VIT®
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

Vellore – 632014, Tamil Nadu, India

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

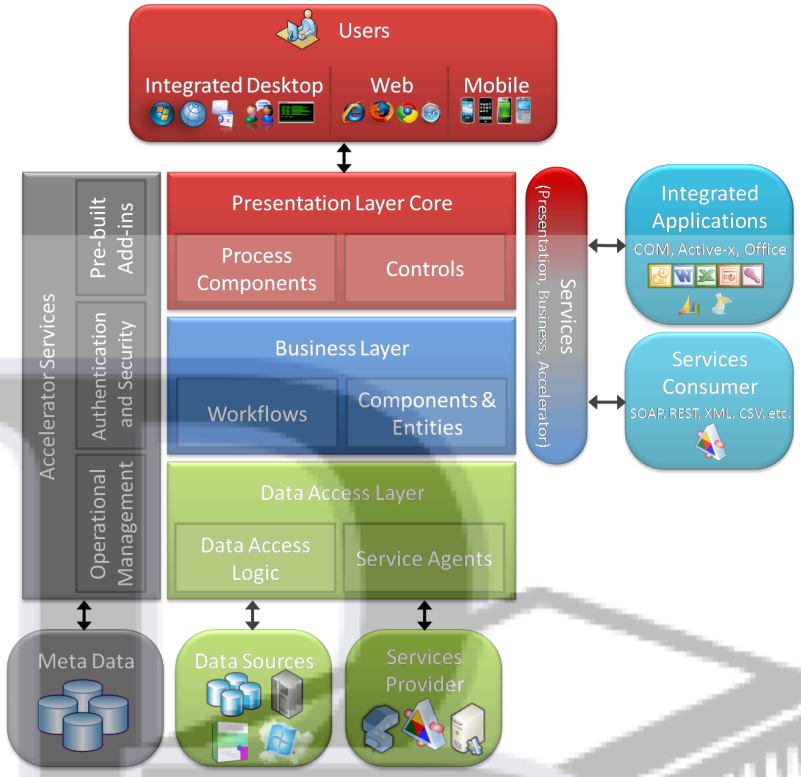
Winter Semester 2023-2024

Digital Assignment – 2

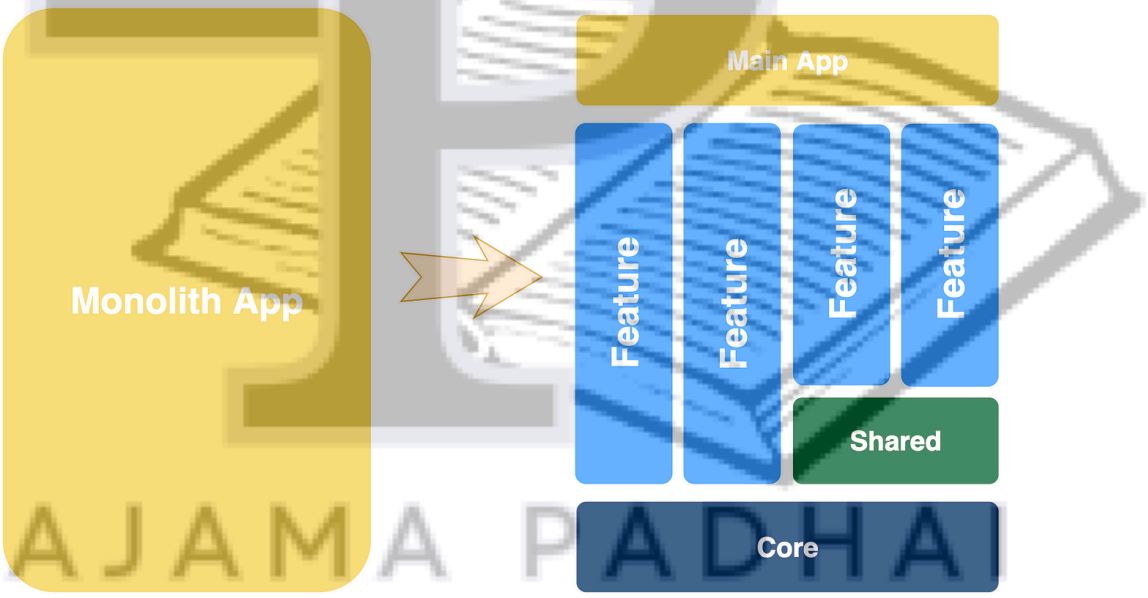
SLOT	**: C2+TC2**
Programme Name	**: B.Tech. CSE**
Course Name & code	**: Software Engineering & BCSE301L**
Class Number	**: VL2023240500732**
Faculty Name	**: Dr. Saurabh Agrawal**

Q.1. When should a modular design be implemented as monolithic software? How can this be accomplished? Is performance the only justification for implementation of monolithic software?

Modular software architecture entails dividing a software system into smaller, self-contained, and reusable components known as modules. These modules possess clear interfaces and functionalities, communicating with one another via standardized protocols or APIs. Such an architecture enables developers to independently construct, test, deploy, and update each module, devoid of any impact on the overall system. This approach enhances the software's modularity, flexibility, and scalability, facilitating seamless adaptation to evolving requirements and technologies.

