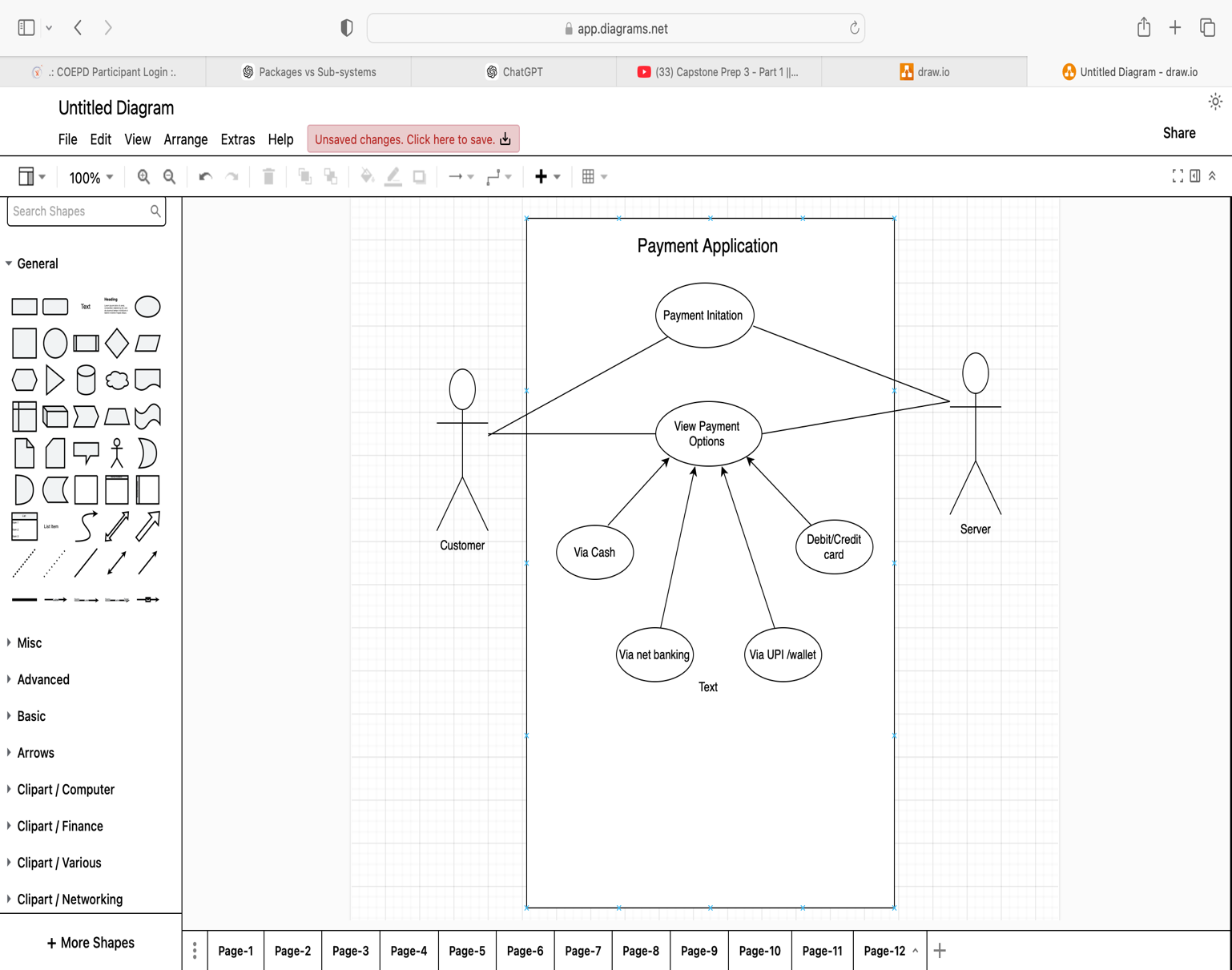
**Capstone Project- 3**

A customer can make a payment either by Card or by Wallet or by Cash or by Net banking.

Q1) Draw a Use Case Diagram.

**Answer-1**

****

Q2) Derive Boundary Classes, Controller classes, Entity Classes.

**Answer-2**

 **Boundary Classes (View/UI Classes)**

* Handle user interaction (UI/UX).
* Act as an interface between the system and the external actors.
* Example: PaymentOptionBoundary ; CardPaymentBoundary

 **Control (Controller) Classes**

**Acts as intermediaries between boundary and entity classes**

* Manage the flow of data between boundary and entity classes.
* Implement business logic and coordinate interactions.
* Example: Pay

 **Entity Classes (Model Classes)**

* Represent business objects with attributes and behaviour.
* Typically map to database tables.

