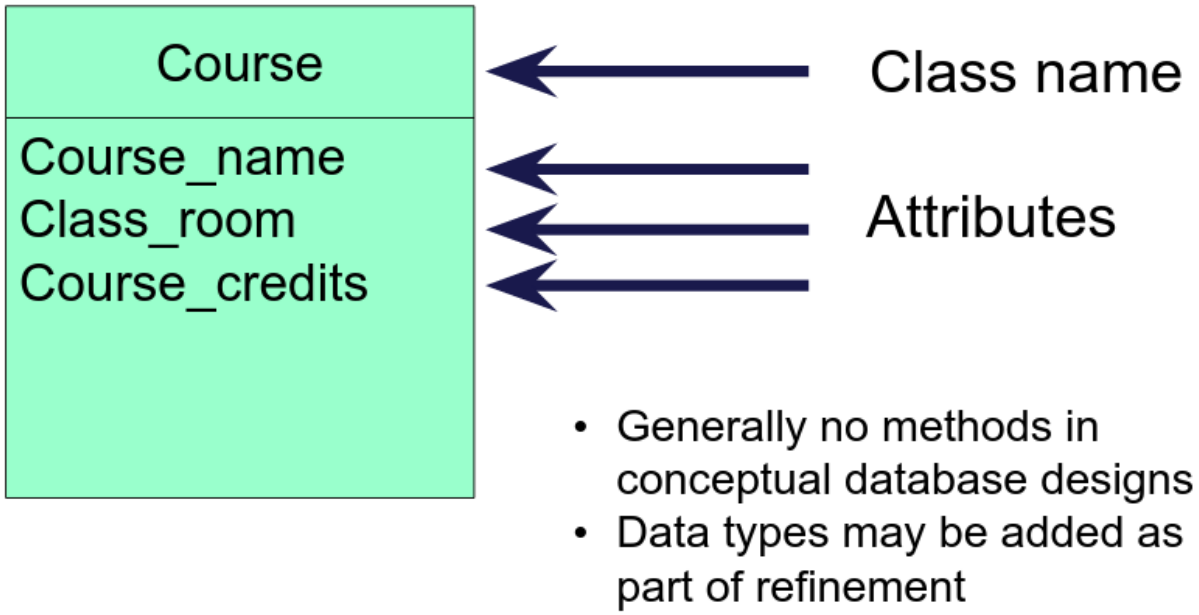
UML Class Diagrams for Database Design



Elements of UML Class Diagram

1. Class
2. Attribute
3. Relationship
    a. Association
    b. Aggregation
    c. Composition
    d. Inheritance

General representation of a class in a Class Diagram



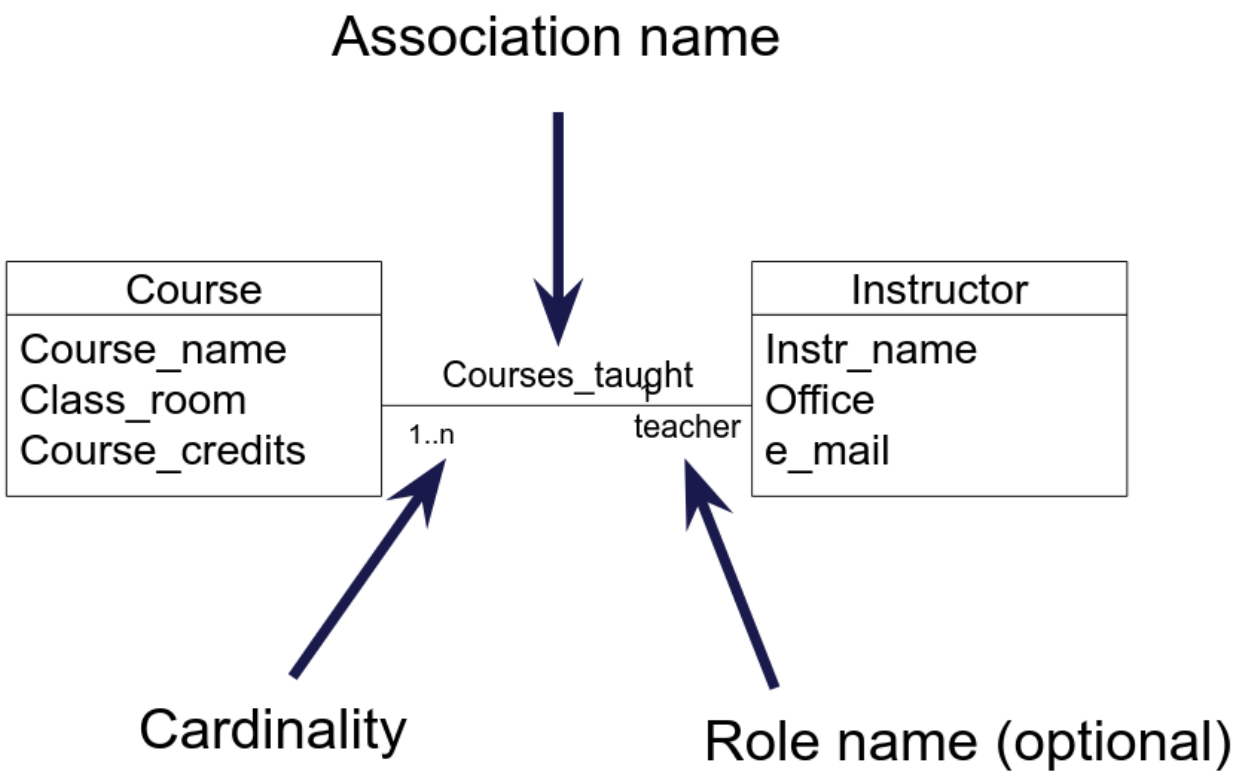
Types of Relationships

1. Association

Association is the semantic relationship between classes that shows how one instance is connected or merged with others in a system.
Basically, it tells you that two classes are related to each other.

Sidenote: Association attributes
Association itself can have attributes like how classes can have attributes.
If an association has any attribute it means that those attributes are relevant only in the context of the relationship between the two classes.
What this means is that such attributes can only be defined properly when you have 1 instance of both classes participating in the association.

Example: Account and Customer have M:N association, in this case, their association can have an attribute AccountCustomerLinkDate which would store the date when the customer got associated with this account. This definition of date requires us to specify both account_id as well as customer_id which means that it should be modelled as an association attribute.
[This is very similar to what we have already learnt in ER Modelling]



This can be described in text as,
Instructors are associated with a course in the role of “teacher” via the “Courses_taught” association.
If cardinality is not written it is assumed to be 1...1

Note: 1...n in the above example means that, n is on the side of Course and 1 is on the side of Instructor (it is the opposite of what’s written basically)

Interpretation of cardinality
Use the two sentence rule.
First sentence to describe how class on the left side is related to class on the right side.
Vice-versa for the second sentence.

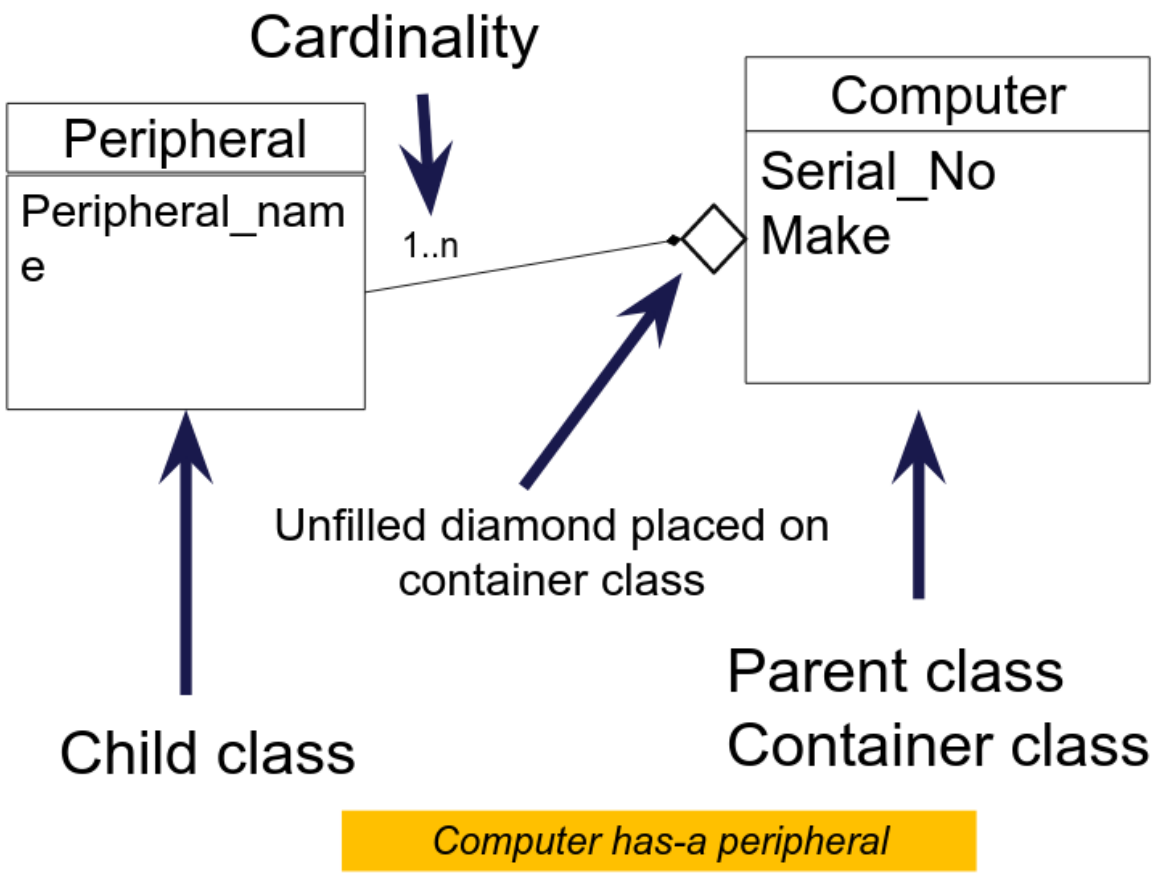
For example, in the above class diagram, cardinality => 1...n
One Course is taught by one Instructor.
One Instructor teaches many Courses.

So fix one class and write how to relate it with the other.
Sentence 1, Fix 1 Course, how many Instructors is 1 Course related to? 1 Instructor
Sentence 2, Fix 1 Instructor, how many Courses are 1 Instructor related to? N Courses

2. Aggregation (HAS-A relationship)

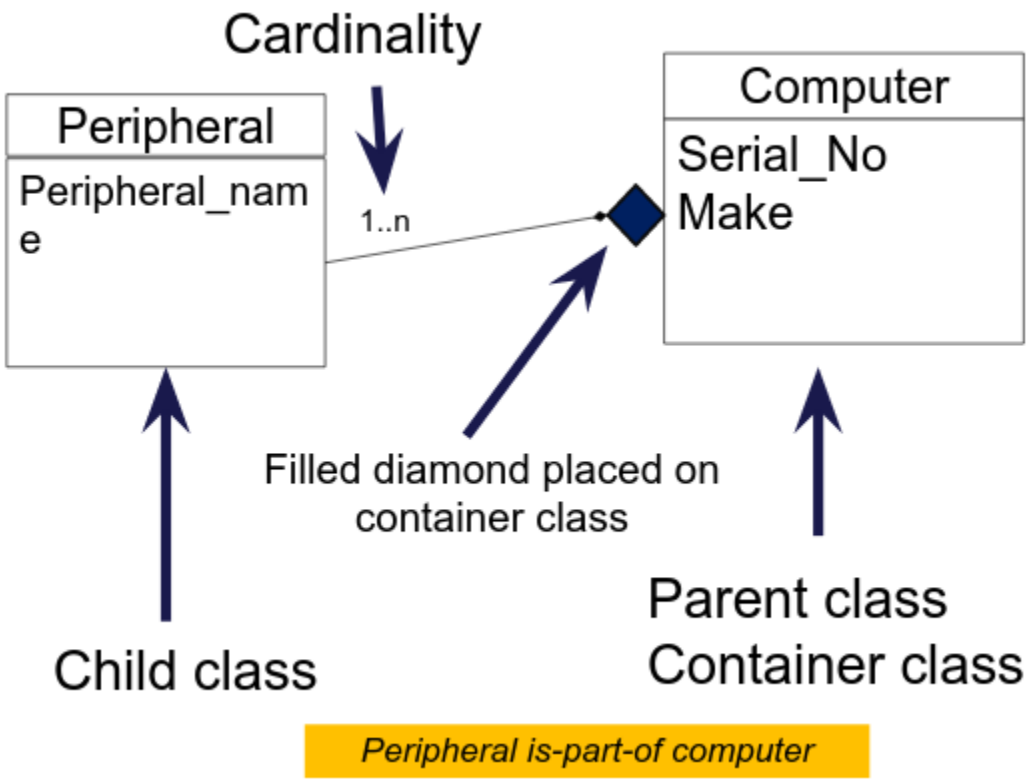
An aggregation is a special type of association in which objects are assembled or configured together to create a more complex object.
It is a relationship which shows a class (child) as part of another class (parent).
An aggregation is very similar to a 1...n association from parent to child.

The below example should help specify everything,



- The association name is fixed to “HAS-A”.
- The additional implication that an Aggregation relationship has is that the relation can never be n...m, so in this example, one peripheral can never be part of more than one computer. So an Aggregation will always have 1 on the side of the parent class in cardinality ratio.
- **Child class can exist independently until attached to a parent (the meaning of this would become more clear in the OR Mapping phase).**

3. Composition (PART-OF relationship)



The relationship name is fixed to “IS-PART-OF”.
Similar to aggregation, its cardinality cannot be many:many.
Child cannot exist independently until attached to a parent (“existence dependency”).
Deep Deletion semantics: All child objects get deleted when the parent is deleted**.**

Aggregation vs Composition: Summary

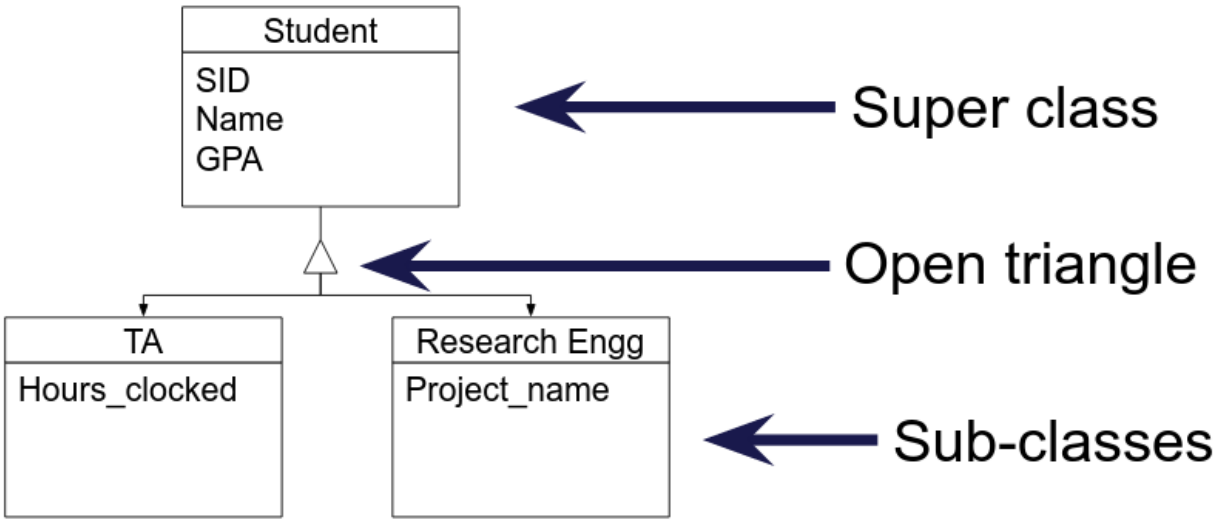
Aggregation and **Composition** are subsets of association meaning they are **specific cases of association**. In both aggregation and composition, the object of one class "owns" the object of another class. But there is a subtle difference:

- **Aggregation** implies a relationship where the child can exist independently of the parent. Example: Class (parent) and Student (child). Delete the Class and the Students still exist.
- **Composition** implies a relationship where the child cannot exist independent of the parent. Example: House (parent) and Room (child). Rooms don't exist separate from a House.