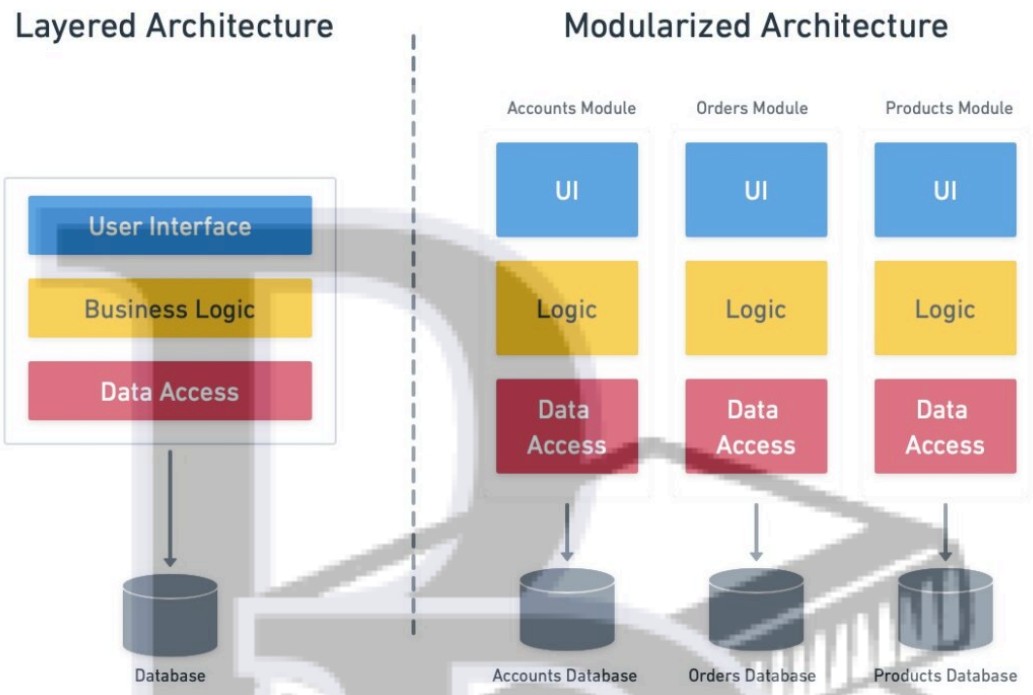


Monolithic software architecture involves constructing a software system as a solitary, unified entity, with all components and functionalities tightly interwoven into a singular executable or library. This approach streamlines the development, deployment, and execution processes, eliminating the necessity to oversee numerous modules or dependencies. Nonetheless, it renders the software more inflexible, intricate, and challenging to alter, expand, and upkeep.



When comparing modular and monolithic software architecture, four criteria can be utilized: performance, scalability, maintainability, and complexity. Generally, monolithic software architecture exhibits superior performance and lower complexity compared to modular software architecture. However, modular software architecture boasts greater scalability and maintainability.

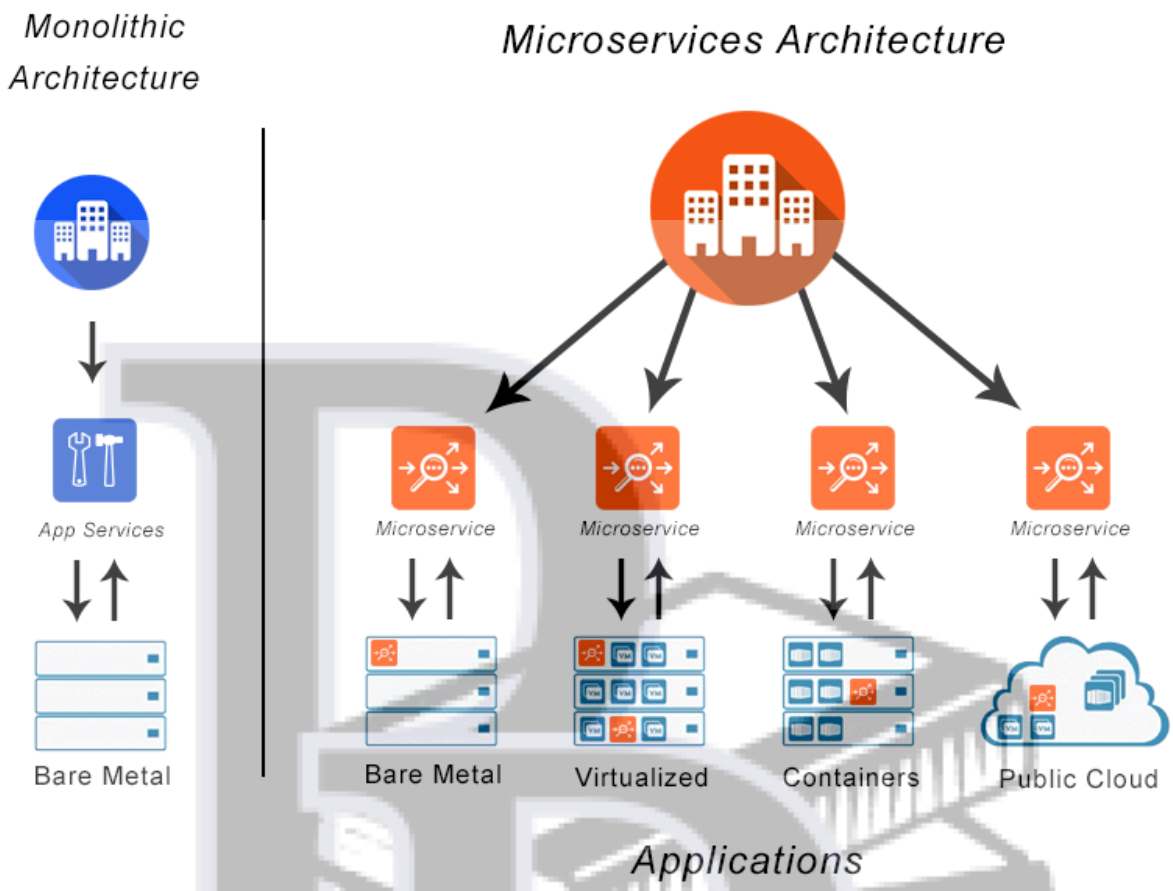
In terms of performance, monolithic software architecture experiences less overhead and latency in executing and communicating within a single unit. Conversely, modular software architecture can optimize the design and implementation of each module and communication protocols to achieve high performance.