**Answer-3**

### **Placing the Payment Classes in Three-Tier Architecture**

1. **Presentation Layer (UI Layer)**

* This layer interacts with users (customers).
* It collects payment details and sends them to the service layer.
* Example **Classes:**

PaymentUI (Handles user input for selecting payment mode)

PaymentForm (Takes card details, wallet selection, etc.

1. **Business Logic Layer (Service Layer)**

 Processes payment and applies business logic.

Determines payment method (Card, Wallet, Cash, or Net Banking).

Example **Classes**:

PaymentService (Processes payments based on the chosen method)

CardPayment (Handles card transactions)

WalletPayment (Processes wallet transactions)

CashPayment (Confirms cash payments)

NetBankingPayment (Handles net banking payments)

1. **Data Access Layer (Persistence Layer)**

* Manages database operations like storing payment records.
* Example **Classes**:
  + PaymentDAO (Handles database transactions)
  + TransactionRecord (Stores payment details)

**Answer-4**

A **Domain Model** represents real-world entities and their relationships in a specific domain. For a scenario where a **customer makes a payment through net banking**, the domain model includes key entities like **Customer, Bank, Payment Gateway, Merchant, and Transaction**.

Entities in the Domain Model

**Customer**

* Attributes: Customer ID, Name, Email, Phone Number, Bank Account Details
* Relationships: Initiates a **Transaction**, linked to a **Bank Account**

**Bank Account**

* Attributes: Account Number, Bank Name, Balance, IFSC Code
* Relationships: Belongs to a **Customer**, used for a **Transaction**

**Payment Gateway**

* Attributes: Gateway ID, Provider Name, Service Fee
* Relationships: Acts as an **intermediary** between the **Bank** and **Merchant**

**Bank**

* Attributes: Bank ID, Bank Name, SWIFT Code, Net Banking System
* Relationships: Processes **Transaction**, linked to **Bank Accounts**

**Merchant**

* Attributes: Merchant ID, Business Name, Account Details
* Relationships: Receives **Payment** from the **Customer** via **Payment Gateway**

**Transaction**

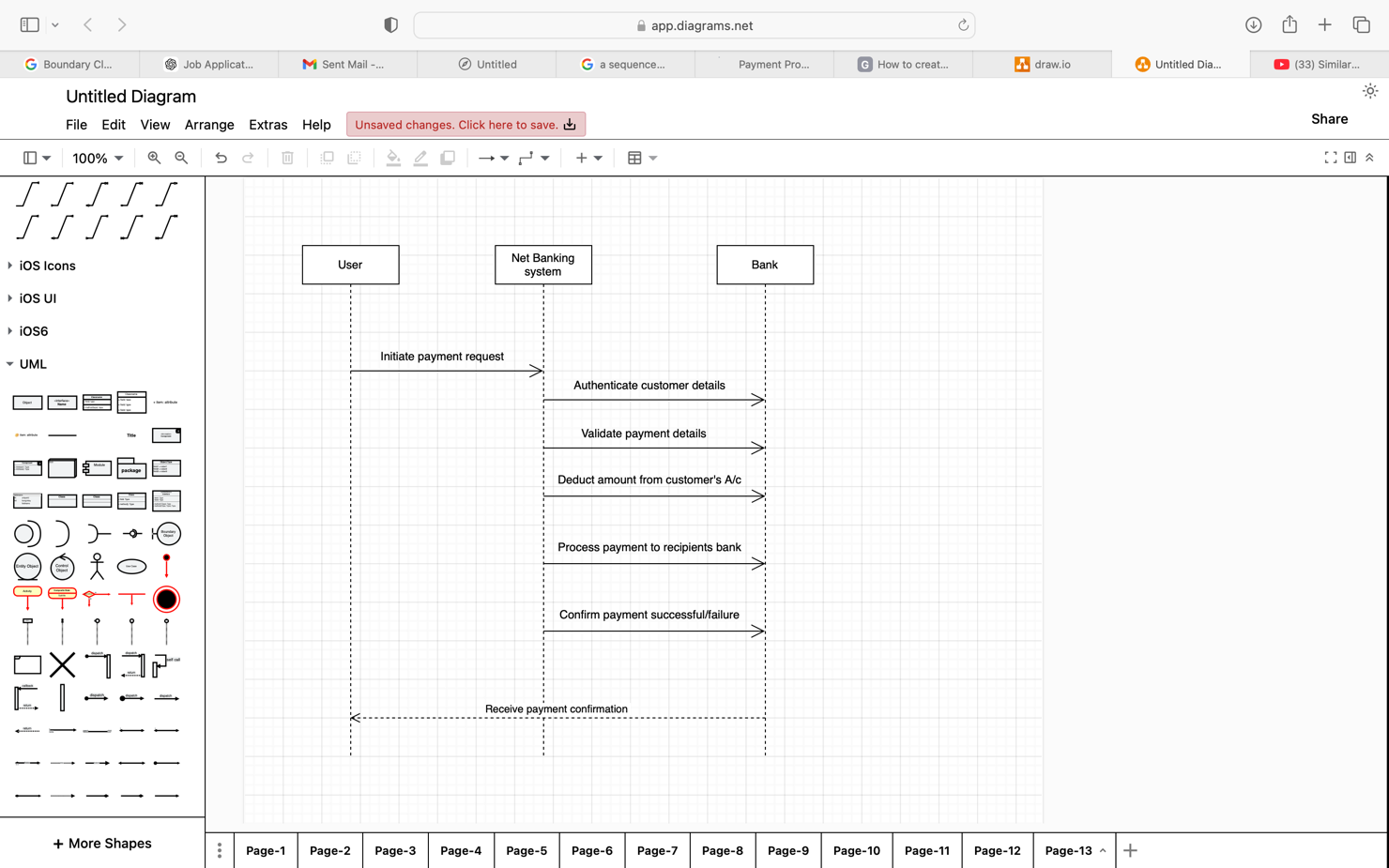
* Attributes: Transaction ID, Amount, Status (Pending, Successful, Failed), Timestamp
* Relationships: Initiated by **Customer**, processed via **Bank** and **Payment Gateway**, received by **Merchant**