4. Inheritance (IS-A Relationship)

We are talking about inheritance of attributes here and not behaviour because behaviour is modelled using methods and those are not part of the conceptual database design.

There are two interpretations of the Inheritance relationship depending on the notation used.
First one is non-partitioned subclasses,



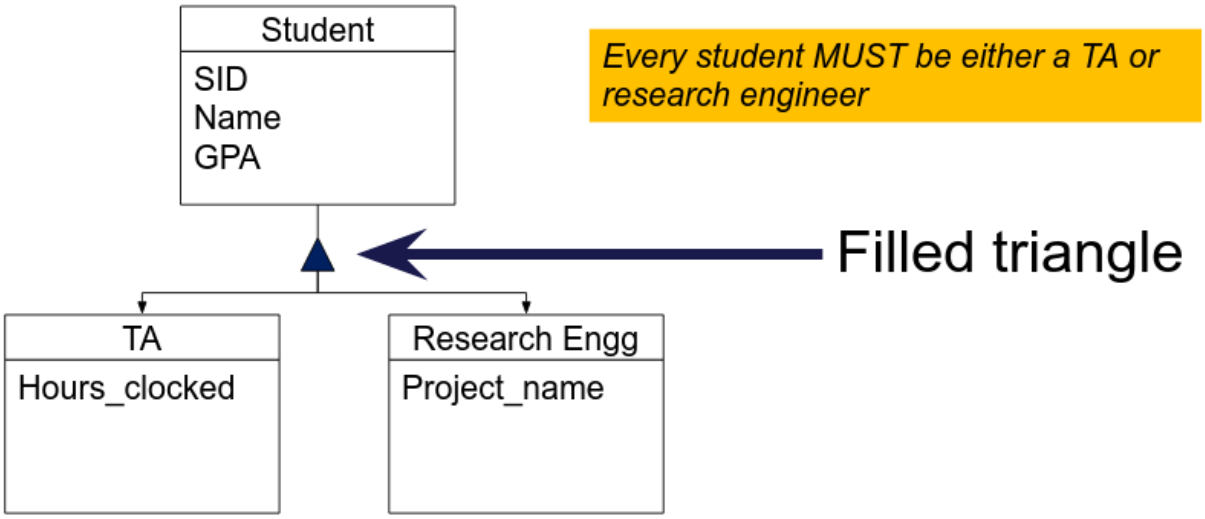
A student can be either a TA or research engineer or neither

The open triangle signifies that the super class is inherited by **non-partitioned sub-classes.**
Which is why it is possible that a student can neither be a TA nor a research engineer.
So Student class can be stand-alone, i.e. **Parent objects can be stand-alone.**

Some properties of Inheritance relationship,

- The relationship name is fixed to “IS-A”.
- Child object inherits all attributes of the parent objects.
- Class hierarchy can be arbitrarily deep.
- Cardinality:
  Cannot be “one:many” => Many students cannot be one TA (one TA is just one student)
  Cannot be “many:many” => Many students cannot be many TAs.
  Cardinality can only be 1:1 or 1:0

And this is the other interpretation of the inheritance relationship which is partitioned subclasses,



So the parent class must also be part of a subclass and **a parent object cannot exist stand-alone.**

It's called partitioned subclasses because the union of instances of all subclasses would give the superclass instances and all the subclasses are mutually exclusive.

In OOPS, the parent of partitioned subclasses can be mapped to abstract classes as an abstract class also cannot exist on its own and must be inherited by a subclass to instantiate an object.

12/11/21

OR Mapping (Object-Relational Mapping)

We were doing the process of conceptual data modelling in which the first step was the conceptual schema design for which we used the UML Class Diagram.
UML diagrams fall under the object oriented schema design paradigm.

Now the next step is the logical schema design which is vendor-neutral table design.
Our end goal is to design tables that can be handled by an RDBMS but our conceptual schema is in Object-Oriented paradigm, so we have to convert this class diagram schema to a relational model schema and this process is called OR Mapping.

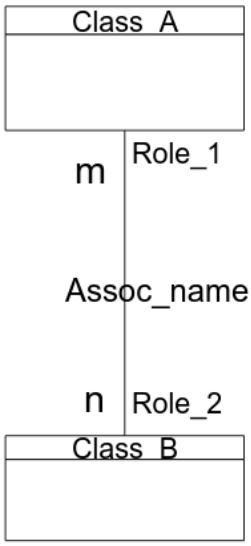
OR Mapping Rules

Input is the Class Diagram that was designed in Conceptual Schema design and Output is the Relational Schema design (Logical Schema).

1. Class

- Create a table for each class in the diagram.
- Include all attributes of the class as columns of the corresponding table.
- Add a surrogate key to the table (typically name for the key follows the syntax, TableName_id)
- Add semantic constraints to the table as per requirement, i.e. unique key, NOT NULL, value constraints, etc.

2. 1 : N Association



- Let the table on “1” side of association be Table_A and on “N” side of association be Table_B.
- Add the surrogate key of Table_A as foreign key on Table_B [we can add it to Table_B as well but then it would be a multivalued attribute which is not allowed in relational schema].
- The name of the foreign key should be the name of the role.
- Add a NOT NULL constraint to the foreign key [because relation is 1:N, if it was 0:N, then NULL can also be allowed].

3. N : 1 Association

Same as above but in the opposite direction.

4. M : N Association

- Create a new table Table_C having the same name as the association name.
- Add a surrogate key to Table_C.
- Add any association attributes that may be present (attributes of the relation itself) as columns of Table_C.
- Add the surrogate keys of Table_A and Table_B as a composite key of Table_C.

5. 0...1 : 1 Association

- Let Table_A be on the “0...1” side and Table_B be on the “1” side of association.
- Add the surrogate key of Table_A as foreign key in Table_B.
- The name of foreign key should be the name of the role.
- Allow NULL values to be the foreign key.

[Similar rules as that of N:1 for N = 1]

6. 1 : 0...1

- Same as above but in the opposite direction.

7. 1:1 Association

- We cannot map a 1:1 association to the relational model!!!
- We have to convert it to a 0...1:1 or a 1:0...1 association.

Why is this so?
Consider a strict 1:1 Association example like, 1 person must have 1 Aadhaar card.
So there are two classes Person & Aadhaar that are linked via 1:1 association.
Let person_id and aadhaar_id be the surrogate keys.
Now both classes have to be linked to each other because it is 1:1, i.e. given an aadhaar_id we should be able to know the person_id and given a person_id we should be able to know the aadhaar_id.
So we add surrogate keys of each table to the other's table as a foreign key and because it is 1:1, a NOT NULL constraint has to be added on the two foreign keys in each table.

Now the question becomes, how can we insert into either of the tables?
If we try to insert an entry into the Person table then we must know that person's aadhaar_id otherwise the NOT NULL constraint will be violated as there's no entries in the Aadhaar table yet.
If we try to insert an entry into the Aadhaar table then we must know the person_id of that Aadhaar card's holder, otherwise again the NOT NULL constraint will be violated.
Basically we're stuck in a deadlock.

8. Inheritance

- Let Table_ B be inherited by Table_A => Table_B subclass, Table_A superclass.
- Make the surrogate key of Table_B as the primary key for the table and also a foreign key that references Table_A's surrogate key
  (remember the Account -> Current Account, Savings Account, Deposit Account inheritance example, "account_id" doesn't take into consideration the account type and so all the Account type tables will have foreign key that refers to "account_id" of Account table).
- Because the surrogate key of Table_A cannot be NULL it ensures that each instance of Table_B is linked to an instance of Table_A (not vice-versa ofc).
- Add a discriminant attribute to Table_A to indicate the name of the subclass to which a given instance belongs.
  (In the banking account example, you will have to add an attribute called "account_type" to the "Account" table which can take discriminant values as per designer's logic).

We cannot implement strict inheritance in its true sense. Why?

- This deals with an OOPS principle that an object cannot belong to more than one class.
- It can be linked indirectly via inheritance hierarchy to any number of classes but no direct linkages (we do not consider Multiple Inheritance in this discussion).
- Now again consider the banking system example,

- Account links to Savings, Current & Deposit Accounts
- Surrogate key of Account “account_id” is the foreign as well as primary keys for the three account type tables.
- You will first insert a new account row in the Account table.
- After this, there is nothing stopping you from inserting 1 row with the same account_id in each of the three account type tables.
- No constraint is violated because the account_id foreign key links to an account which exists in the Account table.
- But now you have an account that can be a savings, deposit as well as current account.
- So this OOPS principle is violated.
- **Key takeaway, Mutual exclusion of subclasses cannot be maintained.**
- You can try to sidestep this issue using triggers and condition checking with account_type, but that will be application specific whereas we want these rules to be as general purpose as possible.
- So in practice, to avoid this problem you will have to use triggers and/or handle it at the application-end

Sidenote: Using triggers makes the database constraints application specific

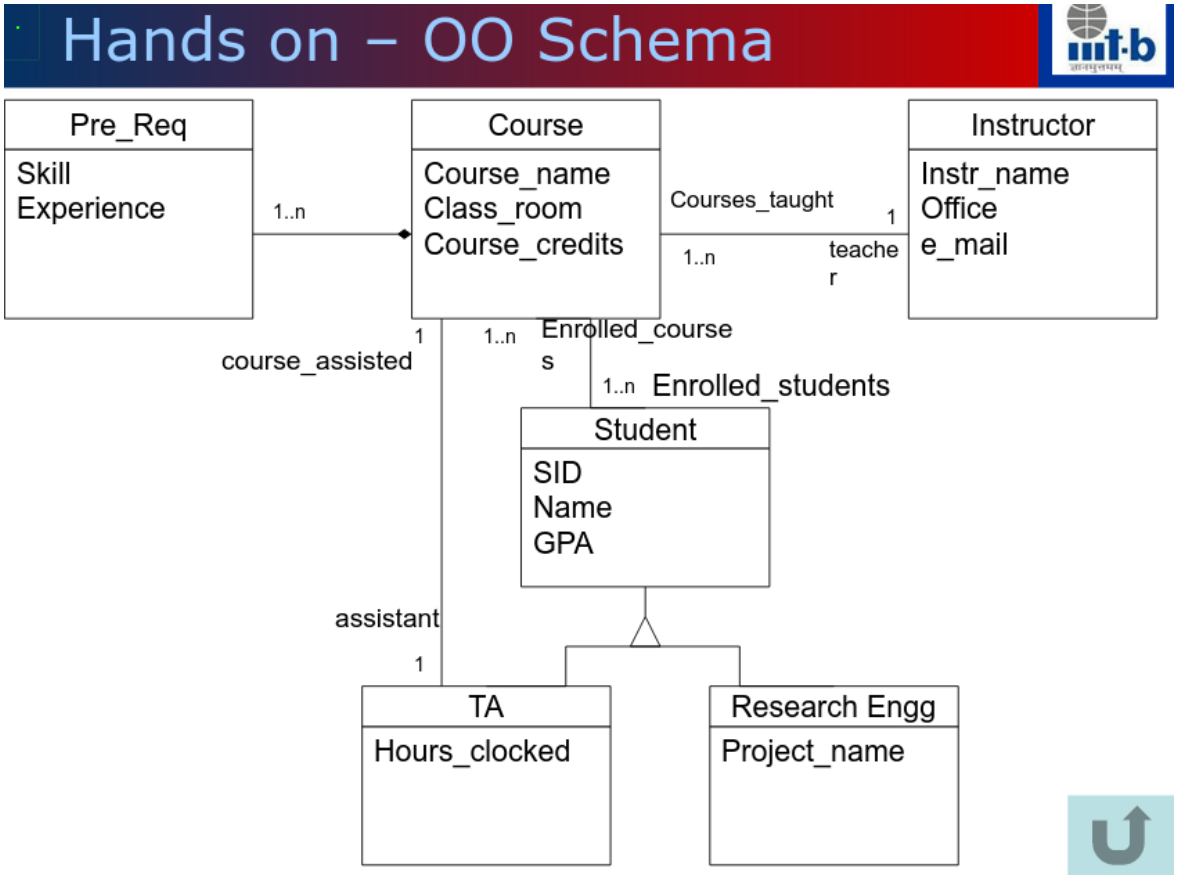
9. Aggregation

- Let Table_A be the container/parent class and Table_B be the child class.
- Add the surrogate key of Table_A as foreign key in Table_B.
- No need for NOT NULL constraint on the foreign key **(because a child class object can exist independently until it is attached to a parent)**.

10. Composition

- Let Table_A be the container/parent class and Table_B be the child class.
- Add the surrogate key of Table_A as foreign key in Table_B.
- Add a NOT NULL constraint on the foreign key **(because a child class object cannot exist independently of a parent class object).**
- **And because of deep deletion semantics we have to add the following,**
  Adding a ON DELETE CASCADE on the foreign key.
  That is, If you delete a parent/container object, then all the child components cannot exist and must be deleted => Cascade deletion

Hands-On OO Schema



A Possible Solution

COURSE					
ID_PK	Course_Name	Class_Room	Course_Credits	**Instr_ID_FK**	**Assisted_by_FK**

PRE_REQ			
ID_PK	Skill	Experience	**Crs_ID_FK**

RESEARCH_ENGG		
ID_PK	**Student_ID_FK**	Project_Name

INSTRUCTOR			
ID_PK	Instr_Name	Office	e_mail

TEACHING_ASST	
ID_PK	**Student_ID_FK**

STUDENT				
ID_PK	SID	Name	GPA	Student_Type

COURSE_STUDENT		
ID_PK	**Crs_ID_FK**	**Student_ID_FK**

Express Intro to SQL

[Only covering very basics of what's required]
We are now entering the Physical Schema phase of Conceptual Modelling which will generate vendor-specific SQL scripts.

Data Definition Language Statements (DDL)
Related to the definition of the schema.
CREATE
ALTER
DROP

Data Manipulation Language Statements (DML)
Related to the data stored in the table.
INSERT
UPDATE
DELETE
SELECT

Because we are still in Conceptual Modelling, we will only deal with DDL as of right now.

How to systematically implement a database in the Physical Schema phase?
You will separate the entire Physical Schema into SQL scripts (.sql files) that will execute some distinct functions.

Consider you have to implement a Bank Database called "bankdb", the implementation would consist of the following SQL files/scripts,

bankdb_create.sql => Would contain all the CREATE TABLE statements which would implement the schema that was finalized in OR Mapping phase (Logical Schema).

bankdb_alter.sql => Would contain all the foreign key constraints as ALTER TABLE statements.

SQL Syntax allows us to define foreign key constraints in CREATE TABLE itself but we should not do it. Why?
Because the referenced table should exist in the first place and this adds dependencies which can get very messy as more tables and subsequent foreign keys are added to the database.
So we can avoid all of that by first running "bankdb_create.sql" to ensure that all tables actually exist and only then run "bankdb_alter.sql" to add all the foreign key constraints.

bankdb_drop.sql => Would contain scripts for dropping all the tables.
Why do we need this?

Over the course of development, the database can undergo many changes including complete redesign of the table schemas in which case dropping the tables and running a new create script would be more efficient.

Because foreign key constraints could be violated on dropping, the dropping sequence has to be ordered such that there are no dependencies.
Another way of doing this is to just drop all the foreign key constraints via ALTER TABLE DROP and then run the DROP TABLE statements on all the tables.

bankdb_adddata.sql => Would contain INSERT & UPDATE statements that initialize the database.
So this is essentially the database initializer (the master data if you will).