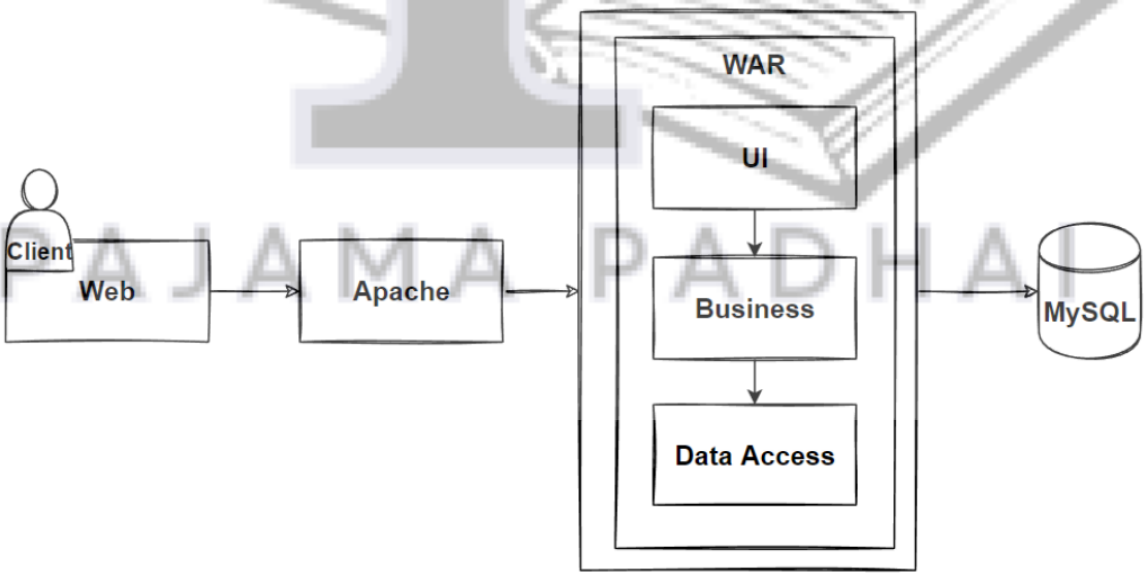
Regarding scalability, modular software architecture offers more flexibility, enabling easy addition, removal, or replacement of modules to adapt to changing workloads and demands. In contrast, monolithic software architecture necessitates more resources and effort to scale up or out the entire system and may encounter limitations concerning hardware or software compatibility.

Concerning maintainability, modular software architecture permits developers to isolate, debug, and rectify errors in each module without impacting the remainder of the system. It also facilitates quicker and more frequent updates and releases, alongside enhanced code reuse and quality. On the other hand, monolithic software architecture complicates the identification, location, and resolution of issues within a large codebase.

Lastly, modular software architecture entails higher complexity than monolithic software architecture due to increased design decisions, coordination, and integration among multiple modules and teams. Additionally, it demands more skills and tools for managing dependencies, configuration, and deployment of the modules.



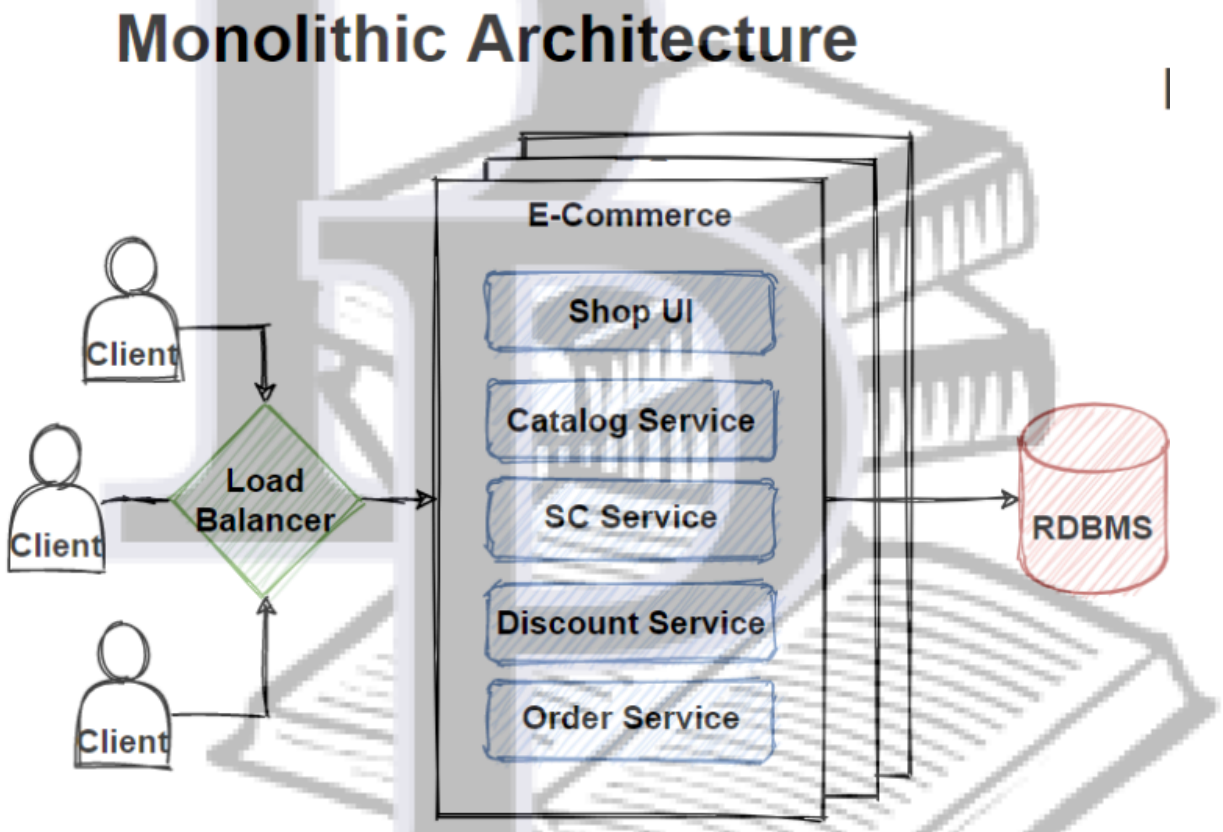
The decision between monolithic and modular software architecture hinges on the specific context, objectives, and limitations of each project. Generally, for smaller, straightforward, and stable systems, monolithic architecture might prove more efficient. Conversely, for intricate and dynamic systems, modular architecture could be better suited. Team size and culture also play a role: smaller, closely-knit teams may lean towards monolithic architecture, while larger, diverse teams might favor modular approaches. Furthermore, if the technology and environment remain uniform and steady, monolithic architecture could offer greater compatibility. Conversely, in environments characterized by diverse technologies and ongoing evolution, modular architecture may offer superior adaptability.