### **Workflow Representation**

1. **Customer selects net banking** as the payment method.
2. **Payment request** is sent to the **Payment Gateway**.
3. **Payment Gateway redirects** the customer to the **Bank’s net banking portal**.
4. **Customer logs in**, selects an account, and **confirms payment**.
5. **Bank verifies** the transaction and **debits the amount**.
6. **Payment Gateway receives confirmation** and **notifies the Merchant**.
7. **Merchant acknowledges the payment**, and **the transaction is completed**.

**Answer-5 : Payment done by Customer Net Banking**

Sequence Diagram



UML Representation

A **class diagram** for the domain model would show:

* **Customer → initiates → Transaction**
* **Transaction → processed via → Payment Gateway**
* **Payment Gateway → interacts with → Bank**
* **Bank → transfers funds to → Merchant**

**Answer-6**

A **conceptual model** provides a high-level abstraction of entities, their attributes, and relationships in a system. In this case, a **customer** can make a payment using **Card, Wallet, Cash, or Net Banking**.

### **Key Entities and Their Relationships**

1. **Customer**
   * Attributes: Customer ID, Name, Email, Phone
   * Relationships: Initiates a **Transaction** and chooses a **Payment Method**
2. **Transaction**
   * Attributes: Transaction ID, Amount, Status (Pending, Successful, Failed), Timestamp
   * Relationships: Linked to one **Payment Method**, belongs to a **Customer**
3. **Payment Method (Abstract Entity)**
   * Attributes: Payment Method ID, Type (Card, Wallet, Cash, Net Banking)
   * Relationships: Specialized into **Card, Wallet, Cash, and Net Banking**
4. **Card Payment (Subclass of Payment Method)**
   * Attributes: Card Number, Expiry Date, CVV, Card Type (Debit/Credit)
   * Relationships: Linked to **Bank**
5. **Wallet Payment (Subclass of Payment Method)**
   * Attributes: Wallet ID, Provider Name (PayPal, Google Pay, etc.), Balance
   * Relationships: Linked to **Customer Account**
6. **Cash Payment (Subclass of Payment Method)**
   * Attributes: Receipt ID, Payment Location
   * Relationships: No external linkage, used for **offline transactions**
7. **Net Banking Payment (Subclass of Payment Method)**
   * Attributes: Bank Name, Account Number, IFSC Code
   * Relationships: Linked to **Bank**
8. **Bank**
   * Attributes: Bank ID, Bank Name, SWIFT Code
   * Relationships: Processes **Net Banking and Card Payments**

**Conceptual diagram representation**

Customer [1] ---- (initiates) ----> [\*] Transaction

Transaction [1] ---- (uses) ----> [1] Payment Method

Payment Method {Abstract} ----> (Specialized into) ----> Card, Wallet, Cash, Net Banking

Card [1] ---- (processed via) ----> [1] Bank

Net Banking [1] ---- (processed via) ----> [1] Bank

Wallet [1] ---- (linked to) ----> [1] Customer

Cash [1] ---- (used for) ----> [1] Offline Transaction

**Answer-7**

What is MVC architecture? Explain MVC rules to derive classes from use case diagram and

guidelines to place classes in 3-tier architecture.