Hands-On

(with CompanyHandout.pdf & Companydb.sql files on LMS)

cd to the folder where you have the “companydb.sql” file stored.

Start MySQL shell prompt
mysql -u <user_name> -p

CREATE DATABASE companydb;
SHOW DATABASES; // Shows created databases in MySQL
USE companydb; // Switches to companydb as current database
SHOW TABLES; // Shows created tables in current database (should be empty for now)
SOURCE companydb.sql // Executes the SQL file whose path is given to SOURCE command

Know the schema of any table using the DESCRIBE query.
DESCRIBE employee;
DESC employee;

Because we were already in the same directory we could specify filename directly, otherwise relative or absolute path is required.

16/11/21

Continuing the Hands-On stuff (for any doubts regarding what’s stored in the database and tables, refer to companyhandout.pdf)

SELECT statements

SELECT * FROM employee;
SELECT fname, lname FROM employee;

SQL is case-insensitive in terms of its commands, however created objects like tables and databases can be case-sensitive depending on the platform you are using. On Windows, the MySQL client is case-insensitive whereas running the MySQL shell from a Linux terminal would result in case-sensitive object names.

Aliasing

Assigning a temporary name to a column for the specified SELECT query (doesn't modify the underlying table in any way).

1. Using AS keyword
   SELECT fname AS first_name FROM employee;

2. Without using AS keyword
   SELECT fname first_name FROM employee;

3. When aliased column name has a space
   SELECT fname 'first name' FROM employee;

SELECT statements can also include expressions,
SELECT fname, salary * 12 'annual salary' FROM employee;

WHERE clause

Q. Select all employees in the table which belong to department number 4

SELECT * FROM employee WHERE dno = 4;

Can also use logic operators and compound conditions here as,
SELECT * FROM employee WHERE dno = 4 OR sex='M';

Nested Query

Q. Select all employees in the table which belong to Administration Department

Employee table only has the department number column, so for determining department name and matching it with department number we'll require a nested query.

SELECT * FROM employee WHERE dno=(SELECT dnumber FROM department WHERE dname='Administration');

Note that the following commands would be an error,

1. SELECT * FROM employee WHERE dno=(SELECT * FROM department WHERE dname='Administration');

   Debug Message: Operand to WHERE clause should contain 1 column

2. SELECT * FROM employee WHERE dno=(SELECT dnumber FROM department);

   Debug Message: Subquery returns more than 1 row
   So one row's dno cannot be checked with dnumber of multiple departments in the same WHERE clause because '=' is a two operand operation, for that you can use IN operator.

IN Operator

Q. Select all employees which work in Houston.

The dept_locations table maps dnumber to dlocation and we can use subquery to do the same mapping.

SELECT * FROM employee WHERE dno=(SELECT dnumber FROM dept_locations WHERE dlocation='Houston');

However this command fails with "Subquery returns more than 1 row" error because there are multiple departments located in Houston, so multiple dnumbers will be returned.
Because of this you should know what the return type would be to use the correct operation, in this case the IN operator should be used.

SELECT * FROM employee WHERE dno IN (SELECT dnumber FROM dept_locations WHERE dlocation='Houston');

You also have NOT IN operator
SELECT * FROM employee WHERE dno NOT IN (SELECT dnumber FROM dept_locations WHERE dlocation='Houston');

MAX()

Q. Find the employee with the highest salary

This command does not work,
SELECT * FROM employee WHERE salary = MAX(salary);

MAX() is an aggregation function and requires a separate SELECT clause for it.

SELECT * FROM employee WHERE salary = (SELECT MAX(salary) FROM employee);

Aggregation Functions

SELECT MAX(salary), MIN(salary), SUM(salary), AVG(salary), COUNT(*) FROM employee;

You cannot add not aggregation function columns in a SELECT query that returns an aggregation function column.
Example,
SELECT fname, MAX(salary) FROM employee;

Returned the following error,
ERROR 1140 (42000): In aggregated query without GROUP BY, expression #1 of SELECT list contains nonaggregated column 'companydb.employee.fname'; this is incompatible with sql_mode=only_full_group_by

COUNT(*) => Return number of rows in the table
COUNT(fname) => Return number of rows in the table that have non-NULL value for fname column.

Example,
SELECT COUNT(*), COUNT(super_ssn) FROM employee;
Returns, 8, 7

GROUP BY (Also called GROUP-ed Aggregation)

Q. Find out the average salary of employees working per department

SELECT dno, AVG(salary) FROM employee GROUP BY dno;

In the column list for the SELECT query with a GROUP BY clause, you can only use aggregation functions along with columns that are used in the GROUP BY clause.

SELECT fname, dno, AVG(salary) FROM employee GROUP BY dno;

However, for some reason this doesn't throw an error on Windows MySQL shell so be very careful of things like this. To ensure that this error is thrown you have to enable the "Full-Group-By" option which you can look up online.

GROUP BY with multiple columns,
SELECT dno, sex, AVG(salary) FROM employee GROUP BY dno, sex;

Returns average salary of all unique combinations of dno and sex values

```
mysql> SELECT dno, sex, AVG(salary) FROM employee GROUP BY dno, sex;
+------+------+--------------+
| dno  | sex  | AVG(salary)  |
+------+------+--------------+
|    5 | M    | 36000.000000 |
|    5 | F    | 25000.000000 |
|    1 | M    | 55000.000000 |
|    4 | F    | 34000.000000 |
|    4 | M    | 25000.000000 |
+------+------+--------------+
5 rows in set (0.00 sec)
```

HAVING clause

Only used after a GROUP BY clause.

SELECT dno, sex, AVG(salary) FROM employee GROUP BY dno, sex HAVING AVG(salary) > 30000;

The HAVING clause can only include columns which are included in the column list of the SELECT query, i.e. columns used in the GROUP BY clause or aggregation functions.

Cannot use WHERE after GROUP BY.

The sequence is, SELECT -> WHERE -> GROUP BY -> HAVING

ORDER BY

Sort employees in ascending order of their salaries,
SELECT * FROM employee ORDER BY salary;

Can also use ASC keyword,
SELECT * FROM employee ORDER BY salary ASC;

Can sort in descending order using DESC

SELECT * FROM employee ORDER BY salary DESC;

Can do multi-level sorting as,
SELECT * FROM employee ORDER BY dno, sex;

Sorts by dno first in ascending order and breaks ties in dno by consider value of ‘sex’ column, because it is a string column (CHAR(1)), lexicographic value of the ‘sex’ column is considered for breaking ties.

```
mysql> SELECT dno, sex FROM employee ORDER BY dno, sex ASC;
+------+------+
| dno  | sex  |
+------+------+
|    1 | M    |
|    4 | F    |
|    4 | F    |
|    4 | M    |
|    5 | F    |
|    5 | M    |
|    5 | M    |
|    5 | M    |
+------+------+
8 rows in set (0.00 sec)
```

Correlated Query

Q. Select all those employees which have salaries more than the average salary of their department.

You can get the average salary of each department using GROUP BY dno query on the employee table, but how to proceed further?

You need a nested query that checks whether the salary of an employee >= avg(salary) but how will you ensure that the current employee’s department is the same as the one being compared against?
You will need **table aliasing** for this.

SELECT * FROM employee e1 WHERE salary >= (SELECT AVG(salary) FROM employee e2 GROUP BY dno HAVING e2.dno = e1.dno);

Think on this a bit further and realize just how unnecessarily complicated this, there is no need for HAVING, you can just use the following,

SELECT * FROM employee e1 WHERE salary >= (SELECT AVG(salary) FROM employee e2 WHERE e1.dno = e2.dno GROUP BY dno);

Now even GROUP BY is unnecessary and the WHERE clause is actually sufficient enough,

SELECT * FROM employee e1 WHERE salary >= (SELECT AVG(salary) FROM employee e2 WHERE e1.dno = e2.dno);

This is called a correlated query because the inner query is correlated with the outer query.
Such queries are evaluated differently from general nested queries because they have to be evaluated for each individual row separately, i.e. the WHERE clause will have to be evaluated separately for each row of the outer query table.

JOIN

Q. Return the department name for each employee

SELECT CONCAT(fname, ' ', lname) full_name, dname FROM employee e JOIN department d ON e.dno = d.dnumber;

CONCAT() function is also used there as you can see and its usage is pretty straightforward.

Can also use INNER JOIN instead of JOIN keyword.

You can also do JOIN without even using the JOIN keyword.

SELECT CONCAT(fname, ' ', lname) full_name, dname FROM employee e, department d WHERE e.dno = d.dnumber

Some theory revision,
What is the necessary condition for the INNER JOIN query to work?
The join should be lossless and dependency preserving and for this the condition is that,
There should be a common column between the two tables and it has to be a key in atleast one of the tables.

So for each JOIN condition you have to ensure that one of the columns is a foreign key in one table that is referring to a primary key in the other table.

SELECT fname, dnumber FROM employee e, department d;
This will return the Cartesian or Cross Product / Cross Join

Multi-table JOINS

SELECT fname, dname, pno, hours

To return these columns you need access to three tables, employee, department and works_on

You must use two join conditions to merge the three tables as follows,

// Without JOIN Clause and WHERE Clause only
SELECT fname, dname, dno, pno, hours FROM employee e, department d, works_on w WHERE e.dno = d.dnumber AND e.ssn = w.essn;

// With JOIN Clause
SELECT fname, dname, dno, pno, hours FROM employee e JOIN department d ON e.dno = d.dnumber JOIN works_on w ON e.ssn = w.essn;

Q. Find out how many projects are being worked on by each department.
SELECT dname, COUNT(*) FROM employee e, department d, works_on w WHERE e.dno = d.dnumber AND e.ssn = w.essn GROUP BY dname;

Q. Find out employees that have more salary than the average salary of their department without using nested query.
SELECT fname, e.dno, salary FROM employee e, (SELECT dno, AVG(salary) d_avg_sal FROM employee GROUP BY dno) d_avg WHERE e.dno = d_avg.dno AND salary >= d_avg_sal;

A bit complicated but you'll get it once you analyse it :(

OUTER JOIN

Q. Find out dependents of employees and if employee has no dependent return NULL

SELECT fname, dependent_name FROM employee e LEFT JOIN dependent d ON e.ssn = d.essn;

Can also use LEFT OUTER JOIN

Note that this query's output is the same as,
SELECT fname, dependent_name FROM dependent d RIGHT JOIN employee e ON e.ssn = d.essn;

Just flipped the position of the tables in the JOIN and changed LEFT to RIGHT JOIN.

FULL JOIN / FULL OUTER JOIN is the union of LEFT and RIGHT JOIN
MySQL doesn't support FULL JOIN

Views

View is not a temporary or dynamically created table.
It does not occupy disk space.

In simple words, a view is an alias for a query.

The output of any relational query is another relation itself, i.e. a table.
If we assign a name to this created table, it is like assigning an alias to the associated query and this is exactly what a view is.

And using this interpretation we can identify the properties of a view,
Everytime you access data from a view you are accessing data from the actual table or tables that are stored.
Everytime you modify data stored in a view you are modifying data in the actual table/s.
Ofcourse you can only perform DML operations in views if there is a primary key present as part of the view created.
Views are used for SELECT queries only but they can be used for DML queries as well if the conditions for the corresponding DML operation like presence of PK is satisfied.

CREATE VIEW rich_folks AS (SELECT * FROM employee WHERE salary >= 40000);

And now the name “rich_folks” can be used anywhere just like a normal table name.

So this also tells you why views are used,
For the convenience that they provide, views can be used to store results of very complex queries and so we don’t have to write them out every time the query’s result is required.

For the purpose of implementing security in a database.
Consider the following example,
In a company database there is a table which includes many many attributes related to different domains like accounts department data, developer related data and so on.
So you only want the accounts department people to be able to view the financial data present in a table and so on for the developers, etc.
In this case what you can do is create a separate view for each subset of data required for a department, only allow that department to access the corresponding view table and block direct access to the underlying base table.
So there will be one view for accounts department data, one for developer data, etc. and only accounts people can access the accounts department view, same for developers, etc.

[Because view is just a table, it can also have access rights like normal tables. Because tables are permanent, views are also permanent.]

So now we can use a view for this query,
SELECT fname, e.dno, salary FROM employee e, (SELECT dno, AVG(salary) d_avg_sal FROM employee GROUP BY dno) d_avg WHERE e.dno = d_avg.dno AND salary >= d_avg_sal;

And then just refer to the view created to make the queries more readable and straightforward.

Note: It was mentioned that the view doesn't consume any disk space. This is kinda misleading because **the schema of the view is stored on disk and for a view its schema is the SELECT query that was used to create it.**
You can check this by checking the "views" table in the "information_schema" database
Use the following code snippet to identify that MySQL indeed stores the creation query for a view.

```
USE information_schema;
SELECT table_name, view_definition FROM views WHERE table_name = 'rich_folks';
```

And you can see the same SELECT query that you wrote to create the "rich_folks" view in the view_definition column data.

In this way, we can use the information_schema to learn a lot of information about tables, columns, views, etc.

CTE

Using views for everything like this can be very tempting and can result in a lot of redundant views which involve complex queries.

To avoid this, a new feature was added to SQL called CTE (Common Table Expressions).

The syntax for this is as follows,
WITH cte_table_name_1 AS (SELECT expression), cte_table_name_2 AS (SELECT expression) SELECT …

This essentially creates cte_tables which can be used just like a regular table for the SELECT query that follows all the cte_table declarations.

Example,

```
WITH
rich_folks AS (SELECT * FROM employee WHERE salary >= 40000),
poor_folks AS (SELECT * FROM employee WHERE salary < 30000)
SELECT r.fname, r.salary, p.fname, p.salary FROM rich_folks r, poor_folks p;
```

This query doesn't really do anything but you get the point about how we can use these CTE statements to properly structure complex queries in a systematic way without having to rely on the overhead of views.

Also note that CTE is not a database object because it has no persistence.

Interview Question: Comparison of Views & CTE

Both are similar because they allow you to create an alias for a query.
However views create a persistent table that lasts until it is dropped using DROP VIEW whereas CTE creates a table that is only persistent for that query.
Views provide the benefit of simplifying complex queries as well as implementing database security whereas CTE only allows us to structure complex queries in a systematic way by creating the complex queries in the WITH statement and only then use those tables in the SELECT statement later.

If the complex query that you are writing is going to be reused many times then it makes sense to write it as a view instead of a CTE.

Nested queries can be completely eliminated if you can use CTEs.

Triggers

They execute DML statements **on a table** when specific conditions are satisfied.
Triggers are also database objects.
In general, triggers are not preferred because in a large project triggers can execute something in the background that is a side-effect and something which is not apparent inherently. And maybe a trigger can lead to a cascading effect because executing a trigger on one table can lead to execution of a trigger on another table and so on.

Each DBMS has its own way of implementing the concept of Surrogate Keys for a table. Like AUTO_INCREMENT for MySQL.

19/11/21

Database Programming

Imperative Programming

- SQL is a Declarative Language (Non-Procedural).
- Declarative language queries specify what we want from the data which is exactly what we do in SELECT statements and son.
- It also makes sense that you cannot design a programming language using declarative constructs alone because implementing dynamism in declarative constructs is very difficult or not possible (example the elements used in a SELECT query like table name, column names, WHERE clause conditions, etc. are static and doesn't change depending on the situation).

- In simple words, SQL only allows us to execute one statement at a time with no other dynamic aspects like conditional statements.

This is why the Imperative Programming paradigm was introduced.

