**Case Study:** A customer can make a payment either by Card or by Wallet or by Cash or by Net banking.

**Question 1:** Draw a Use Case Diagram



Question 2: Derive Boundary Classes, Controller classes, Entity Classes.

**Boundary Class:** Represents the interface or interaction point between the system and its external actors like customer.

* PaymentPage UI for the customer to select payment method
* CardPaymentForm Form to enter card details
* WalletPaymentScreen Screen to choose and authenticate wallet
* CashPaymentInfo Shows instructions for cash payments
* NetBankingForm Form to log in to bank and authenticate

**Controller Class:** Controller classeshandle user input, manage flow and coordinate between boundary and entity classes. it's the "controller" that manages the flow of actions within the use case.

* PaymentController Main controller to handle payment processing logic
* CardPaymentController Handles card payment flow
* WalletPaymentController Manages wallet integration and validation
* CashPaymentController Confirms and logs cash payment requests
* NetBankingController Manages bank authentication and fund transfer

**Entity Classes:** Represents a business logic or object within the system that holds and updates data.

* Payment Superclass with common payment attributes (amount, date, status)
* Card Stores card details (card number, expiry, CVV)
* Wallet Stores wallet ID, balance info
* BankAccount Stores bank credentials, account number, IFSC
* Customer Represents the customer (ID, name, contact info)
* Transaction Stores transaction ID, timestamp, result

**Question 3:** Place these classes on a three tier Architecture.

**Presentation Tier (UI Layer)**

Handles user interaction and displays data. Corresponds to Boundary Classes.

* PaymentPage
* CardPaymentForm
* WalletPaymentScreen
* CashPaymentInfo
* NetBankingForm

**Business Logic Tier (Application Layer)**

Contains application-specific logic. Corresponds to Controller Classes.

* PaymentController
* CardPaymentController
* WalletPaymentController
* CashPaymentController
* NetBankingController

**Data Tier (Persistence Layer)**

Manages access to the database or other storage systems. Corresponds to Entity Classes.

* Payment
* Card
* Wallet
* BankAccount
* Customer
* Transaction

**Question 4:** Explain Domain Model for Customer making payment through Net Banking.

A **Domain Model** is a conceptual representation of real-world entities, their attributes, and relationships in a particular domain (subject area). It focuses on capturing the business logic and key concepts relevant to the system without involving technical details like databases or UI. It helps to:

* Understand what objects exist in your problem space.
* Define how they are related to each other.
* Prepare the ground for design, database structure and development.

In this case, the focus is on how a **Customer** makes a **Payment** using the **Net Banking** payment method.

**Key Entities (Classes):**

|  |  |  |
| --- | --- | --- |
| Entity | Attributes | Relationships |
| Customer | customerId, name, email, mobileNumber | Can place multiple Orders. |
| Payment | paymentId, paymentDate, amount, paymentStatus | Associated with one Payment Method. |
| NetBanking | authentication, fundtransfer, transactionHistory | Is one type of Payment Method. |
| Bank | bankId, bankName, IFSC | NetBanking uses a Bank. |
| Transaction | transactionId, timeStamp, status | Confirmation of the payment. |

Domain Model Class Diagram.



**Question 5:** Draw a sequence diagram for payment done by Customer Net Banking

A sequence diagram is an interaction diagram that visually represents the sequence of messages exchanged between objects in a system to execute a scenario, illustrating the dynamic behaviour of a system.

**Question 6:** Explain Conceptual Model for this Case

A Conceptual Model is a high-level, abstract representation of the system that captures the main concepts, relationships and rules from the real-world domain, without considering technical implementation.

It’s typically used during early stages of system design to help business stakeholders, analysts and developers understand the problem space clearly

**Conceptual Model for:** Customer Making Payment via Net Banking

This model explains the core business concepts and how they relate, from a domain expert's perspective — focusing on what the system should do, not how it will be done.

**Key Concepts** (Classes/Entities):

* Customer - The user who initiates a payment transaction
* Payment - Represents a request to pay a specific amount through a chosen method
* NetBankingAccount - Bank account used to complete the payment
* Bank - Financial institution that verifies and processes the payment
* Transaction - Record of the result of the payment process, including success or failure status

**Relationships between concepts:**

* A Customer can initiate multiple Payments
* Each Payment is made using one NetBankingAccount
* A NetBankingAccount is associated with a Bank
* A Transaction is generated for every Payment

**Business Rules in the Model:**

* A customer must have internet banking credentials to initiate payment via Net Banking.
* Only valid, active bank accounts can be used.
* Transactions are recorded only after successful verification by the bank.
* One payment creates one corresponding transaction.