MVC (Model-View-Controller) is a **software design pattern** used for developing applications by separating concerns into three interconnected components:

1. **Model (M) – Business Logic & Data Management**
   1. Represents the application's data and business rules.
   2. Interacts with the database and updates based on user actions.
   3. Example: Customer, Transaction, PaymentMethod classes.
2. **View (V) – User Interface (UI)**
   1. Displays information to users.
   2. Does not contain business logic; only presentation logic.
   3. Example: Web pages, mobile screens, reports.
3. **Controller (C) – Request Handling & Coordination**
   1. Handles user input and updates the Model/View accordingly.
   2. Acts as a middle layer between View and Model.
   3. Example: PaymentController, UserController.

### **Rules to Derive Classes from a Use Case Diagram in MVC**

1. **Identify Entities (Model Layer)**
   * From **nouns** in use cases (e.g., "Customer," "Order," "Transaction").
   * These become **Model classes** responsible for business logic and data.
2. **Determine UI Components (View Layer)**
   * From **use case scenarios related to UI interactions** (e.g., "Display Order Summary," "Show Payment Options").
   * These map to **View classes** (e.g., OrderView, PaymentView).
3. **Identify Controllers (Controller Layer)**
   * From **verbs or actions in use cases** (e.g., "Process Payment," "Validate User").
   * These translate to **Controller classes** that manage user inputs and direct flow.
4. **Define Relationships Between Model, View, and Controller**
   * Model classes interact with databases.
   * View classes retrieve and display model data.
   * Controllers update models and decide which view to render.

### **Guidelines for Placing Classes in 3-Tier Architecture**

The **3-Tier Architecture** divides an application into **Presentation, Business Logic, and Data Layers**, mapped to MVC components as follows:

1. **Presentation Layer (View in MVC)**
   * Handles user interaction and UI rendering.
   * Includes **HTML, JavaScript, CSS, UI Components**.
   * Example: LoginPage.html, PaymentView.jsp, React Components.
2. **Business Logic Layer (Controller in MVC)**
   * Implements application logic, rules, and validation.
   * Includes **Service classes and Controllers**.
   * Example: UserController, TransactionService, PaymentProcessor.
3. **Data Access Layer (Model in MVC)**
   * Manages database operations and persistence.
   * Includes **Entity classes, Repositories, DAOs (Data Access Objects)**.
   * Example: CustomerDAO, PaymentRepository, Hibernate Entities.

|  |  |  |
| --- | --- | --- |
| **MVC Component** | **3-Tier Layer** | **Example Classes** |
| **Model** | Data Layer | Customer, Order, TransactionDAO |
| **View** | Presentation Layer | OrderView.jsp, PaymentUI.html |
| **Controller** | Business Logic Layer | PaymentController, OrderService |

**Answer-8**

### **Business Analyst (BA) Contributions in a Project Using the Waterfall Model**

In the **Waterfall Model**, project development follows a **linear and sequential approach**, moving through distinct phases. A **Business Analyst (BA)** plays a crucial role in each phase, ensuring clear requirements, smooth communication, and proper alignment between stakeholders and the development team.

### **BA Contributions in Each Stage of the Waterfall Model**

#### ****1. Requirement Gathering & Analysis****

**BA Responsibilities:**

* Interacts with stakeholders to **gather business needs and objectives**.
* Conducts **interviews, surveys, and workshops** to collect detailed requirements.
* Prepares **Business Requirement Document (BRD)** and **Functional Requirement Specification (FRS)**.
* Defines **Use Cases, User Stories, and Business Rules**.
* Helps in **requirement validation and approvals** from stakeholders.

#### ****System Design****

**BA Responsibilities:**

* Translates **business requirements into system specifications**.
* Works with architects and designers to ensure **functional flow is aligned**.
* Creates **process flow diagrams, wireframes, and mockups** for UI/UX.
* Ensures **non-functional requirements (performance, security, usability, etc.)** are addressed.

#### ****Implementation (Development)****

**BA Responsibilities:**

* Acts as a **bridge between developers and stakeholders**.
* Provides **clarifications** on requirements to the development team.
* Assists in refining **logical and functional flows** if issues arise.
* Ensures that **business needs are met during development**.