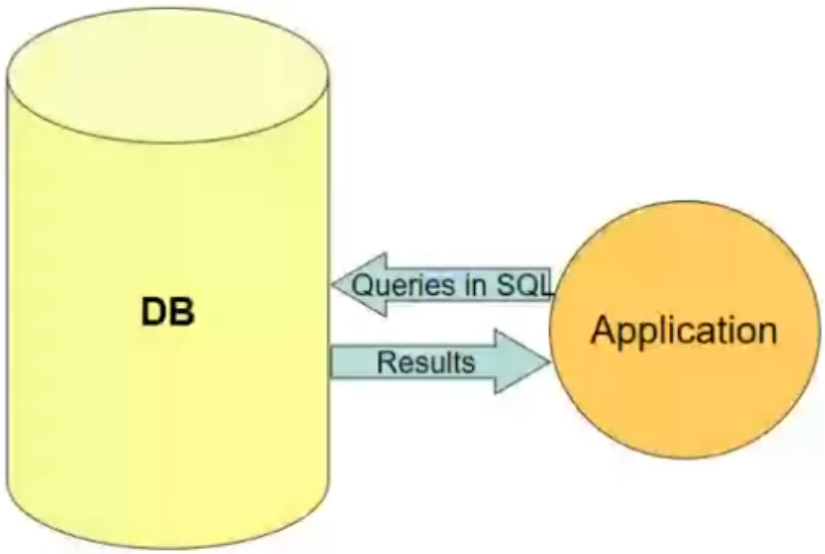
It provides us the following functionalities,
Execute sequence of statements.
Execute statements conditionally.
Execute statements iteratively.

Note: SQL has some new conditional constructs as part of the standard now like,
CASE
SOME IF
But this is at a very basic level and SQL still cannot be used for complex interactions.

DB Programming Approaches

Sir broadly classifies these into 6 categories as described below.

1. SQL



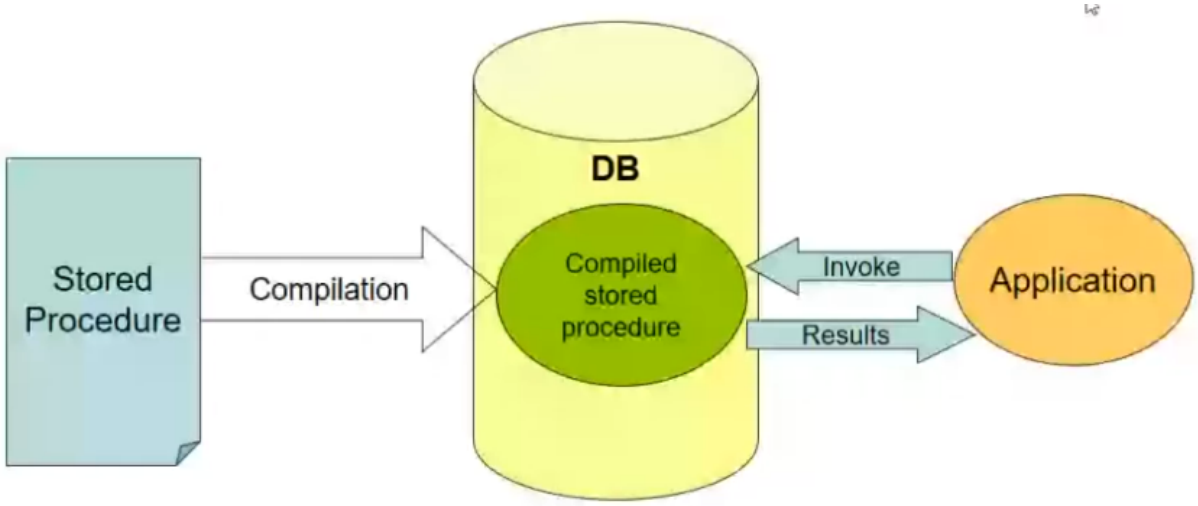
SQL is the industry standard **non-procedural language (declarative)** for relational database access.
In our case, MySQL client is the application that is firing SQL queries typed by us.
Advantages:

- Easy to use for developers.
- Behaves consistently across all vendor implementations (to a large extent).

Disadvantages:

- Difficult to use for end-users.
- Difficult to implement complex scenarios like unforeseen end-user interactions, complex data validations, etc.

2. Procedural Extensions to SQL



SQL is extended with imperative programming constructs (conditionals, loopings, exceptions, etc.). These are typically implemented as stored procedures.

A stored procedure is a sequence of SQL statements with imperative programming constructs that can be compiled by the database itself. It is like a function for our better understanding.

When an application invokes a stored procedure, the database fetches the stored procedure, compiles it and stores the compiled stored procedure in the database itself.
Now the database can execute any invocation of the stored procedure directly and return the results back to the application.

Before, the database would only return the requested data and any programming had to be implemented at the application level which can introduce significant overhead depending on where the application is wrt the database.
Whereas here the database itself performs the computation and only results have to be transferred, so server-side processing is done.

[The application is still the MySQL client]

Examples, PL/SQL in Oracle, T/SQL in MS SQL Server

Advantages:

- Finer control in processing logic as we can now use imperative programming constructs as well.
- Better performance because of server-side processing.

Disadvantages:

- Not portable across DBMS, i.e. the stored procedure extensions to a DBMS is vendor-specific and procedures in one DBMS cannot be shifted to another.

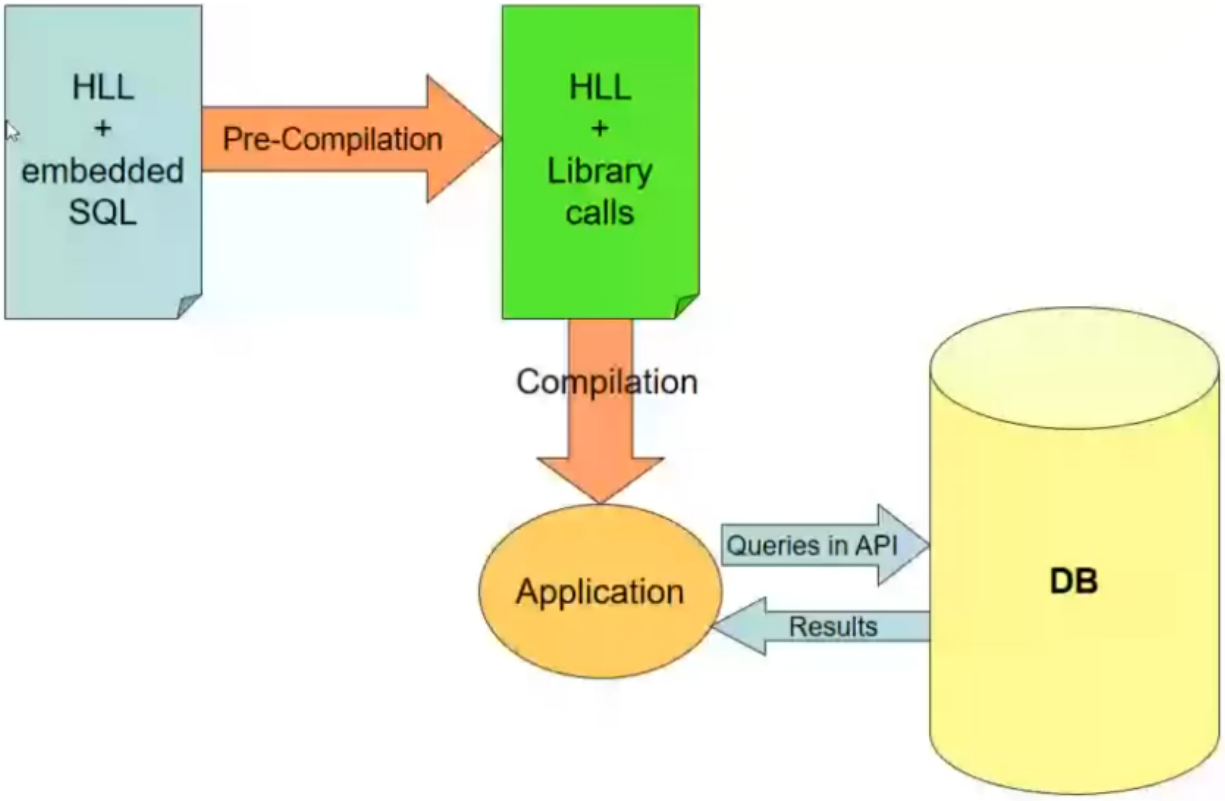
PL/SQL Stored Procedure Example

```
CREATE OR REPLACE PROCEDURE changesalary (ide number, ch_salary number) IS
sal number; /* Local variable "sal" of type "number" */
BEGIN
      SELECT salary
      INTO sal /* INTO = Assigning SQL output to local variable "sal" */
      FROM employee
      WHERE employee.id = ide;
      /* We can use dynamically coded variables in WHERE clause now */

      sal := sal + sal * ch_salary * .01

      UPDATE employee
      SET salary = sal
      WHERE employee.id = ide;
END;
```

You can also use IF statements, LOOPs, EXCEPTIONs, etc.

A Stored Procedure would also be stored as a schema object.

3. Embedded SQL



We take a HLL like C, C++ and modify it to support SQL, so SQL code is embedded within HLL. But the C compiler doesn't understand SQL statements, so the code goes through a pre-compilation phase which is processed by the pre-compiler and in this phase all SQL statements will be replaced by C library calls to the database.
And this can now be executed by the C compiler (HLL + Library Calls) which will give us an executable file.

The application can use this executable file to fire queries to the DB.

Advantages:
No need for learning a new procedural language.

Disadvantages:

- Not portable across DBMS.
- Impedance Mismatch: Mixing of incompatible programming paradigms.

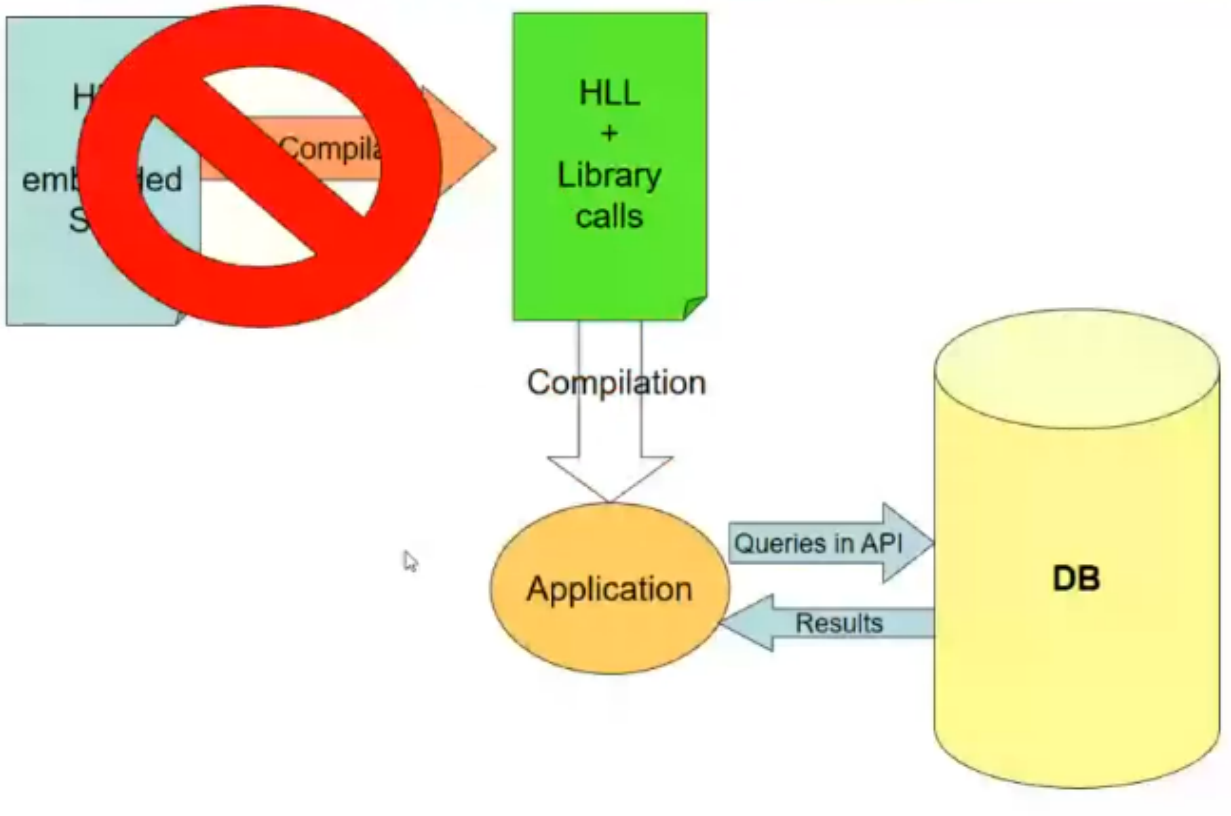
Pro *C Example

```
int loop;
   EXEC SQL BEGIN DECLARE SECTION;
        varchar dname[16], fname[16], …;
        char ssn[10], bdate[11], …;
        int dno, dnumber, SQLCODE, …;
   EXEC SQL END DECLARE SECTION;

loop = 1;

while (loop) {
   prompt ("Enter SSN: ", ssn);
   EXEC SQL
        select FNAME, LNAME, ADDRESS, SALARY
        into :fname, :lname, :address, :salary
        from EMPLOYEE where SSN == :ssn;
        if (SQLCODE == 0) printf("%s", fname, …);
        else printf("SSN does not exist: ", ssn);
        prompt("More SSN? (1=yes, 0=no): ", loop);
   END-EXEC
}
```

This is a C program with embedded SQL calls

As you can see for conditional and looping here, we use the host language's constructs.

4. Database APIs



Database vendors publish proprietary APIs that give low level access to the database.
Instead of writing database queries in SQL, we'll use database APIs in the host language itself that will do the job of accessing the database.
So no pre-compiling stage is required here.
We can directly compile the code and the application can execute the program.

Example: OCI (Oracle Call Level Interface)

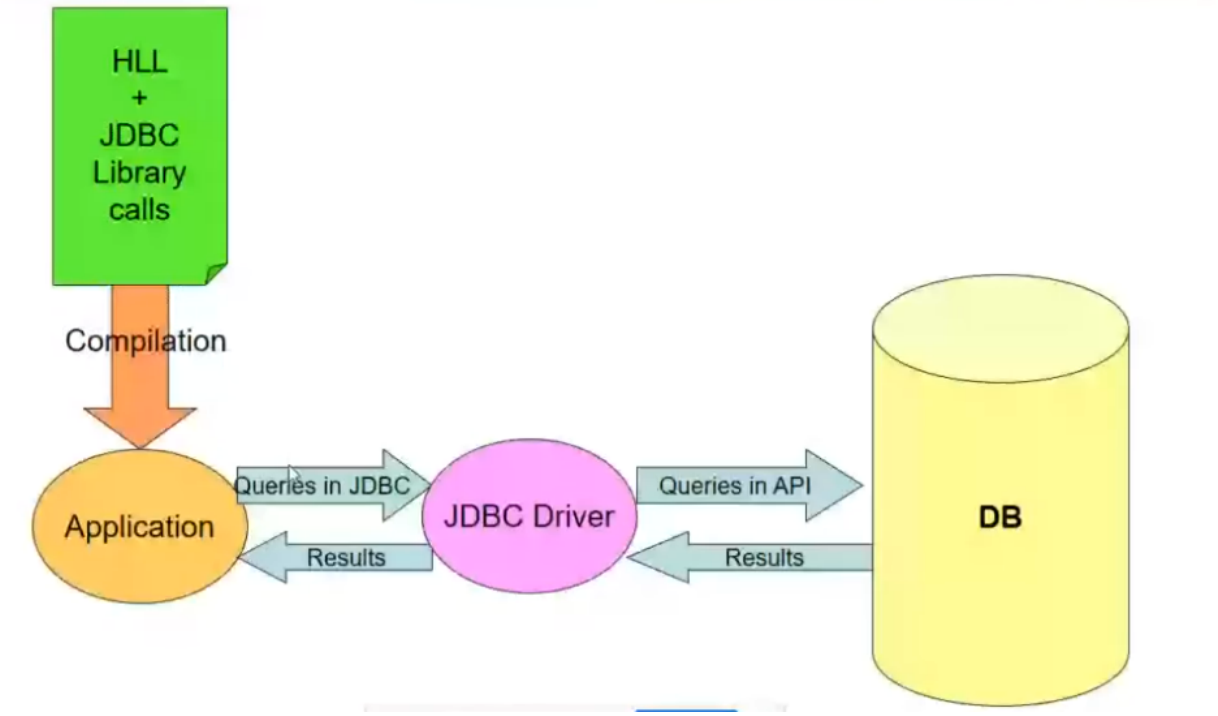
Advantages:

- No impedance mismatch as there is no need to mix two different languages.
- Better performance because it is low-level.