When deliberating over the decision of whether to opt for a modular design or stick with a monolithic approach for software development, it is of paramount importance to thoroughly comprehend the intricacies inherent in each methodology, taking into account the unique contextual factors surrounding the development and deployment of the software in question.

A modular design is often preferred due to its capacity to deconstruct intricate systems into more digestible components. This method encourages reusability, adaptability, and simplified upkeep, since modules can be created, verified, and modified autonomously. Additionally, it streamlines scalability, enabling specific segments of the system to expand or contract as required.

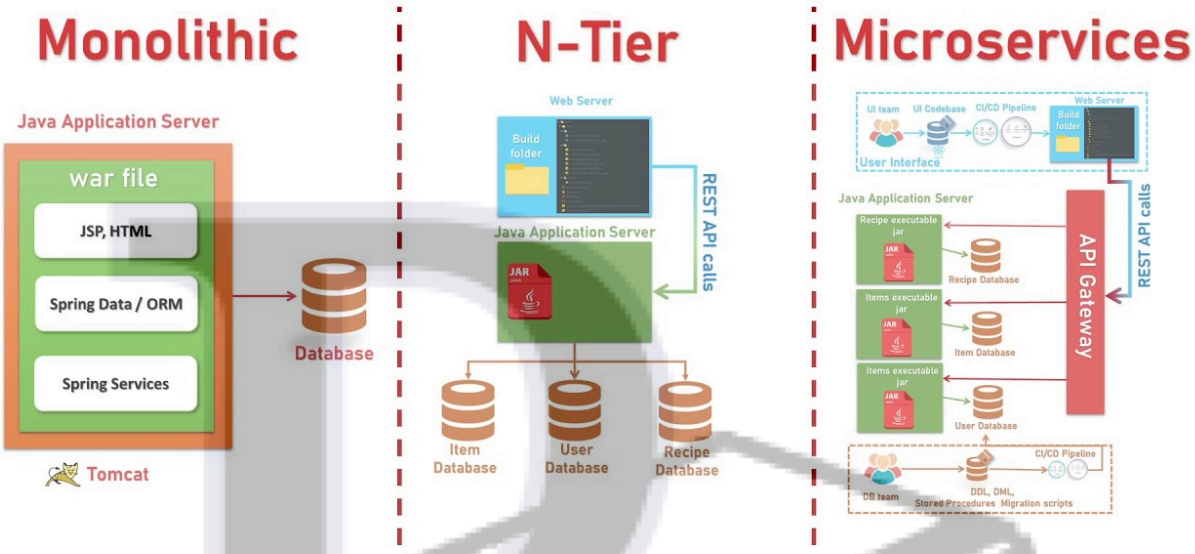
Conversely, monolithic software architecture consolidates the entire application into a singular, cohesive unit. Although this might appear to offer less flexibility when juxtaposed with a modular framework, there are situations where it can yield significant advantages.



In situations where the software requirements remain stable and clearly delineated, with minimal expected necessity for frequent updates or alterations to individual components, a monolithic architecture can be advantageous. Fragmenting the software into distinct modules in such cases may introduce undue complexity without commensurate benefits. Maintaining everything within a monolithic structure enables developers to streamline the development process, diminish overhead, and concentrate on swiftly and efficiently delivering functionality.

Implementing a modular design within a monolithic architecture requires structuring the codebase in a manner that upholds sound modular design principles internally, despite the software being deployed and managed as a unified entity. This entails organizing the code into coherent modules, adhering to suitable naming conventions, and enforcing encapsulation to maintain related functionality together.

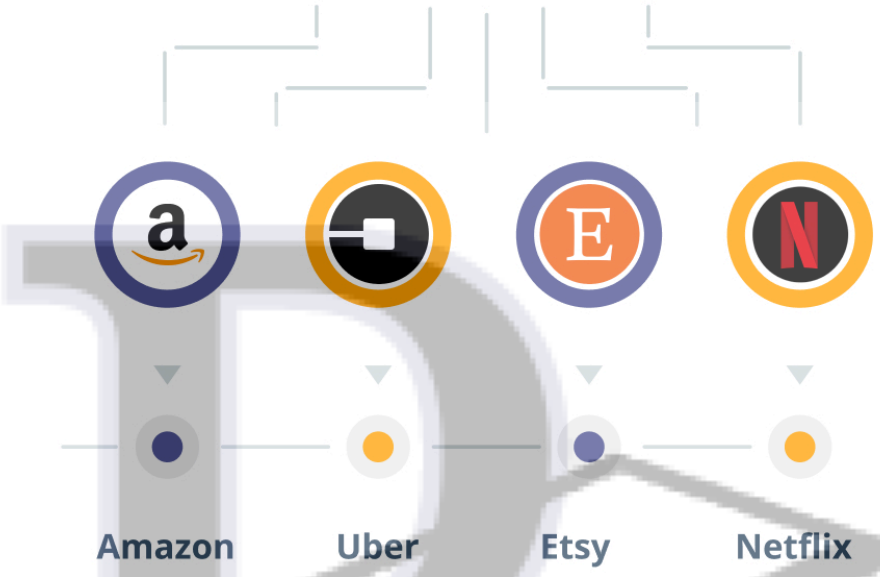
Software Architecture



Moreover, while performance can certainly justify the adoption of monolithic software, it is not the exclusive factor to consider. Other aspects such as development duration, maintenance convenience, and the specific requirements of the application also wield considerable influence.