#### ****Testing (Verification & Validation)****

**BA Responsibilities:**

* Supports **QA team** in understanding requirements.
* Reviews **test cases** to ensure all scenarios are covered.
* Participates in **User Acceptance Testing (UAT)** with stakeholders.
* Identifies gaps between **expected vs. actual outcomes** and helps in resolving issues.

#### ****Deployment & Maintenance****

**BA Responsibilities:**

* Assists in **Go-Live planning** and ensures a smooth transition.
* Prepares **training materials and user guides** for end-users.
* Gathers feedback for **future improvements** or **change requests**.
* Supports **post-production issue resolution**.

|  |  |
| --- | --- |
| **Phase** | **BA Contributions** |
| Requirement Gathering | Collects, documents, and validates business needs. |
| System Design | Translates requirements into system specs, wireframes. |
| Implementation | Provides requirement clarifications to developers. |
| Testing | Assists QA, verifies system alignment with business goals. |
| Deployment & Maintenance | Supports training, collects feedback, manages change requests. |
|  |  |

**Q9.**

**Answer-9**

Conflict management is the process of handling disputes and disagreements in a constructive way to minimize negative impact and enhance collaboration. It involves strategies to address conflicts **effectively, fairly, and efficiently**, ensuring that all parties involved reach a resolution that aligns with their goals.

### **Thomas-Kilmann Conflict Management Model**

The **Thomas-Kilmann Conflict Mode Instrument (TKI)** is a widely used technique to manage conflicts based on two factors:

1. **Assertiveness** – The degree to which one **seeks to satisfy their own needs**.
2. **Cooperativeness** – The degree to which one **attempts to satisfy the needs of others**.

|  |  |  |
| --- | --- | --- |
| **Conflict Style** | **Description** | **When to Use** |
| **1. Competing (High Assertiveness, Low Cooperation)** | One party pursues their goals aggressively, often at the expense of others. | When quick decisions are needed (e.g., emergencies) or when enforcing unpopular but necessary rules. |
| **2. Collaborating (High Assertiveness, High Cooperation)** | Both parties work together to find a win-win solution that satisfies everyone. | When a creative or long-term solution is required and relationships matter. |
| **3. Compromising (Medium Assertiveness, Medium Cooperation)** | Both parties give up something to reach a mutually acceptable solution. | When a temporary or partial resolution is acceptable, or when time is limited. |
| **4. Avoiding (Low Assertiveness, Low Cooperation)** | One or both parties withdraw from the conflict instead of addressing it. | When the issue is trivial, or when emotions are high and a cooling-off period is needed. |
| **5. Accommodating (Low Assertiveness, High Cooperation)** | One party gives in to the other's demands to maintain harmony. | When maintaining relationships is more important than winning, or when the issue is more important to the other party. |

**Answer-10**

Project failure can occur due to various factors, including poor planning, lack of resources, or ineffective communication. Below are the key reasons why projects fail:

### **Poor Requirement Gathering & Scope Creep**

* **Unclear or incomplete requirements** lead to mismatched expectations.
* **Scope creep** occurs when new features are added without proper assessment.
* Lack of stakeholder involvement in defining the scope.

**Solution:** Conduct detailed requirement analysis, create a Scope Document, and get stakeholder approvals.

* Inadequate Planning & Estimation
* Poor project scheduling and unrealistic deadlines.
* Lack of proper risk assessment.
* Underestimating resource needs (time, budget, manpower).

**Solution:** Use project planning tools (Gantt charts, WBS), conduct risk analysis, and allocate buffer time

* Weak Leadership & Poor Decision-Making
* Lack of vision and leadership direction.
* Slow or ineffective decision-making.
* Failure to manage conflicts and team expectations.

**Solution:** Ensure experienced leadership, promote clear decision-making frameworks, and enhance conflict resolution.

* Lack of Communication & Collaboration
* Misalignment between stakeholders, teams, and clients.
* Poor documentation and lack of status updates.
* Information silos within teams.

**Solution:** Regular stand-up meetings, clear reporting structure, and collaboration tools (Slack, Jira, Trello).

### Inadequate Risk Management

Ignoring potential project risks (technical, financial, legal).  
No contingency planning.  
Failure to respond to unexpected challenges.

**Solution:** Conduct risk assessment early, create a risk register, and define mitigation strategies.

### Budget Overruns & Resource Mismanagement

Poor cost estimation leads to budget exhaustion.  
Lack of skilled team members or improper resource allocation.  
Uncontrolled spending without tracking financials.

**Solution:** Use cost-tracking tools, define budget limits, and ensure proper resource allocation.

### Technical Challenges & Poor Quality Deliverables

Selection of wrong technology or outdated tools.  
Poor software architecture leads to performance issues.  
Inadequate testing and QA processes.