**Conceptual Flow Description:**

* The Customer selects Net Banking as the payment method.
* Enters credentials associated with a NetBankingAccount.
* The system sends the information to the corresponding Bank for verification.
* If successful, a Payment is confirmed.
* A Transaction record is created and linked to that payment

**Question 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 stands for Model-View-Controller — it's a software design pattern that separates an application into three interconnected components:

* Model - Represents data and business logic
* View - Handles the display/UI
* Controller - Handles input, processes it, updates model/view

**Purpose of MVC:**

* Separation of concerns
* Easier to maintain and extend
* Allows multiple views of the same data
* Improves testability and modularity.

**MVC Rules to Derive Classes from a Use Case Diagram:**

When you're working from a Use Case Diagram and moving toward classes, you can apply these principles:

|  |  |  |
| --- | --- | --- |
| **Use Case Element** | **Corresponding Class Type** | **Purpose** |
| Actors | Boundary Classes (View Layer) | Interfaces between users and system |
| Use Cases / Flow Logic | Controller Classes | Orchestrates the process & logic |
| Conceptual Objects / Entities | Entity Classes (Model Layer) | Holds data & business logic |

**Example:** For the use case "Customer makes payment via Net Banking"

|  |  |  |
| --- | --- | --- |
| **From Use Case Diagram** | **Derived Class Type** | **Example Class** |
| Customer (Actor) | Boundary Class | PaymentPage, NetBankingForm |
| Make Payment (Use Case) | Controller Class | PaymentController, NetBankingController |
| Payment, Bank, Transaction | Entity Class | Payment, Bank, Transaction |

**Guidelines to Place Classes in 3-Tier Architecture:**

After identifying classes, you map them to the 3-tier structure like this:

|  |  |  |
| --- | --- | --- |
| **Tier** | **Typical Classes** | **Responsibilities** |
| Presentation Layer | Boundary Classes (UI) | Interacts with users, handles input/output |
| Business Logic Layer | Controller Classes | Contains core logic, decision making |
| Data Layer | Entity Classes | Data storage, persistence, business rules |

**Example:**

|  |  |
| --- | --- |
| **Class** | **Placed In Tier** |
| PaymentPage / NetBankingForm | Presentation Tier (UI) |
| PaymentController / NetBankingController | Business Logic Tier (Controller) |
| Payment, Transaction, BankAccount | Data Tier (Entity) |

**Question 8:** Explain BA contributions in project (Waterfall Model – all Stages)

Waterfall Model is a linear and sequential software development approach where each stage must be completed before moving to the next.

|  |  |
| --- | --- |
| **Stage** | **BA Role & Contribution** |
| Requirement Gathering & Analysis | - Elicit detailed requirements from stakeholders.- Conduct interviews, workshops, and document analysis.- Create BRD (Business Requirement Document) and Use Case Diagrams.- Ensure clarity, completeness, and correctness of business needs. |
| System Design  |  - Collaborate with architects and designers to ensure requirements are accurately translated.- Validate logical data models, process flows, and UML diagrams.- Support mapping of business rules to design decisions. |
| Implementation (Development) | - Clarify requirements and address queries from developers.- Assist in refining user stories or specifications.- Validate whether the code is adhering to business logic during development reviews. |
| Testing | - Help QA team create Test Cases based on requirements.- Perform Requirement Traceability Matrix (RTM) to ensure full coverage.- Participate in User Acceptance Testing (UAT) preparation and execution. |
| Deployment | - Validate deployed solution against business goals.- Assist in Go-Live checklists and coordinate feedback from end-users.- Support change management and user training documentation. |
| Maintenance | - Collect user feedback post-deployment.- Analyse change requests and document new enhancements.- Support bug triage, root cause analysis, and communicate fixes back to business. |

**Question 9:** What is conflict management? Explain using Thomas – Kilmann technique

Conflict Management is the process of identifying and handling conflicts fairly, efficiently and sensibly in a way that prevents escalation and leads to productive outcomes.

In any project — especially involving multiple stakeholders, departments, or teams — conflicts over ideas, resources, timelines, or priorities are natural. A Business Analyst, Project Manager, or Team Lead needs strong conflict management skills to ensure collaboration and smooth project execution.

**Thomas-Kilmann Conflict Management Model**

The Thomas-Kilmann Conflict Mode Instrument (TKI) is a well-known framework for understanding how people handle conflict.

It defines conflict-handling styles along two dimensions

* Assertiveness - The extent to which you try to satisfy your own concerns.
* Cooperativeness - The extent to which you try to satisfy others’ concerns.