For example, in the scenario where the application is relatively compact in size and experiences low to moderate traffic, the administrative burden associated with handling distinct modules could surpass the potential performance enhancements. In such instances, selecting a monolithic architecture could prove to be a practical decision, enabling developers to concentrate on expeditiously and effectively delivering functionality without becoming entangled in the intricacies of a distributed system.

To illustrate the contrast between modular and monolithic software architecture, let's examine the following examples. Modular software architecture is epitomized by microservices, a prevalent architectural style that segments a system into small, autonomous, and loosely connected services that interact via APIs. Prominent software platforms like Netflix, Amazon, and Uber embrace this approach. On the other hand, monolithic software architecture is evident in operating systems, fundamental software components that oversee and coordinate the hardware and software resources of a computer. Operating systems such as Linux, Windows, and MacOS adhere to this architecture.



Deciding to adopt a modular design within a monolithic software setup demands thorough evaluation of multiple factors, such as requirement stability, development duration, maintenance complexities, and performance considerations. Although a modular approach is typically preferred due to its adaptability and scalability, there are instances where a monolithic architecture can provide simplicity and effectiveness without compromising significantly on modularity and maintainability. Ultimately, the decision should be guided by the precise requirements and limitations of the project in question.

Q.2. Using a data flow diagram and a processing narrative, describe a computer-based system that has distinct transform flow characteristics. Define flow boundaries and map the DFD into software architecture?

Data Flow Diagram (DFD):

A data flow diagram (DFD) serves as a visual map illustrating how data moves through a system. It's a powerful tool for understanding the dynamics of information within a given framework. Here's a breakdown of its key components:

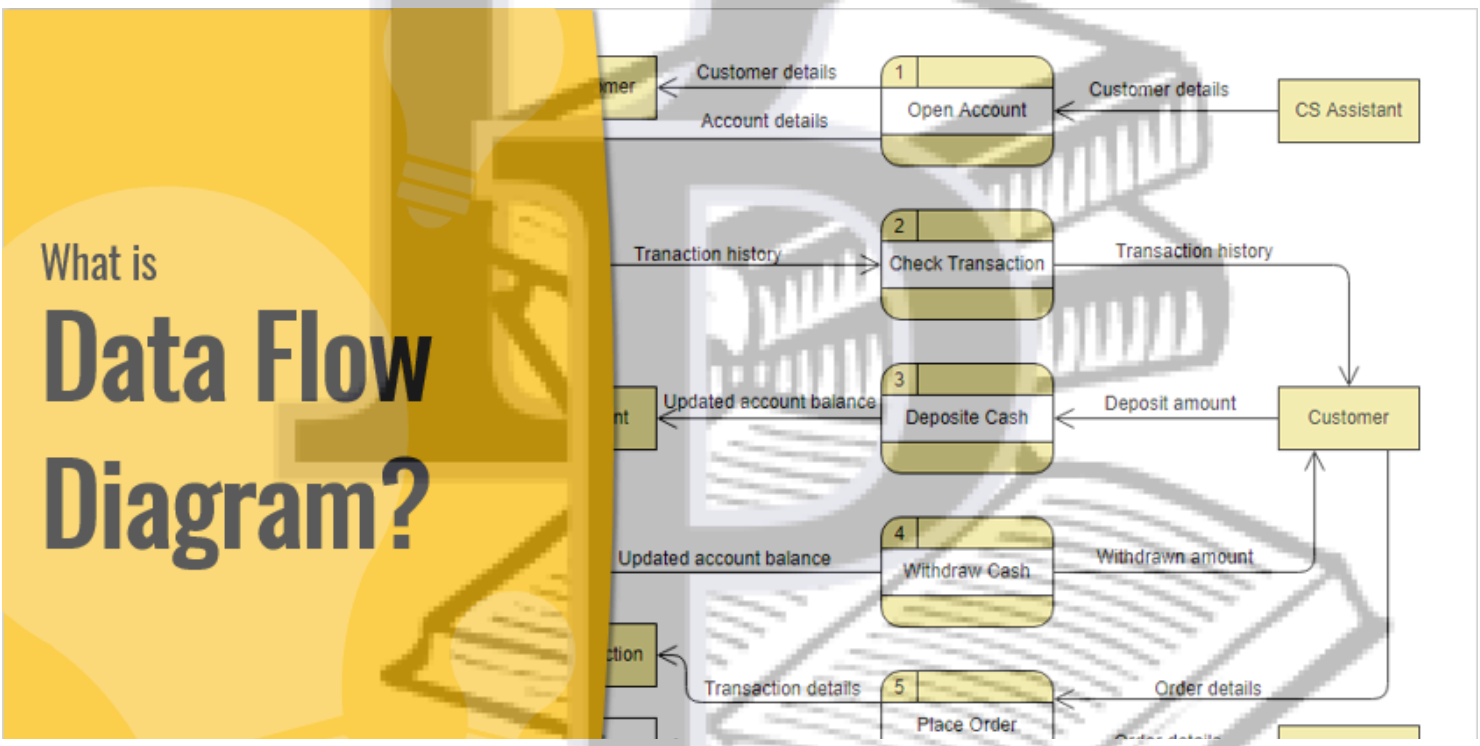
Processes: These are the activities or functions that manipulate data within the system. Processes could be anything from calculations, transformations, or validations performed on the data. Each process is typically represented by a circle or rectangle in the diagram.