**Solution:** Perform technical feasibility studies, ensure proper testing (unit, integration, UAT), and follow best coding practices.

### Resistance to Change & Lack of User Adoption

* Employees resist adopting new systems or workflows.  
   Lack of proper training and onboarding.  
  Poor change management strategies.

**Solution:** Implement change management frameworks (Kotter’s Model), provide training, and involve users in decision-making.

### External Factors & Market Changes

Sudden regulatory or legal changes.  
Market conditions shifting (economic downturns, competition).  
Vendor or third-party dependencies failing.

**Solution:** Stay updated on market trends, conduct regulatory compliance checks, and have backup vendors.

### **10.** Lack of Post-Implementation Support & Maintenance

No long-term maintenance or support plan.  
System issues arise post-launch but remain unaddressed.  
No feedback mechanism for continuous improvement.

**Solution:** Plan for maintenance, monitor system performance, and gather user feedback for enhancements.

Project failures often result from a combination of **poor planning, mismanagement, lack of communication, and technical risks**. Proactively addressing these challenges through structured project management frameworks (e.g., Agile, Waterfall) can increase project success rates.

**Answer-11**

A **Business Analyst (BA)** plays a crucial role in project success, but they often face various challenges that impact project timelines, quality, and stakeholder alignment. Below are the key challenges a BA may encounter in different project stages:

### **Unclear or Changing Requirements**

🔹 Stakeholders provide vague, incomplete, or conflicting requirements.  
🔹 Requirements change frequently, leading to **scope creep**.  
🔹 Business users may struggle to articulate their actual needs.

### **Stakeholder Conflicts & Misalignment**

🔹 Different stakeholders have **conflicting priorities** and expectations.  
🔹 Lack of proper engagement from decision-makers.  
🔹 Resistance from teams unwilling to adopt changes.

Communication Gaps & Misinterpretation

🔹 Misunderstandings between technical and business teams.  
🔹 Lack of documentation or improper requirement translation.  
🔹 Communication issues with geographically distributed teams.

### **Managing Scope Creep**

🔹 Frequent addition of new features beyond initial requirements.  
🔹 Poor requirement prioritization leads to delays.  
🔹 Stakeholders pushing last-minute changes without impact analysis

### **Time Constraints & Unrealistic Deadlines**

🔹 Short project timelines limit proper requirement analysis.  
🔹 BAs are expected to deliver documentation quickly, risking quality.  
🔹 Stakeholders push for faster delivery without considering feasibility.

### **Lack of Technical Understanding**

🔹 BAs may not fully understand complex technical constraints.  
🔹 Difficulty translating business requirements into technical specifications.  
🔹 Miscommunication with developers leads to incorrect implementation.

### **Difficulty in Prioritizing Requirements**

🔹 Stakeholders want everything as "high priority."  
🔹 Limited budget and time force trade-offs between features.  
🔹 No proper framework to evaluate critical vs. non-critical requirements.

### **Resistance to Change from End-Users**

🔹 Employees resist new system implementations.  
🔹 Users fear job losses or struggle with new processes.  
🔹 Lack of training and onboarding support.

### **Inadequate Documentation & Traceability**

🔹 Missing or outdated **BRD, FRS, and SRS** leads to confusion.  
🔹 No proper version control for requirement updates.  
🔹 Traceability gaps between requirements and testing.

### **Challenges in Agile Environments**

🔹 Continuous requirement changes make documentation difficult.  
🔹 BAs struggle to adapt to short sprint cycles.  
🔹 Difficulty in balancing **BA role in Agile vs. traditional Waterfall models**.

**Answer-12**

## **Key Objectives of Document Naming Standards**

1. **Consistency:** Ensures all files follow a uniform structure.
2. **Searchability:** Helps users find documents quickly.
3. **Version Control:** Tracks document revisions efficiently.
4. **Clarity:** Avoids confusion by using descriptive names.
5. **Compliance:** Ensures adherence to industry and organizational policies.

Example- [Project/Department]\_[DocumentType]\_[Title]\_[Version]\_[Date]\_[Author]

**Answer-13**

Never say NO to client

There is no word called as “BY DEFAULT”

Never imagine anything in terms of GUI.

Question the existence of existence /question everything in the world.

**DO’s**

### **Understand Business Needs Clearly**

✔ Conduct **requirement-gathering sessions** (interviews, workshops, surveys).  
✔ Ask the right **questions** to uncover hidden needs.  
✔ Use **elicitation techniques** like brainstorming, prototyping, and observation.