Disadvantages:

- Not portable across DBMS.
- APIs may be complex to use.

5. ODBC/JDBC



ODBC (Open Database Connectivity) was originally proposed by Microsoft as a DBMS-independent way to connect to any compliant data source.
JDBC is the Java-equivalent of ODBC.
We still use library calls here but the library calls are much much simpler to learn and use and to set up as well which was not the case with Database APIs.

The biggest change that was made here is that it was standardized.
How was this done?
First the library was made available as an open API and then each vendor was expected to make a vendor-specific software called the Driver which would be able to translate the open API queries into the respective database queries, query the database and send the results back to the application.

The JDBC API defines interfaces and classes for writing database applications in Java.

Advantage:
Highly portable and standardized across DBMS

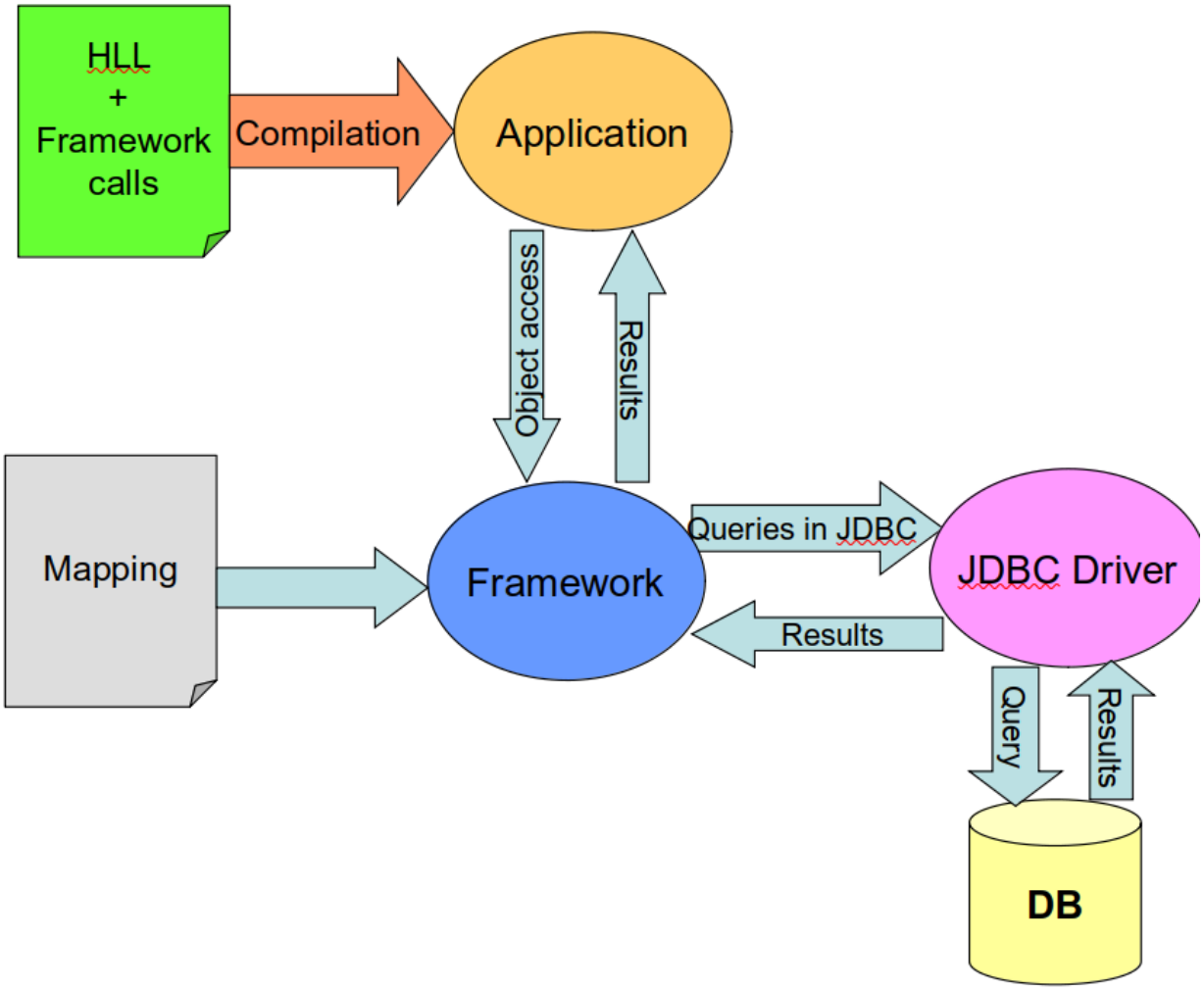
Disadvantage:
Impedance mismatch in programming styles, one end is the DBMS which uses relational model and the other end is the HLL which in the case of JDBC is Java which uses Object Oriented model.

So if we have to avoid this mismatch of styles while using JDBC we have to convert a given database result which is a bunch of rows in a table into a collection of objects (one object for each row in result).
Or we can continue to operate with both formats and accommodate them.

JDBC Sample Code

```
// Load the database driver
    Class.forName( "com.mysql.jdbc.Driver" ) ;

    // Get a connection to the database
    Connection conn = DriverManager.getConnection( "jdbc:mysql:retaildb" ) ;

// Get a statement from the connection
    Statement stmt = conn.createStatement() ;

    // Execute the query
    ResultSet rs = stmt.executeQuery( "SELECT * FROM Customer" ) ;

    // Loop through the result set
    while( rs.next() )
      System.out.println( rs.getString(1) ) ;

    // Close the result set, statement and the connection
    rs.close() ;
    stmt.close() ;
    conn.close() ;
```

6. Database Frameworks



Here impedance of two styles is avoided by introducing the Framework (which is a more customizable library).
Frameworks help with database access without the need to mix two different languages like Java and SQL.
They primarily provide automatic mapping between relational rows in DBMS and objects in OOPS.
We write code in HLL to access objects of the database and the framework translates this into SQL based API queries (JDBC queries) which the Driver can map to actual DB queries.
So SQL is completely eliminated from Application code.

More details on how this works,
We write queries in HLL which asks the framework to access objects of the database. Each object is mapped to a row in a table and so the framework can access this mapping in order to return the specified object to the user in HLL and also fire queries to JDBC using this mapping.

So the framework essentially serves as a mapper from Object Oriented model to relational model.
The Framework generates **plumbing code** (auto-generated) whereas this same code would have to be manually written if we were using plain ODBC/JDBC.

Example, Hibernate (framework on top of JDBC).

Advantage: No impedance mismatch (because of no need for SQL).

Disadvantage: Not portable as the code is specific to a framework, sometimes hard to debug.

Java Persistence API (JPA)

Equivalence of JDBC for ORM (Object-Relational Mapping).
JPA operates directly with Java objects.
Just like JDBC, JPA is also an interface specification.
Needs to be combined with an ORM implementation framework like Hibernate, Toplink, etc.

JPA is essentially the standardized version of the Database Framework. Hibernate is a very specific implementation of JPA but it got so popular that it was later standardized as well.

JPA Mapping Example

```
@Entity
@Table(name = "USERS")
public class User {
   private Integer id;
   private String fullname;
   private String email;
   private String password;


   @Column(name = "USER_ID")
   @Id
   @GeneratedValue(strategy =
GenerationType.IDENTITY)
   public Integer getId() {
      return id;
   }
```

JPA Relationship Annotations

Hibernate Association Mapping Annotations	
@OneToOne	import javax.persistence.OneToOne;
@ManyToOne	import javax.persistence.ManyToOne;
@OneToMany	import javax.persistence.OneToMany;
@ManyToMany	import javax.persistence.ManyToMany;
@PrimaryKeyJoinColumn	import javax.persistence.PrimaryKeyJoinColumn;
@JoinColumn	import javax.persistence.JoinColumn;
@JoinTable	import javax.persistence.JoinTable;
@MapsId	import javax.persistence.MapsId;
Hibernate Inheritance Mapping Annotations	
@Inheritance	import javax.persistence.Inheritance;
@DiscriminatorColumn	import javax.persistence.DiscriminatorColumn;
@DiscriminatorValue	import javax.persistence.DiscriminatorValue;

JPA Sample Code

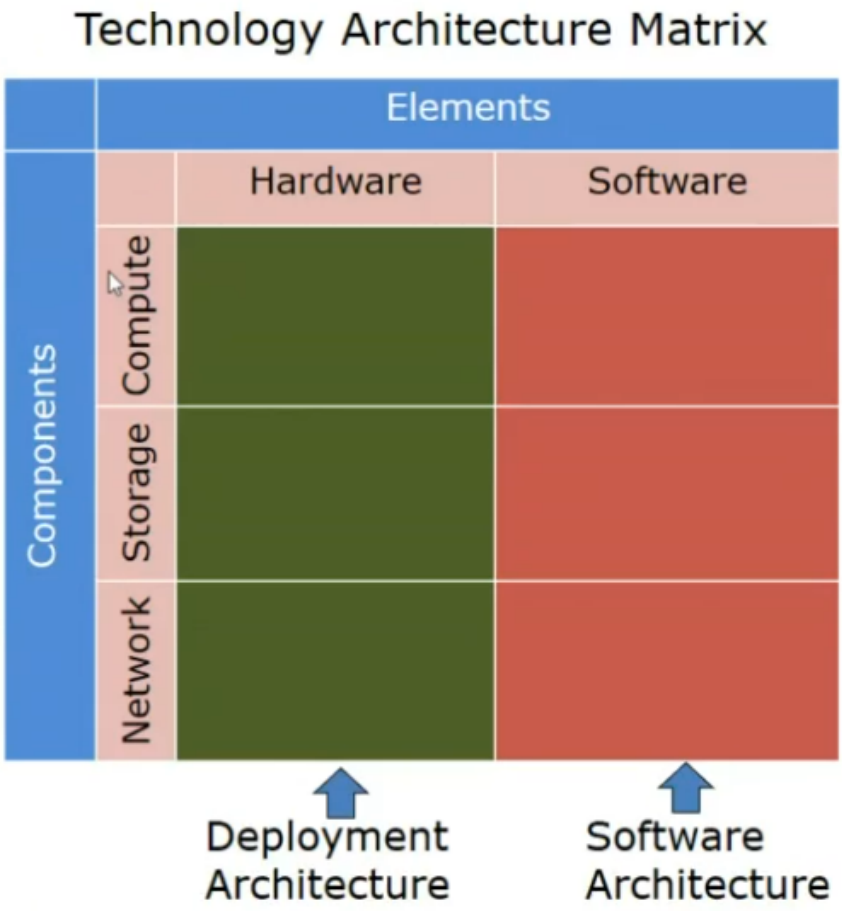
```
EntityManagerFactory factory = Persistence.createEntityManagerFactory("xyz");
EntityManager entityManager = factory.createEntityManager();

entityManager.getTransaction().begin();

User newUser = new User();
newUser.setEmail("billjoy@gmail.com");
newUser.setFullname("bill Joy");
newUser.setPassword("billi");

// To INSERT
entityManager.persist(newUser);

entityManager.getTransaction().commit();

entityManager.close();
factory.close();
}
```

23/11/21

Technology Architecture

Consists of 3 dimensional specification that involves a combination of hardware and software and the society/business/enterprise/problem space of the problem that we're solving, this third component is often called the IT solution.

Example, if we don't consider the society/business that we're solving the problem for, we might end up making a website based application for farmers which is absolutely useless because the closest we can bring farmers to technology in an easy way is mobile phone, so we need to actually build mobile app in this case.



This assumes that we already have a good understanding of the third dimension of the architecture, i.e. the society elements.

Compute => How does the processing take place?

Storage => How is information persistence (ability to store and retrieve information for an extended period of time) managed?
Network => How is the communication with users established?

All three components involve hardware and software aspects as you can probably think of.

Elements of Technological Architecture

Software Elements

1. COTS (Commercial Off The Shelf) vs Bespoke
    - Whether the problem requires a custom solution / bespoke solution or can we repurpose an existing commercial product/solution to fit the solution to the given problem (using a COTS solution) or can we build a solution that can be generalized to solve more problems (building a COTS solution).
    - Using a COTS product **if applicable** can be a very time efficient solution as compared to Bespoke solution.
    - **If you are the one making a COTS product, then comes the question of how flexible is your product?**
    - Flexibility is provided in two ways, configuration and customization.
    - **Configuration allows you to modify config files.**
    - **Customization allows you to modify the code directly.**
    - You generally shouldn't work with bespoke solutions if your core business is not software development because development of the product is fine to an extent but an even bigger issue is the maintenance aspect of the created bespoke solution product.

2. Open Source vs Proprietary
    - Using an open source product or a proprietary product & Making an open source product or a proprietary product.
    - The main benefit of using open source products is that it's at zero cost.
    - The problem with being open source and using open source products is that ownership becomes debatable.
    - Another major issue is security as the entire codebase is open for anyone to see and exploit.
    - IS WHAT YOU WOULD THINK BUT **IT IS WIDELY ACCEPTED THAT BEING OPEN SOURCE IS NOT A SECURITY RISK. WHY?**
    - Because if anyone can exploit a security vulnerability in your code, similarly anyone can also help fix this vulnerability and that is the spirit of open source development.
    - A workaround for the ownership problem to developing open source products is the introduction of community and enterprise editions.
    - The community edition is fully open source and is free whereas the enterprise edition charges you a premium for maintenance of the product, whereas the

community edition's maintenance is done by the community themselves as and when problems are found and fixed.

3. There can be a number of questions regarding software elements here but those two were the major ones to be discussed here.

Hardware Elements

Compute
Storage
Network

Already discussed briefly before.

Architecture Principles

Rules governing the choice and use of hardware.
Rules governing the choice, creation and use of software.
Rules governing the interface between software and hardware.

Example of a filled Technology Architecture matrix,

		Elements	
		Hardware	Software
Components	Compute	• Web server • Database server	• Operating Systems • Web Server • Application Server • DBMS
	Storage	• SAN • NAS • SATA	• Storage O/S • Backup Software
	Network	• Routers • Switches • Cabling media	• Network O/S • Firewall • Proxy servers • Load Balancer

The hardware column of this matrix essentially gives you a **Bill of Materials** for which you can place an order.

Software Architecture

Subset of Technology Architecture that includes all the software required to provide compute, storage and networking capabilities.

Software Architecture Styles

- A style specifies a consistent mechanism for designing software.
- A given IT solution may choose a mixture of styles to suit the specific needs of the solution.
- A given application in an IT solution generally is based on only one style.

Principles in Software Architecture are as follows,

1. Principle of Cohesion

Degree to which different components **within a layer** are focused on a single objective.

Advantages of High Cohesion:
Improves consistency (helps with maintenance and debugging)
Increases reuse
Plug and play (ability to take out one component and replace it with something else, example, replacing web-app frontend with a mobile app frontend)
Ease in engineering (development and testing)
Localization of expertise

Binding forces of Cohesion:
The binding forces refer to the different criteria under which two or more components can be grouped together to improve cohesion.