Data Stores: Data stores are where information is held within the system. They can represent databases, files, or any other storage medium. Data stores are depicted by rectangles in the diagram and show where data is stored or retrieved from.

Data Flows: Data flows are the pathways through which information travels within the system. They depict the movement of data from one point to another, whether it's between processes, data stores, or external entities. Data flows are usually represented by arrows in the diagram.

External Entities: These are the sources or destinations of data that exist outside the system being depicted. They could be users, other systems, or even physical devices. External entities are represented by squares or rectangles in the diagram and illustrate interactions between the system and its external environment.

By visually representing these elements and illustrating how they interact, a data flow diagram provides a clear understanding of how data is processed, stored, and exchanged within a system. It helps stakeholders grasp the system's functionality and identify potential areas for improvement or optimization. Additionally, DFDs serve as valuable communication tools, facilitating discussions among stakeholders and aiding in the design and documentation of systems.



Processing Narrative:

A processing narrative offers a detailed textual account of how data moves through the system, elucidating the operations executed on the data at each stage. Here's an example to illustrate:

FOR EXAMPLE:
Processing Narrative for Online Shopping System

User Selection of Items: The process initiates when a user selects items to purchase from the online store. These selections are captured by the system as input data.

Validation of User Input: The system validates the user input to ensure that the selected items are available in the inventory and that the user has provided necessary details such as quantity, size, and color.

Calculation of Total Cost: After validation, the system calculates the total cost of the selected items, including any applicable taxes or discounts.

Payment Processing: Once the total cost is calculated, the system prompts the user to select a payment method. Upon selection, the system securely processes the payment transaction.

Update Inventory: After successful payment processing, the system updates the inventory to reflect the purchased items. It deducts the quantity of purchased items from the available stock.

Generation of Order Confirmation: Following the completion of the transaction, the system generates an order confirmation for the user. This confirmation includes details such as order number, items purchased, total cost, and estimated delivery date.

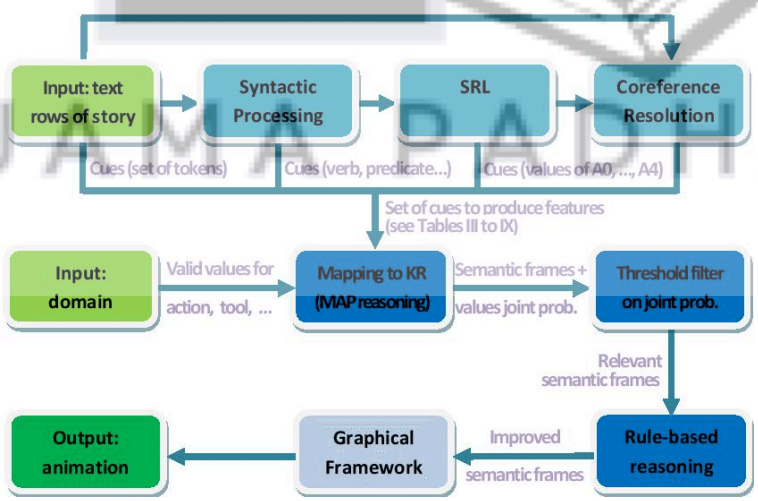
Notification to User and Shipping: Simultaneously, the system notifies the user about the successful transaction and initiates the shipping process. It provides tracking information to allow the user to monitor the status of their order.

Feedback Collection: After the order is delivered, the system solicits feedback from the user regarding their shopping experience and the quality of the products received.

Data Analysis and Reporting: Feedback collected from users is analyzed by the system to identify areas for improvement. Additionally, the system generates reports on sales performance, inventory status, and customer feedback for managerial review.

Archiving of Transaction Data: Finally, the system archives transaction data for record-keeping and future reference. This data may be used for accounting purposes, customer support inquiries, or market analysis.

This narrative provides a comprehensive overview of how data flows through the online shopping system, detailing the various processes and operations involved at each step.



Defining Transform Flow Characteristics:

A system characterized by distinct transform flow features comprises processes that alter data in significant and purposeful ways. These processes engage in activities like calculations, conversions, or other manipulations that modify the state or structure of the data. Here's an elucidation:

Distinct Transform Flow Characteristics in a Financial Analysis System

Data Aggregation: The system aggregates raw financial data from various sources such as transactions, market feeds, and historical records.

Calculation of Key Financial Metrics: Once aggregated, the system performs calculations to derive essential financial metrics such as revenue, expenses, profitability ratios, and growth rates.

Normalization and Standardization: To ensure consistency and comparability, the system normalizes and standardizes financial data across different time periods, currencies, or accounting standards.

Forecasting and Predictive Modeling: Utilizing statistical algorithms and historical data, the system generates forecasts and predictive models to anticipate future financial trends and outcomes.

Risk Assessment and Scenario Analysis: The system conducts risk assessments and scenario analyses by applying mathematical models to evaluate potential outcomes under different economic conditions or business scenarios.

Data Transformation for Reporting: The system transforms raw financial data into visually appealing and informative reports, charts, and graphs suitable for decision-making and presentation purposes.

Conversion of Currency and Units: In multinational operations, the system converts financial data from one currency to another and standardizes units of measurement to facilitate cross-border comparisons and analysis.