### **Maintain Clear & Concise Documentation**

✔ Create **structured** documents (BRD, SRS, FSD, Use Cases).  
✔ Use **diagrams and visual models** (flowcharts, wireframes) for clarity.  
✔ Keep **version control** and update documents regularly.

### **Communicate Effectively with Stakeholders**

✔ Ensure clear communication between **business & technical teams**.  
✔ Use **simple, jargon-free language** for non-technical users.  
✔ Conduct regular **meetings, stand-ups, and status updates**.

### **Be Proactive & Adaptable**

✔ Anticipate potential **risks & changes** in project scope.  
✔ Stay updated with **industry trends & new technologies**.  
✔ Adapt to **Agile, Waterfall, or Hybrid methodologies** as needed.

### **Focus on Requirement Prioritization**

✔ Use techniques like **MoSCoW (Must have, Should have, Could have, Won’t have)**.  
✔ Ensure business value alignment before finalizing requirements.  
✔ Collaborate with stakeholders to define **realistic project goals**.

### **Collaborate with Development & Testing Teams**

✔ Involve developers & testers **early** in requirement discussions.  
✔ Clarify **technical feasibility** and system limitations.  
✔ Assist in **User Acceptance Testing (UAT)** and issue resolution.

### **Ensure Traceability & Change Management**

✔ Use a **Requirement Traceability Matrix (RTM)**.  
✔ Track **changes systematically** using Change Request Logs.  
✔ Analyse the **impact of changes** on project timelines and cost.

## **Don’ts of a Business Analyst**

### **Don’t Assume Requirements – Always Validate**

Avoid making **assumptions** about business needs.  
Do not rely on a **single stakeholder’s opinion**—validate across teams.  
Never finalize requirements **without approval** from key stakeholders.

### **Don’t Overlook Stakeholder Conflicts**

Ignoring stakeholder **conflicts** can lead to project delays.  
Don’t **exclude end-users** from discussions—they are the system’s real users.  
Avoid taking sides—maintain a **neutral, analytical approach**.

### **Don’t Skip Documentation & Version Control**

Avoid keeping requirements **only in emails or informal notes**.  
Don’t neglect proper **naming conventions & version tracking**.  
Never **alter requirements without documentation and approval**.

### **Don’t Focus Only on Functional Requirements**

* Ignoring **non-functional requirements (NFRs)** (performance, security, scalability) can cause system failures.
* Always discuss **UI/UX, compliance, and scalability** with technical teams.

### **Don’t Resist Change or Be Rigid**

Business needs evolve—don’t resist **agile changes**.  
Avoid saying **“It’s not my job”**—BAs must be flexible problem-solvers.  
Be open to **feedback & process improvements**.

### **Don’t Ignore Testing & UAT Involvement**

A BA’s job doesn’t end with documentation.  
Don’t assume developers will **interpret requirements correctly** validate their understanding.  
Always participate in **test case review & UAT discussions**.

**Answer-14**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Package** | **Sub-system** |
| **Definition** | A group of related classes or components | A larger functional unit consisting of multiple packages |
| **Scope** | Narrower, focuses on specific functionality | Broader, encompasses major system components |
| **Granularity** | Smaller, less complex | Larger, more complex |
| **Dependency** | May depend on other packages | Depends on and communicates with other sub-systems |
| **Example** | A Java package like com.bank.transactions | A sub-system like User Management Sub-system |
|  |  |  |

A **package** refers to a group or collection of related classes, modules, or components bundled together within a software project. It is primarily a structural element within a single layer of an application.

**Example**: packages like:

* com.bank.transactions
* com.bank.customers
* com.bank.utils

A **sub-system** is a larger, more self-contained component or a group of packages/modules that together fulfil a major part of the overall system's functionality. It represents a higher-level architectural component that can communicate with other sub-systems to form the complete system.

**Example**: In an enterprise software system, you could have sub-systems like:

* User Management Sub-system
* Inventory Management Sub-system
* Order Processing Sub-system

**Answer-15**

**Camel casing** is a naming convention commonly used in programming where a compound word or phrase is written without spaces, and each subsequent word starts with a capital letter (except the first word, which starts with a lowercase letter). This style of writing mimics the humps of a camel, which is where the name comes from.

1. **Programming**: It’s commonly used in many programming languages to name variables, functions, methods, and other identifiers.

For example: userLogin

1. **File Naming**: Camel case is sometimes used for naming files and directories in codebases, especially when spaces or underscores are avoided.