1. Logical => Things that are to be considered as one objective because they are logically related to each other.
   Example: Credit Card payment and NEFT payment can be grouped together under the objective of “payment”.
2. Temporal => Things that are to be considered as one objective because they happen at the same time.
   Example: Initialization of DB, web server and OS; objective of “initialization”.
3. Procedural => Example: Loan verification, Loan evaluation, Loan underwriting. These are related to procedural handling of loans. Can link this to methods of a class in OOPS. Most intuitive approach out of the given 6.
4. I/O Oriented => Example: PO (Purchase Order) Printing, PO Saving, PO Mailing. Related to input/output functionality of an entity (in this case PO)
5. Sequential => Set of actions that need to be done in a sequence are considered as one objective (like a DBMS transaction). Example, Search -> Select -> Add -> Checkout, “purchasing product” is the objective.

6. Functional => Self-contained/Single functions form a single objective.

2. Separation of Concerns

Governs relationships **across layers**.
Minimize overlap of responsibilities across layers.
Responsibilities are well-defined.

Advantages:
Avoid duplication
Better maintainability
Reduces coupling/dependencies across layers

More principles are to be covered later.

Overview of Deployment

What?
Deployment is the process of implementing an IT solution consisting of all the necessary hardware and software.

Where?
On-Premise (On-Prem)
Cloud

Who?
System admin, Hardware specialists, Network admins, etc.

When?
Once a “ready-to-use” product is available.

How?
Manual or Automated or a combination of both.

Why?
Ease of use, Better agility, Cost efficiency.

30/11/21

Layered Architecture

Layered Architecture is a software architecture style/pattern.

What is a Layer?

- It is a collection of services/functions.
- It has to be cohesive.
- There must be a separation of layers, so there cannot be overlap between two layers.
- There has to be a well defined list of concerns/tasks for each layer.
- A layer can depend on other layers or be a dependency for other layers, i.e. provider and consumer.
- Layers can be “OPEN” or “CLOSED”.

Layers must be cohesive and separation of layers is required => Principle of Cohesion and Separation of Concerns must be followed.

Closed Layer
A layer is considered to be closed if its interaction is permitted only across adjacent layers.

Open Layer
A layer is considered to be open if it provides shared services accessible across all layers.

1-Tier Architecture (Monolithic Design)

In this, the entire functionality of the application is associated with just 1 Tier/Layer, so everything is cobbled up together and it can’t be called systematic as such.
Clearly it’s a very old design pattern and it is used with mainframe applications, legacy databases and dumb terminals (no computing power, directly connected with mainframe servers to display information).

These legacy databases refer to predecessors of the relational data model like hierarchical data model, network data model and so on for which separate DBMS’ existed.
COBOL, IMS (DBMS for hierarchical data model), etc. were the tools that were used with such data models.

The main disadvantage of this architecture is clearly visible, there is no separation of concerns. So if anything breaks, you have to figure out exactly where in the entire application the issue happened.
Other issues include having to house the entire hardware in a single center as there’s no interconnectivity between different components.

The biggest non-technical disadvantage of this architecture is that the customer had no choice in selecting software/hardware for their particular solutions.
For example, if a customer wants to solve a particular problem and goes to a solutions provider, they have no choice but to get the entire software and hardware stack from that provider only as everything is bundled up as one package. You cannot do something like, DBMS will be given by one provider, front-end by another, hardware by another, OS by another, etc.

This can cause monopolies in the solution provider space because a particular solutions provider will create their own processors -> instruction sets -> hardware -> OS -> software -> everything basically.
So everything is proprietary and there is no way you can mix and match.

To counter the proprietary OS issue, the **UNIX OS** was proposed and implemented in order to finally create a publicly accessible OS. **The key advantage of this was that it was built to work on any processor architecture.**

Once more such open systems started popping up, there were more options for the customer to mix and match the tech stack at the server-side.

And once more options were available, the principle of cohesion and separation of concerns were applied to break architecture into 2-Tiers.

2-Tier Architecture [Late 90s to early 00s]

And this 2-Tier Architecture is the Client & Server architecture.

Tiers	Technology	Tools
Client Tier	Desktop PCs Windows GUI ODBC	Microsoft Windows Visual Basic Developer 2000
Server Tier	Unix Servers Relational Databases	HP-UX, Solaris Oracle / DB2

Here, the main separation was that the data (databases + servers) and the processing of data (client-side devices) were separated into two different components called the Server & Client tiers, respectively.

The other big change happened on the front-end side of things.
Before the front-end only consisted of dumb terminals/green screen monitors, now Apple & Microsoft introduced Graphical User Interfaces for their OS in order to make life simpler. These desktop PCs which ran Windows & macOS were not dumb terminals because they required

their own separate processing power and used that power for the GUI processing as well as many other things.

So data resides on the server and processing of data is done on desktop PCs/clients.

This is when Visual Basic and ODBC were introduced by Microsoft to create open client-side standards.

And the final important change was the introduction of Networks. There was still no Internet as such but Networks helped connect client and server without the need of cables which were required for the dumb terminals in the 1-Tier Architecture.
Note: The only networks available were of LAN type.

The three main proponents to 2-Tier Architecture were the emergence of open systems and the UNIX OS, Microsoft pushing open standards and the emergence of LANs.

Disadvantages:
The introduction of PCs increases the cost as compared to dumb terminals and the processing load is put on the client devices which seems like a not so optimal design choice.

If any updates are made to the client-side applications, then the updates have to be individually installed on each client PC because they are all running their own version of the application (remember that there is no internet still, just LAN).

Sir mentioned that his team faced this specific issue many times because employees used to go on vacation and by the time they came back, the app had already been updated many times and it is possible for the employees to continue their work on the old version itself, so they resolved this issue by including a version number parameter in their database as well as in the login of each client. So if the client's version number doesn't match with the one in the database then updates would be forced.

Advantages:
An advantage that this architecture provides is that the 1-Tier Architecture had a Single Point of Failure, where if the mainframe failed, then all the dumb terminals connected to it fail whereas in this case, if the mainframe fails, the PCs can still operate as required, only interactions with the database will need to be taken care of or perhaps if only one PC fails, then remaining PCs will still function as normal.

Now as the demand for such architecture rose, more and more devices/clients were connected on the same server. This put a tremendous load on the server as it only dealt with database interactions, i.e. SQL, no imperative programming is permitted, so huge amounts of data transfer happened between many, many clients and one server, so network traffic and the network in general became the bottleneck.

2 ½ -Tier Architecture

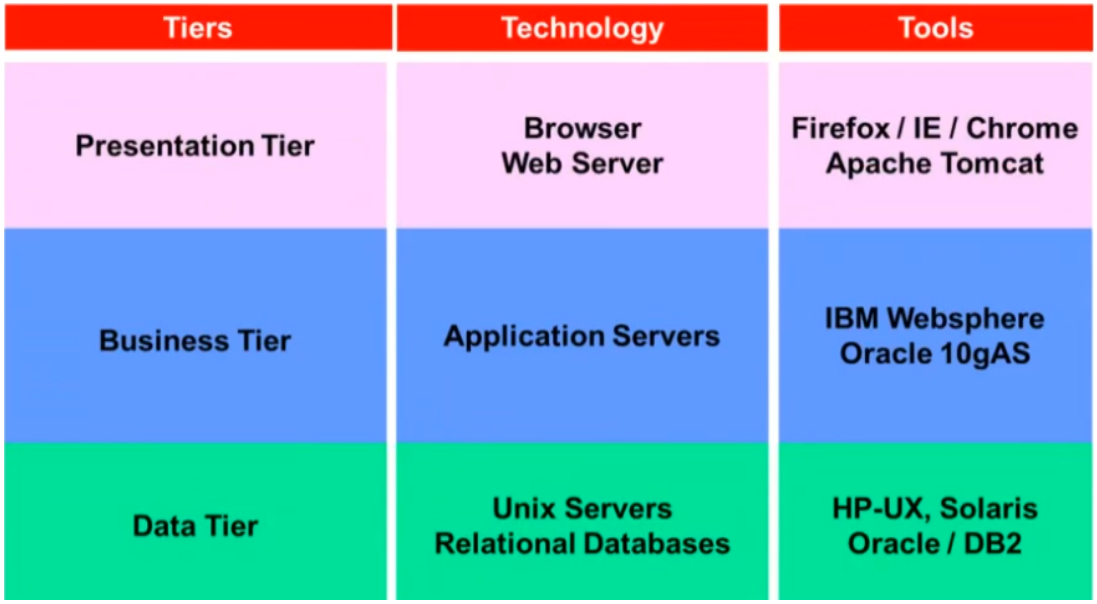
Because of the network bottleneck problem described before, the client side was broken down into three components,

- User Interaction
- Biz Logic
- Database Access

The reasoning behind this breakdown is by using the principle of cohesion and separation of concern on the client tier.

Among these only User Interaction was left in the client side and a stop-gap solution was made by creating a ½ Tier which included Biz Logic + Database Access.
This service was provided with the help of Stored Procedures, e.g. PL/SQL which as we have seen before provided all the imperative programming facilities with SQL.
This PL/SQL code is stored in the DBMS itself and so there's no communication over the network in this case as this ½ Tier can directly communicate with the database on the server.

3-Tier Architecture



This is a proper Layered Architecture and each layer in this is a Closed Layer.
So there's no user interaction in the data tier, no SQL in the presentation layer and so on.
And this is the architectural principle that guides our design of layers.

The main proponent behind this architecture is the advent of web browsers and the internet.

How did this approach solve the problem of having to install the updated application on each client PC?
By avoiding the installation altogether!

We use the browser to communicate with the web server which provides application access on the browser itself via the internet.

Note: For deployment of web-apps in Java, we package the entire application into a **WAR file** (Web Archive), whereas in the proper application case, we used .exe files.

How does this approach solve the issue of the network bottleneck?
It doesn't!
The entire application can now be broken down into three servers, a web server for showing the application to the user on browser, an application server to implement business logic on the data and a database server to actually store the database.

Even in this case, huge amounts of data transfer will happen between application and database servers to implement business logic and run queries and this happens across the network. However the application server can limit the web server to only receive and send data that's required.
We cannot dictate where the client will be operating from so we have no control over the web server but we can dictate where the application and database servers will be. So we'll try to increase their proximity so that they are as close as possible in order to ensure very little latency in the data transfers.
It's even possible that they both are running on the same physical server, on the same LAN, etc. So even though there's three tiers in the logical design, the second and third tier get a bit coupled while at the implementation level.

Each of these three servers are **independently deployable units** and in such a case, we can determine where we want to deploy each of these units, we can even deploy them on the same machine, which is what we do when we test out our web-app projects.

What are the new features that this architecture provides apart from fixing the existing problems?
The clients can once again go back to being dumb terminals (almost), they still need some processing power but only to display the web-app on a browser. That's literally all that's required. The modern name for dumb terminals is **thin clients.** They will have some memory (sir gave an example with 512MB RAM) for processing but it is very minimal as can be seen. Modern Android TVs are also thin clients, which is also why devices like Fire Stick by Amazon are possible, where plugging in a USB stick to a TV can convert it into a smart TV.

Is this architecture scalable?
YES!
You can make the whole architecture distributed now.
You can add more and more web, application as well as database servers to get a "swarm"/"cluster" of each type of server.
All the servers of the same type will be connected to a load balancer which will determine to which server the given request needs to be sent to depending on the current load.

This makes our architecture horizontally scalable which is a huge, huge benefit.

BUT CAN WE DO BETTERR??????????
We again try to apply the principle of cohesion and separation of concerns on the 3-tier architecture.

4-Tier Architecture

We said in 3-Tier architecture that the clients might as well be dumb terminals, i.e. thin clients. But in reality, clients are running browsers on PCs which are becoming more powerful by the day.
So why not utilize their processing capacity as well?
That is why the 4th tier was introduced which is the Client Tier and splits apart from the Presentation Tier.

The Client tier does all the processing that's required at the client side like displaying relevant form data, dialog boxes, form validation and all that jazz.
Previously there was no facility for any of this as the browser just did the task of displaying HTML code.

So the Client tier handles client-side processing and the Presentation tier continues with the task of displaying the webpage/web-app.

To implement the client-side processing we got new tools like JavaScript and AJAX (Asynchronous JavaScript & XML).

[Note: In Java, the task of converting business logic from application layer and sending it to the presentation layer in pure HTML is what a servlet does]

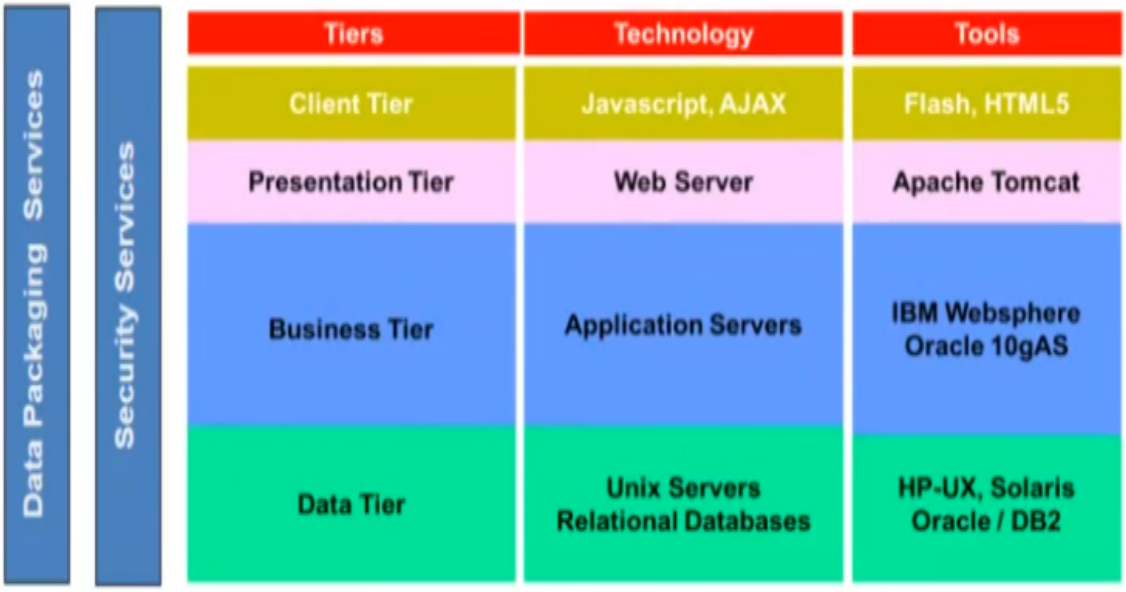
Tiers	Technology	Tools
Client Tier	Javascript, AJAX	Flash, HTML5
Presentation Tier	Web Server	Apache Tomcat
Business Tier	Application Servers	IBM Websphere Oracle 10gAS
Data Tier	Unix Servers Relational Databases	HP-UX, Solaris Oracle / DB2

And so-on for N-Tier Architectures!

Adding Open Layers

All of the layers in these discussed architectures were closed layers.
Now we can think of adding open layers, an example is as follows,

- We can add a new tier called Security Services which the Presentation tier can use for authentication services.
- The Data tier can use the encryption keys provided by the Security Services tier to send data to the Business tier.



You can see how we add the Open Layers to the left so that they are pictorially accessible by all the Layers in the architecture.

Impact on Deployment Architecture

- Greater the number of tiers, greater the deployment flexibility.
- Provides multiple deployment options.
- N software tiers may be deployed on **UPTO N** hardware tiers (because we can deploy different server types on the same physical server as well).
- Provides the option to scale horizontally.

Vertical Scaling vs Horizontal Scaling

The point of scaling comes up when you increase the capacity of traffic that your web-app can handle. You can handle it via vertical scaling or horizontal scaling.