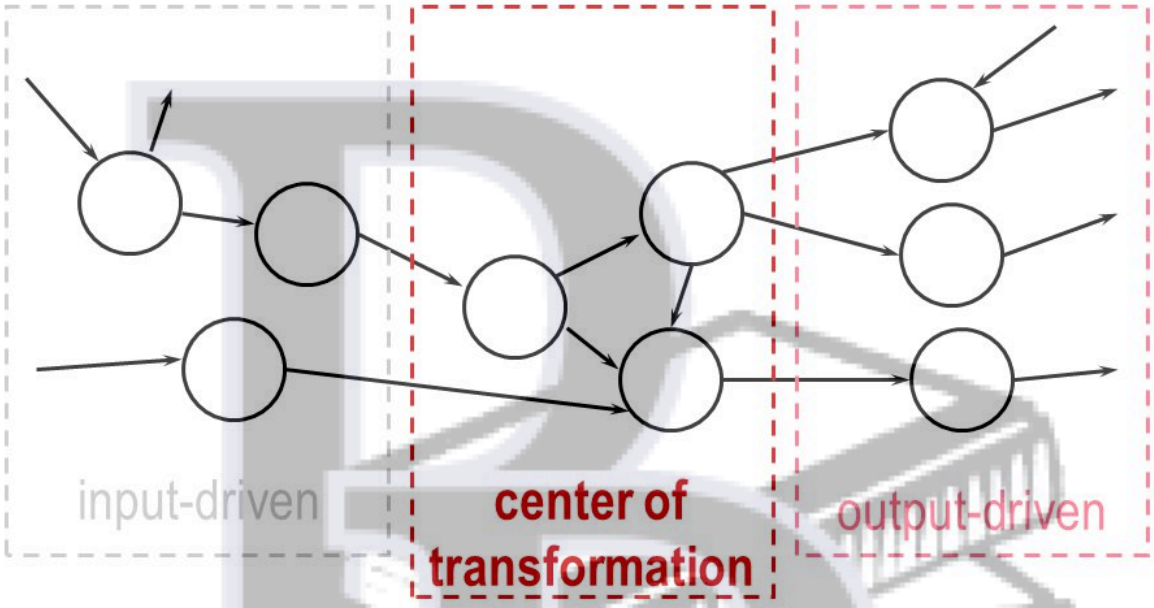
Performance Benchmarking: Through comparative analysis, the system benchmarks the financial performance of the organization against industry standards or competitors, identifying areas of strength and weakness.

Optimization Algorithms: Using optimization algorithms, the system recommends strategies to improve financial performance, minimize costs, or maximize returns on investment.

Auditing and Compliance Checks: The system conducts audits and compliance checks to ensure adherence to regulatory requirements and internal policies, flagging any discrepancies or anomalies for further investigation.

In this context, the system's distinct transform flow characteristics involve processes that manipulate financial data through calculations, conversions, forecasts, and other transformative operations, ultimately facilitating informed decision-making and strategic planning within the organization.

DFD with Transform Flow Characteristics



Flow Boundaries:

Flow boundaries serve to demarcate the extent of the system, delineating what components are considered internal versus external. This clarification aids in discerning the points at which data enters and exits the system. Here's an elaboration:

For example:
Flow Boundaries in an E-commerce Platform

User Interaction: The boundary encompasses interactions initiated by users within the e-commerce platform, including browsing products, adding items to the cart, and completing purchases.

External Data Sources: Data streams from external sources, such as supplier databases, market feeds, and payment gateways, are considered to cross the flow boundary into the system.

System Outputs: Information generated by the system, such as order confirmations, invoices, and shipping notifications, constitutes data exiting the system and crossing the flow boundary.

User Feedback and Reviews: User-generated content, such as product reviews, ratings, and feedback, is considered to flow across the boundary into the system, providing valuable insights for improvement.

Integration with Third-party Services: Interfaces with external services, such as social media platforms, analytics tools, and advertising networks, involve data exchange across the flow boundary.

Regulatory Compliance: Compliance with legal and regulatory frameworks necessitates monitoring and managing data flows across the boundary to ensure adherence to privacy, security, and consumer protection laws.

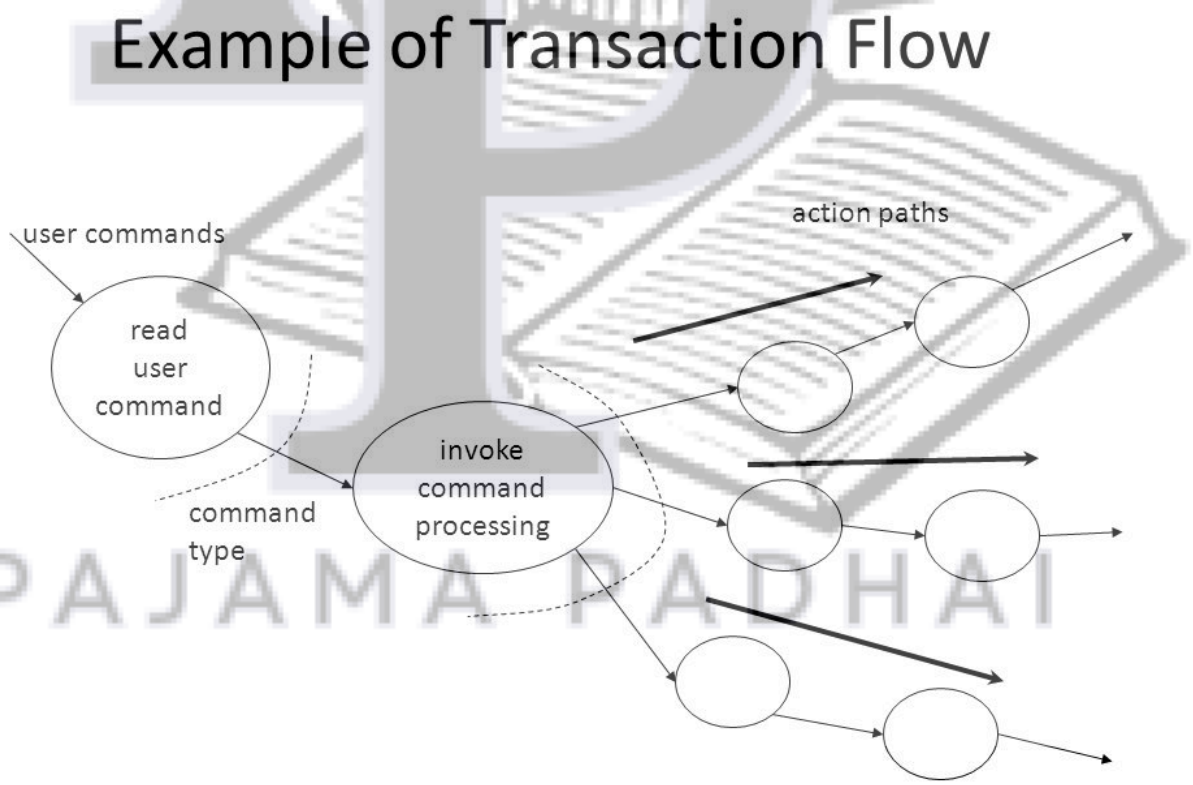
Customer Support Interactions: Communication channels for customer inquiries, complaints, and support requests involve data crossing the flow boundary into the system for resolution and response.