For example : userProfileImage.png

1. **JavaScript**: In JavaScript, camel casing is the convention for naming variables, functions, and method names.
   * let userAge = 25;
2. **API and Web Development**: Many API endpoints and paths also use camel case to ensure readability and uniformity, for example:
   * getUserInfo
   * postNewOrder

Use

* It enhances readability, especially in places where spaces are not allowed, like variable names in code.
* It ensures consistency across naming conventions, particularly in programming.

**Q16) Illustrate Development server and what are the accesses does business analyst has ?**

**Answer- 16**

A **Development Server** is an environment where developers write, test, and debug code before deploying it to higher environments like staging or production. It is an essential part of the software development lifecycle (SDLC) and allows developers to implement and verify new features or fixes without affecting end users.

#### ****Key Features of a Development Server:****

* Used for coding, debugging, and testing new functionalities.
* Contains incomplete or experimental features that are not ready for production.
* Often runs on local machines or isolated network environments.
* May not have strict security controls since it is only for internal use.
* Typically integrates with version control systems (e.g., Git) to track changes.

#### ****Illustration of a Development Server Setup:****

A typical software development workflow involves multiple environments:

1. **Development Server:** Where the initial coding and unit testing happen.
2. **Testing (QA) Server:** Where testers verify the functionality and report bugs.
3. **Staging Server:** A replica of production used for final testing before release.
4. **Production Server:** The live system used by end-users.

Developer --> Development Server --> QA Server --> Staging Server --> Production Server

A Business Analyst (BA) typically does not have direct access to the development server, as their role is more focused on gathering requirements, analyzing business processes, and bridging the communication between the business and development teams.

Read on access

Access to test environment

Limited access to development tools

Communication and collaboration with development teams

Access to development documentation

**Q17)** What is Data Mapping ?

**Answer-17**

**Data Mapping** is the process of linking or matching data fields from one source to corresponding fields in another system. It is commonly used in data integration, migration, transformation, and analytics to ensure consistency, accuracy, and usability across different databases, applications, or formats.

### **Key Aspects of Data Mapping:**

1. **Source and Target Mapping:** Identifying the **source** data fields and mapping them to the correct **target** fields in another system.
2. **Transformation Rules:** Defining rules for data modification, such as format changes, unit conversions, or data type adjustments.
3. **Validation & Quality Checks:** Ensuring the mapped data meets business rules and does not introduce inconsistencies.
4. **Automation:** Using tools to automate mapping for large-scale data transfer.

### **Where is Data Mapping Used?**

1. **Data Migration:** Moving data from one database or system to another (e.g., migrating from an old CRM to a new ERP system).
2. **Data Integration:** Combining data from multiple sources for a unified view (e.g., merging customer data from different platforms).
3. **ETL (Extract, Transform, Load) Processes:** Used in data warehousing to extract, transform, and load data into a central repository.
4. **API Data Exchange:** Mapping data between different applications via APIs for seamless communication.
5. **Regulatory Compliance:** Ensuring data follows legal and industry standards (e.g., GDPR, HIPAA).

### **Example of Data Mapping**

#### ****Source Data (CSV file)****

| **Name** | **DOB** | **Contact Number** |
| --- | --- | --- |
| XXX | 01-02-1990 | 9876543210 |
| ZZZ | 12-05-1985 | 8765432109 |

#### **Target Database (CRM System)**

| **Full Name** | **Date of Birth** | **Phone Number** |
| --- | --- | --- |
| XXX | 1990-02-01 | +91-9876543210 |
| ZZZ | 1985-05-12 | +91-8765432109 |

Here, the **Name** field is mapped to **Full Name**, and the **DOB** format is transformed into **YYYY-MM-DD** format.

**Answer-18**

An **API (Application Programming Interface)** is a set of rules and protocols that allow different software applications to communicate with each other. APIs define how requests and responses should be structured so that systems can exchange data efficiently and securely.

### **API Integration**

API integration is the process of connecting two or more applications using APIs to enable seamless data exchange. This is commonly used in:

* **Third-party services** (e.g., payment gateways, social media logins).
* **Data synchronization** (e.g., integrating a CRM with an ERP).
* **Automation** (e.g., fetching real-time stock prices or weather updates).

If your application expects dates in **dd-mm-yyyy** format but receives data from a US-based application in **mm-dd-yyyy** format, you need to implement a **date transformation process** before storing or processing the data.

Solution Approach

**Receive the API Data:** with a date field formatted as **mm-dd-yyyy**.

Extract & Transform the Date Format : Before saving it into your system, convert "dob": "12-25-1990" (MM-DD-YYYY) → "25-12-1990" (DD-MM-YYYY).

Store or Use the Transformed Data