Vertical Scaling
Adding more resources to the same processing unit, mostly a server.
For example, increasing the RAM and storage space of the server, upgrading the processor, etc.
As you can imagine there is a ceiling to how far you can go with vertical scaling because you cannot keep adding RAM infinitely, same for hard disk, physical space required by the server itself.
The only advantage is that it’s easy to do, especially compared to horizontal scaling.

Horizontal Scaling
Adding more processing units to a cluster of processing units.

For example, adding more servers to a collection of servers and having them behave as one cohesive unit, i.e. distributed computing.
Here we need load balancers and a lot of compatibility stuff needs to be figured out.
But the benefit that it provides is tremendous as we have no ceiling on any computing resource capacity as we can keep on adding servers to the cluster till the end of time as per our web app's requirements.

Do note that cost and availability doesn't factor in as a differentiating factor between horizontal and vertical scaling as both techniques will be affected severely by these factors.

Review of Layered Architecture

Pros:
Ease of engineering (dev, test, deploy).
Better control on quality attributes.
Eases project management.

Cons:
Too many layers leads to effort fragmentation.
Maybe overkill for less complex solutions.
Increases monitoring and troubleshooting complexity.
More moving parts leads to more points of failure.

07/12/21

Service Oriented Architecture (SOA)

Business Process Concepts

[Terms that will be useful to understand service oriented architecture]
Business Process: A formally defined method for achieving a specific goal.
Task: Steps involved in a specific business process.
Process Input: Information provided when a process is triggered.
Process Output: Information generated after a process is completed.
Manual Process: Process in which all tasks are exclusively done by humans.
Automated Process: Process that is made up of tasks that are fully automated.
Hybrid Process: Process where some tasks are manual and some are automated.

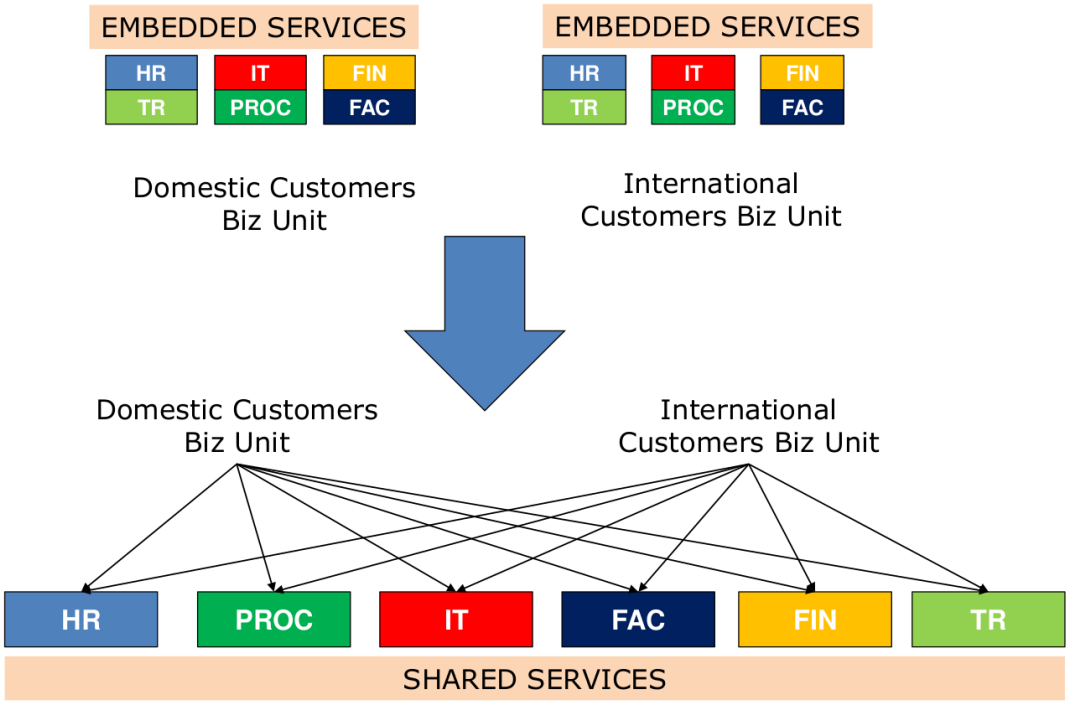
Why do we need SOA Models?

To implement something that's like a Shared Service Model.

Why do we need Shared Service Models?
Otherwise the alternative is the **Embedded Services Model**.

Embedded Services Model

Consider the following scenario,



Here we are a business and we have two business units in it, a domestic customers business unit and an international customers business unit.
Each of these units require their own HR team, IT department, Finance department (FIN), Training (TR), procurement (PROC), facilities (FAC), etc.
These are **services** which are embedded in each business unit which is why they're called Embedded Services.

Note: In this entire discussion, Services are considered to be stand-alone.

Advantages:

- Easy communication within a unit.
- Can give dedicated service to that unit, so experiences can be more customized and tailor-made for specific use-cases which can make the business very efficient.

Disadvantages:

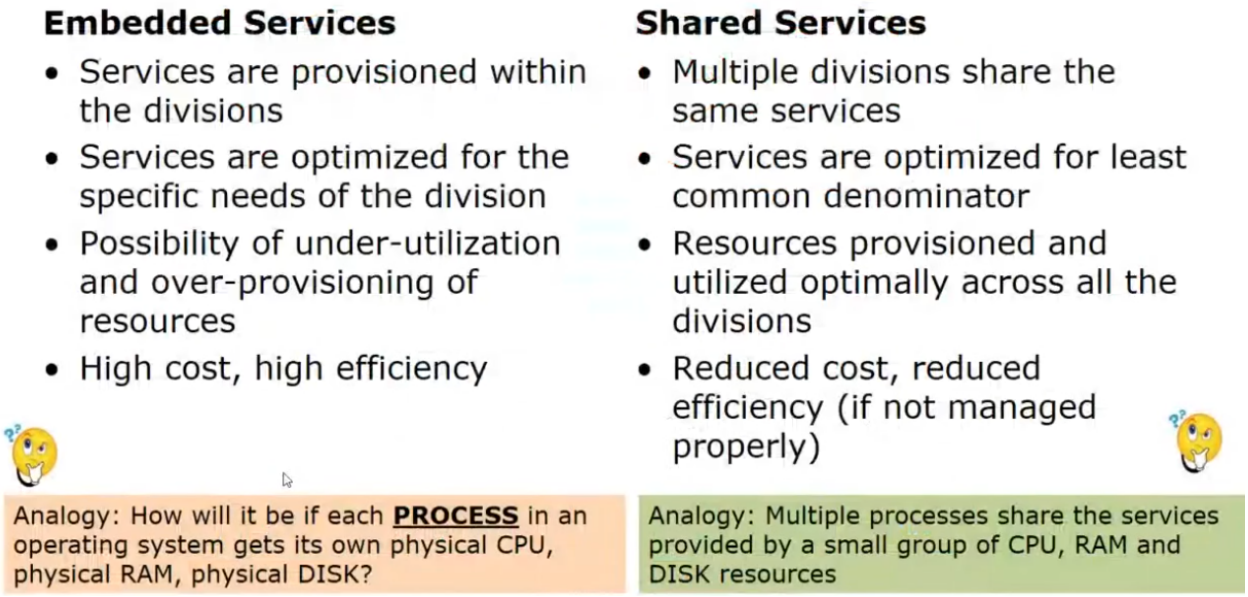
- Optimum utilization of teams/services becomes difficult.
- You can optimize to a certain extent by allocating more/less resources to a particular unit depending on business demand but you cannot dynamically reallocate services from one unit to the other, otherwise the whole dedicated aspect is lost.

- So if one unit is such that it is almost free for most of the day except for some few peak hours, then there is no facility of shifting the team from one unit to the other for the remaining less peak hours to optimize working hours further. There are several other situations where this happens as well.

Shared Services Model

As you can see in the previous image, in this model we will share the services between the two business units and allocation will happen as per requirement.
So the system is more dynamic and optimum utilization of services can be achieved.

Shared Services vs Embedded Services Models



SOA Model

SOA is an architecture style for enterprise IT systems.
Inspired by the shared services model.
At a conceptual level, WHAT needs to be done in a business process remains the same.
However in the logical and physical level, HOW it needs to be done will vary in SOA.

SOA Characteristics

1. Functionality is divided into **composable services** and not single use local functions.
    - In this context, local functions mean something which has some dependencies and prerequisites in terms of what needs to be executed before the local function is executed and services as mentioned before are stand-alone, so they should have no such dependencies.

- Composable services refers to the fact that a service which is stand-alone can be combined with other services to comprise a bigger service which is also stand-alone.
- For example, to implement “Shopping Cart as a Service” as a general purpose solution, we could be looking at various services like “Payment as a Service”, “Checkout as a Service”, etc.

2. Loose coupling between service provider and service consumer.
   - That is, there is an intermediary interface/service between the service provider and service consumer.
   - Can map this to the analogy of encapsulation in OOPS and what all benefits we get via encapsulation, we also get by making something loosely coupled.
   - So we can change the service provider without having to change anything in the service consumer similar to how encapsulation helps provide an API/abstraction layer that can mask how the method actually runs so that the implementation of such methods can be changed anytime without it impacting the client’s workflow.

3. Serves as building blocks for more complex services (using composable services).
4. Makes business processes more dynamic (with the help of loose coupling).

Service Concepts

[Again a bunch of jargon to know about]

Service Capability: Every service is capable of carrying out a certain FUNCTION.
Service Contract: Capability is formally published in the form of a CONTRACT.
Service Discovery: Service consumers should be able to find out about all the available services from a CATALOG and their capabilities via contracts.
Service Invocation: Compose a set of capabilities to carry out TASKS.

To map this to something more concrete,
Function prototypes in programming languages are like Service Contracts because it tells you the flow of communication with the function, i.e. what input it requires and what output it will give you.
Similar logic can be applied to a REST API endpoint, it defines @Consumes, i.e. what input is required to be sent to it and @Produces, i.e. what output it sends back.

Service Invocation involves actually using the services to carry out tasks which are part of a business process.

Key Elements of SOA

1. Open Standards
SOA involves things like Service Contracts and the like which involve publicly declaring information/data. This inherently demands that Open Standards have to be followed.

2. Integration Points
Because the entire business process is implemented via services, there has to be integration points where other entities can interface with your service and use its functionalities.

3. Virtualization
Location, Platform, Service Provider.
The service provider and consumer need not be concerned with the location of the service, i.e. where it is invoked, where it is provided from, etc, also with the platform being used by the service provider and consumer.
The service consumer should also not be concerned with who the service provider is, e.g, mobile phones are not concerned with who the SIM card provider is.

4. Automation
Business Process vs Workflow, they more or less refer to the same thing but in different contexts. This got a bit too technical so I'm leaving it as it is :(

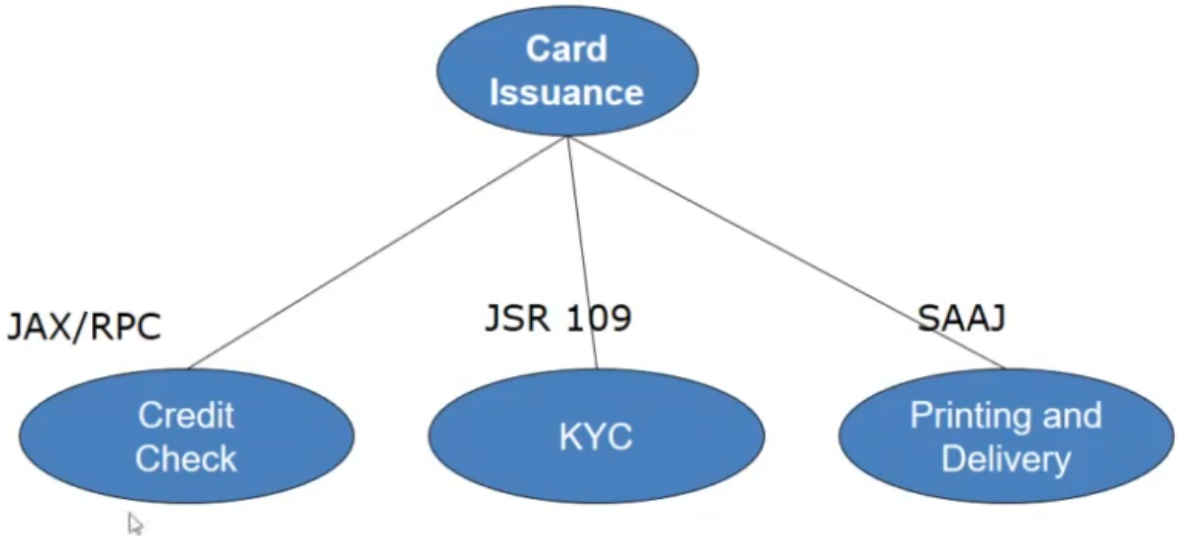
5. Orchestration
The controller for each business process **invokes** services as part of the implementing process.
Analogous to a conductor conducting an orchestra, no musician plays unless asked to by the conductor.

6. Choreography
The controller for each business process **announces** tasks that need to be done as part of the implementation process.
Services pick up the tasks and do what is needed.
Analogous to dancers doing their own moves once music starts playing.
This is a trigger and event driven style.

Orchestration and Choreography are two ways to implement the same aspect.

Composable Services with Open Standards

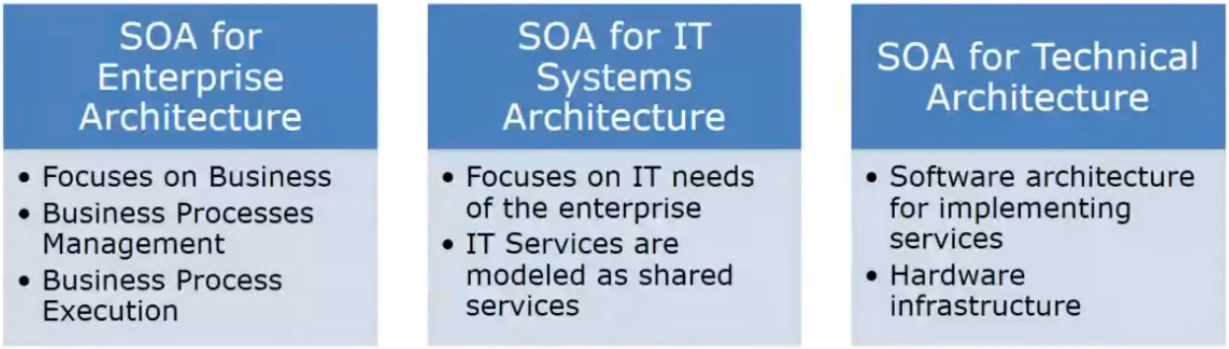
Example of Card Issuance service with the help of various services each of which have their own Open Standards as described here, JAX/RPC for Credit Check, etc.



What does Closed Standards mean?
Proprietary Standards which are vendor specific.

You should use Open Standards when developing a service because you will lose the feature of Composable services in that case. Because if you make a proprietary service then it is not possible for others to integrate via open standards on your service.

Flavours of SOA



Complexity increases as we go from left to right.

Architecture Principles

Lot of stuff was in this like,
Service Loose Coupling => Minimizes dependencies
Service Abstraction => Minimize the availability of meta information
Service Composability => Maximizes composability
Standardized Service Contract => Implement a standardized contract

Service Reusability => Implement generic and reusable logic and contract
Service Autonomy => Implement independent functional boundary and runtime environment

Service Statelessness

The service provider need not remember information on who is invoking the service.
That is, the server/service provider doesn't remember anything across invocations.

Implementing a Shopping Card Service - Stateful vs Stateless
The service has two functions,
addToCard(), removeFromCart()

Stateful
c = new ShoppingCart();
c.addToCard(Item i);

The ShoppingCart is stored in object c which means it must have an ArrayList type set up to store the added items.
This is a state that is being stored by the object and exactly why this implementation is stateful.