From this, five conflict-handling styles are derived:

|  |  |  |  |
| --- | --- | --- | --- |
| **Style** | **Assertiveness** | **Cooperativeness** | **When to Use** |
| Competing | High | Low | Quick decisions, when you’re sure you’re right. |
| Collaborating | High | High | When you want win-win solutions, shared goals. |
| Compromising | Medium | Medium | Temporary solutions, both sides give and take. |
| Avoiding | Low | Low | When the issue is trivial or emotions are high. |
| Accommodating | Low | High | When the relationship matters more than the issue. |

**Example:**

Situation: Two developers disagree on which design pattern to use for payment module logic.

* If the BA wants to ensure the fastest delivery with one person’s idea, they may use **Competing**.
* If the BA wants both developers to work out a design that blends both views, they use **Collaborating**.
* If the deadline is near and the choice is not critical, they could suggest **Compromising**.
* If the disagreement is minor and can be deferred, **Avoiding** is used.
* If maintaining team harmony is more important than this one technical choice, the BA might suggest **Accommodating**.

**Question 10:** List down the reasons for project failure

**Top Reasons for Project Failure**

|  |  |
| --- | --- |
| **Reason** | **Description** |
| Unclear Requirements | Poorly defined, incomplete or changing requirements. |
| Lack of Stakeholder Involvement | Key users or decision-makers are not actively engaged. |
| Unrealistic Planning | Deadlines, budgets or resources are underestimated. |
| Scope Creep | Continuous addition of new features without proper control. |
| Poor Communication | Misunderstandings or gaps between team, stakeholders, or clients. |
| Inadequate Risk Management | Failing to identify and mitigate risks early. |
| Weak Project Management | Poor planning, tracking, and leadership by the PM. |
| Technical Challenges | Overcomplicated solutions or reliance on unstable technologies. |
| Budget Overrun | Costs exceed estimates due to poor planning or scope changes. |
| Resource Constraints | Shortage of skilled people or lack of critical tools. |
| Lack of Prioritization | Teams work on low-value features while neglecting critical tasks. |
| Inadequate Testing | Software is deployed without thorough validation or UAT. |
| Poor Change Management | Teams fail to adapt when requirements or environments evolve. |
| Lack of Executive Support | No visible or active support from sponsors or management. |
| User Resistance to Adoption | Solution is delivered but users reject or avoid it. |

**Question 11:** List the Challenges faced in projects for BA

|  |  |
| --- | --- |
| **Challenge** |  **Description** |
| Unclear or Evolving Requirements | Stakeholders often don’t fully know what they want at the start or requirements change mid-project. |
| Stakeholder Conflicts | Different stakeholders have conflicting expectations and priorities. |
| Poor Communication Channels | Misunderstandings or delays due to unclear or broken communication flow. |
| Incomplete Business Knowledge | BA may lack domain knowledge, especially in new industries or technical contexts. |
| Tight Deadlines / Time Pressure | Business pressure to deliver fast often affects quality or thoroughness of analysis. |
| Scope Creep | Continuous addition of new features beyond original plan, leading to project overrun. |
| Technical Complexity | BA must bridge the gap between technical teams and non-technical stakeholders, which is challenging when solutions are highly technical. |
| Organizational Resistance to Change | Employees and stakeholders often resist new systems or process changes. |
| Budget Constraints | Solution expectations may exceed what’s affordable for the project. |
| Lack of Stakeholder Availability | Stakeholders are too busy or disengaged, delaying requirement gathering and validation. |
| Changing Business Environment | Business goals or market conditions change mid-project, forcing requirement revisions. |
| Misinterpretation of Requirements | Developers or testers misunderstand written or verbal requirements, causing rework. |
| Balancing Business Needs vs. Technical Limitations | Sometimes the business wants something that isn’t technically feasible (or affordable), and the BA must manage expectations. |

**Question 12:** Write about Document Naming Standards

Document Naming Standards are a set of rules and conventions that help ensure files are consistently named, so they are easy to:

* Identify, Retrieve, Sort, Share and Maintain.

This is especially important in collaborative environments like projects, software development, audits and knowledge management.

**Why Are Naming Standards Important?**

* Improves Search ability: Easier to locate the right document.
* Reduces Duplication: Avoids creating multiple versions of the same document.
* Provides Context: Clear names tell the user what’s inside the file.
* Simplifies Version Control: Makes tracking document revisions easy.
* Promotes Collaboration: Everyone can follow a unified structure.

**Common Document Naming Format:**

[ProjectCode]\_[DocumentType]\_[Subject/Title]\_[Version]\_[Date/Author]

**Example:** SCRUMFD\_BRD\_PaymentModule\_V1.0\_20250411\_JSmith.docx

**Best Practices for Document Naming:**

* Be Clear & Descriptive — the name should reflect the document’s purpose.
* Avoid Special Characters — use underscores \_ or dashes - instead of / \ : \* ? " < > |.
* Use Date Format YYYYMMDD — for chronological sorting.
* Include Version Control — e.g., V1.0, V2.1.
* Use Standard Abbreviations — e.g., BRD (Business Requirements Document), SRS (Software Requirements Specification), UAT (User Acceptance Testing).
* Keep It Consistent — apply the same rule across teams, folders, and projects.

**Question 13:** What are the Do’s and Don’ts of a Business Analyst?

**Do’s of a Business Analyst**

|  |  |
| --- | --- |
| **Do** |  **Why it Matters** |
| Understand the Business Need | Focus on solving the real problem, not just writing requirements. |
| Communicate Clearly | Use precise, unambiguous language in all documents and meetings. |
| Listen Actively | Give stakeholders space to express their needs without assumptions. |
| Ask Questions | Clarify uncertainties early to avoid misunderstandings later. |
| Use Visual Models | Diagrams (UML, flowcharts, mockups) help explain complex ideas. |
| Prioritize Requirements | Help the team focus on what's valuable, feasible, and time-sensitive. |
| Maintain Documentation | Keep BRDs, SRS, Use Cases, and Change Logs up-to-date. |
| Facilitate Collaboration | Be the bridge between business users and technical teams. |
| Adapt to Change | Requirements often evolve — stay flexible and responsive. |
| Support Testing and UAT | Help validate whether the solution meets business expectations |

**Don’ts of a Business Analyst**

|  |  |
| --- | --- |
| **Don’t** |  **Why it's a Mistake** |
| Assume Requirements are Obvious | Always verify and document, even if something seems "obvious." |
| Ignore Stakeholder Conflicts | Misaligned stakeholder expectations cause project failure. |
| Overcomplicate Solutions | Simplicity saves time, cost, and confusion. |
| Jump to Solutions Too Soon | Understand the problem fully before proposing solutions. |
| Use Technical Jargon with Business Users | Keep language accessible to your audience. |
| Neglect Documentation Updates | Outdated documents lead to rework and errors. |
| Be Rigid with Change | Projects evolve — being inflexible can block progress. |
| Work in Isolation | Frequent communication avoids misunderstandings. |
| Ignore Business Constraints | Requirements must respect time, budget, and resource limits. |
| Forget to Validate Requirements | Unverified requirements can lead to delivering the wrong solution. |

**Question 14:** Write the difference between packages and sub-systems

|  |  |
| --- | --- |
| **Package** |  **Sub-System** |
| Grouping of related code classes to simplify organization. (Think of it like folders on your computer.) | A complete functional block that can stand on its own or interact with others. (Think of it like a whole "mini-application" inside a bigger application.) |
| A Package is a logical grouping of related classes, interfaces, or components in software. | A Sub-System is a self-contained, larger, independent part of a system designed to perform a specific business function. |
| Scope: Small scale - groups classes for better organization and manageability. | Scope: Large scale - can include multiple packages, modules, and interfaces. |
| Purpose: Organize code, reduce complexity and improve maintainability. | Purpose: Divide system into independent, manageable chunks - improves architecture scalability. |
| Ex: com.project.payment (package for payment classes) | Ex: Payment Processing System (sub-system for handling all payment operations). |
| Contains classes, interfaces, sometimes other packages. | Contains packages, components, classes and even other sub-systems. |
| Supports modularity at the code level (usually for developers). | Supports modularity at the architectural level (for designers, architects and BAs). |
| Fine-grained. | Coarse-grained. |
| Drawn as a small folder icon (usually). | Represented as a component with interfaces and dependencies. |

**Question 15:** What is camel-casing and explain where it will be used

Camel-Casing is a naming convention used in programming where:

* The first word starts with a lowercase letter.
* Each subsequent word starts with an uppercase letter.
* No spaces, hyphens, or underscores are used.

Example: userName, customerPaymentInfo, totalAmount etc.

Camel-casing is widely used in **software development** for:

|  |  |
| --- | --- |
| Use Case |  Example |
| Variable Names | orderAmount, userName |
| Function / Method Names | processPayment(), getDetails() |
| Object Names | customerData, paymentStatus |
| JSON Keys | { "orderId": 1234 } |

**Camel-Case is used for?**

* Improves Readability — Easy to distinguish words in a single identifier.
* Avoids Spaces & Special Characters — Perfect for programming languages.
* Consistency — Makes code easier to maintain and understand across teams.