Stateless
The following code is implemented at the client-side,
Service s = new ShoppingCartService();
Cart c = new ShoppingCart();
c = s.addToCart(Cart c, Item i);

Because this is client-side, the Client itself stores its own cart.
So the server never actually remembers what is in the cart, it just provides access to the ShoppingCartService interface which can be used by the client's cart to add or remove items.

So the client does actually store a state but the whole concept of stateless is referring to the service provider, i.e. the server in this case not remembering any information about the service consumer, i.e. the client in this case.

Service Discoverability

Services are supplemented with communicative metadata by which they can be effectively discovered and interpreted.
An example for this metadata can be as follows,
Name of Service: addToCart()
Name of parameters: ShoppingCart c, Item i
Return type of Service: ShoppingCart
etc.

If services are not discoverable, there can be wasted effort in implementing multiple embedded services for the same service.

SOA using Web Services

[Technical Architecture for SOA => Actually trying to implement at a technical level]
Web Service is an open-standards based protocol for implementing SOA over the web.
[Note that Web Service is the actual name of the standard, a proper noun :)]

Web Services provide an open-standard and machine-readable model for creating explicit implementation-independent descriptions of service interfaces.
Web Services provide communication mechanisms that are location-transparent and interoperable.

Key standards,
TCP/IP, XML, HTTP
WSDL (Web Services Description Language, “Wisdel”), SOAP (SOA Protocol), UDDI (Universal Description, Discovery and Integration)

WSDL => Contract
SOAP => Communication Interface
UDDI => Discovery

The main disadvantage using SOAP is that it puts a lot of load on the communication interface as huge amounts of XML data needs to be transferred in order to perform any kind of communication.

SOA using REST

REST stands for **RE**presentational **S**tate **T**ransfer.
Lightweight alternative to SOAP-based Web Services.

Key behavioural characteristics,
- Stateless
- Idempotent or Non-Idempotent
- RO/RW

Key protocols
- HTTP based methods such as GET, POST, PUT, etc.
- JAX-RS for JAVA implementations

Summary

- SOA can be implemented at various levels, enterprise level, IT system level, service level.
- SOA can be implemented via SOAP or REST, REST is more preferred and SOAP is slowly being phased out and will most likely become for legacy purposes only.

10/12/21

Enterprise Level Testing

Classical SDLC View of Testing

1. Unit Testing
Answer the WH questions to this,
What? Testing functionality of a function by a developer in a constrained environment.
Who? By developers
When? At the beginning of testing
Where? In the development environment
How? By writing test-cases as another program => easy to automate

2. System Testing
Answer the WH questions to this,
What? Testing the entire system together by considering all components, i.e. units as one whole unit.
When? All the functionalities have been developed and units have been tested.
How? Elaborate process that can involve a lot of setup and planning and can involve both automation as well as manual work.
Where? In the testing environment, so a separate environment is created for it.
Who? By the testing team.

3. UAT (User Acceptance Testing)
What? Same idea as System testing
Who? The users themselves.
The tests may not be as involved and in depth as done by the testers but it holds utmost importance regardless.
When? Usually done as the final testing phase once the unit and system testing checks have been passed.
Where? In the user's environment.

These types of testing will be valid for any kind of systems that have been discussed in this course.

But for enterprise software there are some extra considerations with regards to testing

1. Integration Testing
2. Non-Functional Testing
3. Regression Testing

Integration Testing

- Integrating external systems into the system and testing whether the integration is successful or not.
- Example, GST is a compliance organization for the system and we have to upload GST reports at EOD, so we have to integrate the GST APIs with our system and testing how well this integration works is part of integration testing.

So you can have system testing without integration testing, in this case, the external systems part is filled with dummy data, i.e. **stub functions** in order to avoid having to integrate the external modules but still test the entire system without it.

Stub functions => Empty functions, function A() { }

To understand more about integration testing, we need to know about the various kinds of interfaces available,

Application Interface

- Exists between modules and components within the same software application.
- Communication across this interface is synchronous but can be asynchronous as well.
- The interface is usually accessed locally but can be exposed remotely too (local access means that the same process is accessing the interface, so directly we can assign the appropriate interface object, in case of remote, it means the access can happen from anywhere, not even the same machine as well), so the testing must account for all such possibilities.
- This is the most natural form of interface and is generally quite robust.
- By saying they are robust, we mean to say that they generally don't break, as in, making a particular call or doing a particular action just results in the interface to stop working.

Data Interface

- Collaboration without function invocation.
- This interface can exist between components belonging to two different applications so direct data transfer is typically not done between the two applications and instead it goes through this interface.

- Data interfaces are usually asynchronous in nature, e.g. file uploads.
- Example, uploading the day's forex rates at the beginning of the day to the relevant forex rates analyzer/firm, etc. so the only transfer happening is data and so a data interface is involved.
- Also referred to as **Batch Interface** because data is sent in batches as well as EOD jobs.
- The two components are not concerned with the internal working/technology/location of each other because the only thing that matters is the transfer of data.

This is less robust as compared to Application Interfaces because it can easily happen that one of the two endpoints in the communication are down and all hell breaks loose in that case depending on how important the data being transferred is.
Whereas in Application Interfaces if we have a situation where local access happens then there is no way that the communication fails because they are on the same process literally.

Messaging Interface

- Messaging interface helps two components interact via message queues.
- This interface is like a bus that connects n number of components/devices on it.
- Message queues serve as a common repository that both senders and receivers have access to.
- If component A wants to send a message to component B, then it will place a message on this interface/queue.
- As can be observed, in the Messaging Interface there are two entities, Producer that sends the message to the queue/interface and Consumer that receives the message from the queue and processes it.
- In Data Interfaces, the data being transferred/payload can only be a file whereas here it can be anything, a data structure like ArrayList or Map or whatever.
- This is clearly an asynchronous communication mechanism ("fire and forget").
- An easy example is the concept of Emails.
- More jargon: **Publisher/Subscriber Model (Pub-Sub)**, the publisher keeps publishing the content like email and subscriber keeps consuming the content of publishers to which they are subscribed to.

Guarantees robustness.
Once a message has been enqueued in the queue then it is guaranteed to be sent to the destination and will not get lost.

Callback Interface

Suppose A needs some services from B.
A registers itself with B.
B makes calls to A instead of the other way around.
Example, in GUI programming say you have a GUI manager and GUI code,

So you tell a GUI manager "call myaction when this button is pressed" and the GUI manager calls the action when the button is pressed.
So we are registering the fact that method “myaction” is to be run when this button is clicked and so on for other events as well.

Other examples include SAX Parsers and any MVC frameworks.

This is also referred to as the Hollywood Principle, “You don’t call us, we’ll call you”

Note: Event Handlers are a bit more complex version of this.

Callback Interfaces are one way of implementing event based systems, you can also think of using Message Queues for the same task as well (in fact Event Handlers do actually use both, Callback and Messaging Interfaces in a way).

Infrastructure Interface

This comes into the picture when two or more components try to interact with each other where one component belongs to an application and another belongs to the underlying software infrastructure.

Examples, J2EE Containers, .NET Framework.

This kind of interface has been phased out and has become mostly legacy code that has to be maintained, so read up on it if you are interested.

Interface ErrorsL Types

When it comes to software, why is the sum of parts, i.e. unit tested modules, not equal to the whole, i.e. the full system?

Construction Error

Programmers may overlook interface specifications while coding.
Example, a C header file may ask “int” and the calling function may give “float”.

Because the programmers do not have access to the exact interface environment to verify and validate during unit testing, they will just place stub functions in places where the integration units would come in later and so unit testing would not show any issues and even system testing would work properly whereas integration testing might fail.

Inadequate Functionality

Caused by implicit assumptions made by either/both, the provider and consumer of an interface.
In this case, the requirements document does not sufficiently document the assumptions.
And when do developers ever read the requirements document, am I right or am I right ;)

This is why such problems can also happen.

Location of Functionality

Disagreement on where, i.e. which module, a specific functionality should be implemented.
Example, where should the validation be? Client-side JS or server side?

May also be caused by assumptions that do not hold.

Changing/Adding Functionality

Changing a module or functionality without proper impact analysis.
Not doing proper impact analysis of the changed functionality/module is a big issue because you can end up slipping in a completely new functionality in the guise of a change.

Incorrect Use and Misuse of an Interface

Incorrect use generally occurs at the program level.
Example,

```
char *s1 = “Hello”, *s2;
strcpy(s2, s1);
printf(“%s”, s2);
```

Output:
Segmentation fault (core dumped)

Why?
Because we never allocated any memory for s2.
If we had done something like, char s2[10];
And executed the same code, then it would have run properly.
Or even, s2 = (char*) malloc((strlen(s1) + 1) * sizeof(char));
Then also it would run properly.
Remember such silly things can mess you up in interviews.

So here, we did an incorrect use of the strcpy() function by not understanding what its interface (prototype) asked for.

Rigid Data Interfaces

The data interface may be very brittle, i.e. may not be able to cope with even very minor variations in the data stream, e.g. 1.0 vs 1 as input.

The design may not have foreseen all the possible uses of a specific data integration.

For example you can have a text file based database in which you are storing the data entries and your interface is programmed such that each column (field) in a row (line) is determined by which character number it occurs at, e.g. customer id starts at 5th character in a line, etc.
So in such a situation, 1.0 vs a 1 makes a huge difference and messes up the entire logic.
And this implementation would be a very brittle/rigid data interface.
To make it more robust, we can separate each field by a delimiter like commas and now it doesn't matter how many characters each field's value contains.

Inadequate Error Processing

A called module may return an error code that the calling module may not be ready to cope with or may be processing it incorrectly.

Inadequate Post Processing

Failure to clean-up after using the interface.

Example,

- Not deleting temp files and ending up reusing the same connection to the files for processing resulting in duplication of processing.
- Not releasing connections after usage leading to connection pool contention, i.e. connecting to databases via JDBC but not releasing them and soon the number of connections to the same database will be tremendous resulting in very poor performance as JDBC will have a limited pool of connections to hand out and there will be competition/contention amongst the requesting objects/services.

Note: Connection pool is the initial connection capacity that the JDBC instance will be instantiated with to avoid delays during runtime, startup will be a bit slower but once that's done, its smooth sailing from there as closed connections will be used for connecting to new requests. This still doesn't solve the problem discussed before because the client has to first close the damn connection.

Performance Problems

Most performance problems surface only during integration testing.
Primarily caused by defects in upstream activities,

- Not factoring in infrastructure limitations/constraints.
- Assuming infinite bandwidth/CPU/disk space.

Non-Functional Testing

Non-Functional Requirements are also often called **Architecture Quality Attributes** or just Quality Attributes because the term Non-Functional is absolutely stupid.
It is called Non-Functional because testing is done not on the functionality of the system but instead on the architecture and things like performance, scalability, security, etc. which are not exactly required for the system to function. It is still a dumb name though :(

- Involves product quality factors
- Tests product behaviour with respect to user experience
- Huge data collected and analyzed
- Focus is less on defect detection in code as opposed to defect detection in architecture/design
- Requires sound product, design, architecture, analytical skills
- Failures usually attributed to design and architecture
- Typically done only as part of system testing

Scalability Testing

Scalability refers to, “The ability of a system to do more **given more resources**”.
Linear Scalability => If you have double the normal load, then you require double the normal resources.
If the system is not scalable, then Linear Scalability cannot be maintained.

Ofcourse in the industry, situation is such that we even require sub-linear systems to be truly scalable :(

Objectives of Scalability Testing:

Find out the capability of the product to grow to the increasing needs, i.e. scale.
Identify the maximum capability of a given configuration.
Does NOT end when the requirements are met! And instead the testing continues till the requirements are met.

Scalability Issue Symptoms

Simply throwing more resources at a system may not increase its scalability.
For example, your system can have a memory leak (used memory is not being freed up or is being held down for no reason), then overtime the innocuous memory leak can build up to consume huge amounts of RAM and no matter how much RAM size you increase, it will never be enough because this memory leak was never caught.

So a lot of work has to be done to tinker with the system and figure out what can mess up the scalability parameters.

Stress Testing

Objectives:

Evaluate the system beyond the limits of the specified requirements or resources.
To find out how the product degrades under extreme “unplanned” conditions.
It generally deals with external environment variables.

Scalability Testing finds out the maximum capability of the system, whereas Stress Testing tries to figure out what happens if you exceed this capability.

It is preferred to have “**graceful degradation**” in such scenarios, where if the capacity is exceeded, the system’s performance degrades but it is such that it doesn’t take in any new tasks and properly finishes all the existing tasks, so some amount of damage control is there.

Key Outcomes:

When is the stress point reached beyond which failures show up?
We are generally not interested in tracking specific parameters like in Scalability Testing. Instead the aim is to look at the system as a whole.

How quickly does the system recover once the stress is removed?
Is recovery even feasible and lots of good stuff like that is to be checked.

The scope of Stress testing is boundless and can involve many, many things. Scalability testing is also similar but not as vast as Stress testing.

Triggers for Performance Testing

Proactive

Done as part of the planned testing activity within the SDLC.
Generally guided by some high level performance requirements (may be vague).

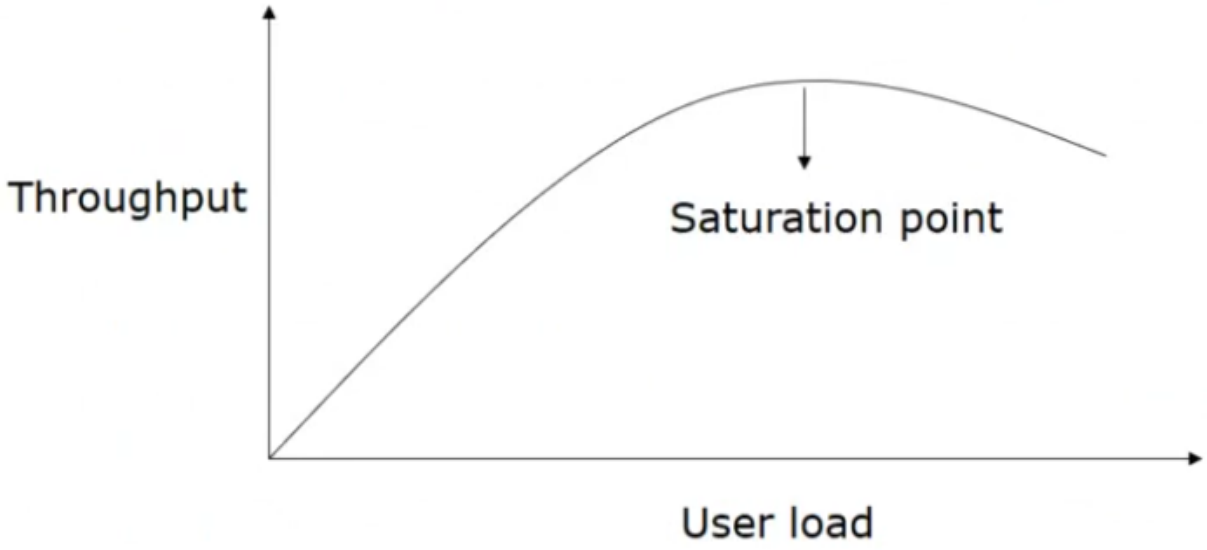
Reactive

Specific performance issues arise.
Done only if needed.

Performance Parameters

There were many here but sir focused on Throughput only,

1. Throughput Trend



Performance and Throughput might not always be correlated.
Can you have high throughput and low performance?

Other types of testing

Alpha/Beta Testing
The finished/almost finished product is tested by developers/QA testers => Alpha Testing => “Eat your own dog food” ;)
The finished/almost finished product is tested by end-users => Beta Testing (type of UAT but only for consumer systems and not for enterprise systems).

Testing for acquiring Certificate of Compliance
Deployment testing

And that’s the end of the course. Very abrupt ending but it is what it is!