**Camel-Casing** is a standard practice in most modern programming languages for naming variables and functions, especially in: Java, JavaScript, Python, C#.

**Question 16:** Illustrate Development server and what are the accesses does business analyst has?

**What is a Development Server?**

A Development Server is an environment where software applications are built, tested, and validated before they are moved to production (live environment).

It’s mainly used by:

* Developers (to write and test code),
* Testers (to validate features),
* Business Analysts (to verify business logic implementation),
* And sometimes Product Owners (for review).

**What Access Does a Business Analyst (BA) Have on a Development Server?**

|  |  |
| --- | --- |
| **Type of Access** |  **Purpose** |
| Read/Review Access | **YES** - To review how the application looks & behaves against documented business requirements. |
| Test Data Input Access | **YES** - To enter dummy data and simulate user/business scenarios for validation. |
| Requirement Verification | **YES** - Confirm that features are built as per Business Requirement Document (BRD) or Use Cases. |
| Code-Level Access | **NO** - that’s the developer's responsibility. |
| Bug Logging & Reporting | **YES** - If BA finds mismatches between business rules and implementation, they log issues (usually in tools like JIRA, Bugzilla). |
| Demo/Walkthrough Access | **YES** - Sometimes BAs use the Dev server to conduct internal demos for stakeholders or UAT preparation. |
| Deployment Control | **NO** - BAs usually do not handle code deployments — DevOps or developers manage that. |

**Why is BA Access to the Development Server Important?**

* Early Validation: BAs can spot requirement gaps early in the Dev stage.
* Effective Communication: If something isn’t implemented as intended, the BA can immediately inform developers.
* Preparation for Testing: BAs often support testers by helping clarify acceptance criteria on the actual build.
* Demo Readiness: BAs ensure the application flows make sense before stakeholder presentations.

**Question 17:** What is Data Mapping?

**Data Mapping** is the process of matching fields from one data source to another - ensuring data is correctly transferred, transformed, and used between systems, databases, or formats.

In short, Data Mapping defines how data in one place corresponds to data in another place.

**Why is Data Mapping Important?**

* When you move data from one system to another.
* When you integrate two applications (like CRM to ERP).
* When you perform data migration or data transformation.
* When generating reports that pull data from multiple sources.

**Example:**

|  |  |
| --- | --- |
| **Source System (CRM)** | **Target System (ERP)** |
| FirstName | Customer\_Name |
| PhoneNumber | Contact\_Number |
| EmailID | Customer\_Email |
| AddressLine1 + AddressLine2 | Full\_Address |

**Data Mapping Used for?**

|  |  |
| --- | --- |
| **Scenario** | **Purpose** |
| Data Migration | Moving data between old and new systems. |
| System Integration | Connecting different applications. |
| ETL Process | Extract, Transform, Load pipelines in data warehousing. |
| Reporting / Analytics | Combining data from multiple sources for insights. |

**Benefits of Data Mapping:**

* Ensures data consistency.
* Reduces human error in migrations or integrations.
* Simplifies complex data transfers.
* Enables automated data transformations.

**Question 18:** What is API. Explain how you would use API integration in the case of your application Date format is dd-mm-yyyy and it is accepting some data from Other Application from US whose Date Format is mm-dd-yyyy?

**API** stands for **Application Programming Interface**. It’s like a messenger or bridge that allows two software applications to talk to each other, exchange data and perform operations without needing to know the internal details of each system.

**Simply Example**: Think of an API as a **waiter** in a restaurant:
You (the client) place an order → the waiter (API) takes your request → the kitchen (server) prepares your food → the waiter brings it back to you.

**In Software Terms:**

When two systems (your application & another application) need to exchange data:

* Your app sends a request to the other app via an API.
* The API returns data or a result back.

**Given Case:** Date Format Problem

Your Application's Date Format: dd-mm-yyyy

US Application's Date Format: mm-dd-yyyy

If you integrate with their API and directly accept their date values without handling the difference, it will cause:

* Invalid data.
* Date parsing errors.
* Wrong report outputs.

**How to Handle This via API Integration:**

**Step 1:** Receive Data via API, with US Format (MM-DD-YYYY)

**Step 2:** Apply Date Conversion Logic

When your application receives the data, you should:

* Parse the incoming date.
* Detect the format (mm-dd-yyyy from US).
* Convert it to your required format (dd-mm-yyyy).

**Step 3: Store or Display Converted Date**

Now your application uses the correct date format for:

* Storing in your database.
* Displaying to users.
* Passing to other internal modules.