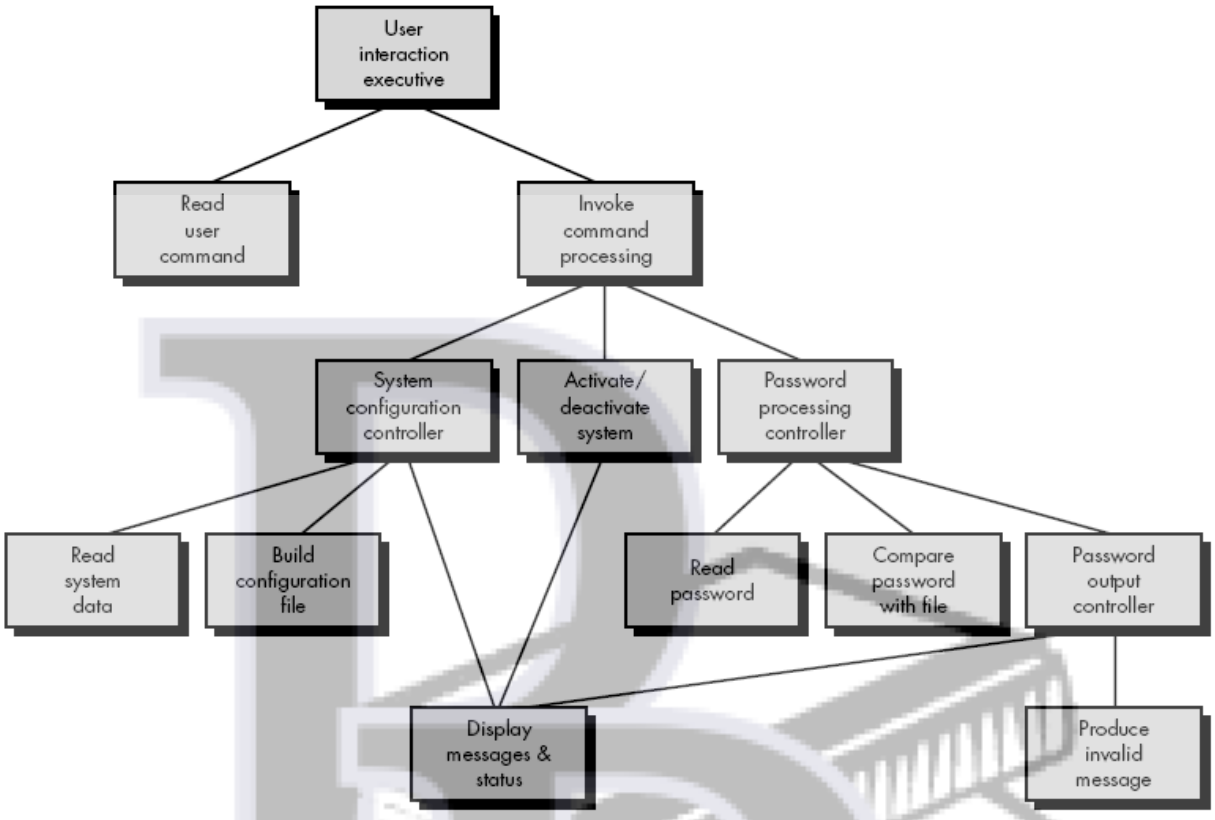
Inventory Management and Supplier Communications: Integration with inventory management systems and communication with suppliers involves data exchange across the flow boundary to maintain accurate stock levels and facilitate procurement processes.

Analytics and Reporting: Data collected within the system, including user behavior, sales trends, and performance metrics, is aggregated and analyzed to generate reports and insights, crossing the flow boundary for decision-making purposes.

System Updates and Maintenance: Processes related to system updates, maintenance, and upgrades occur within the boundary, ensuring the system's functionality, reliability, and security.

In this context, flow boundaries delineate the scope of the e-commerce platform, clarifying where data enters and exits the system from various internal and external sources, contributing to a comprehensive understanding of the system's interactions and dependencies.





Mapping DFD into Software Architecture:

Processes to Components/Modules:

Each process in the DFD represents a distinct functionality or operation performed on the data. These processes can be mapped to corresponding components or modules in the software architecture.
For example, a process responsible for user authentication and authorization might be mapped to a security module within the software architecture.
Data Stores to Data Storage Layers:

Data stores in the DFD represent repositories where data is stored within the system. These data stores can be mapped to specific data storage layers or databases in the software architecture.
For instance, a data store containing user profiles and preferences may be mapped to a user data storage layer in the software architecture.
Data Flows to Communication Channels:

Data flows in the DFD depict the movement of data between processes, data stores, and external entities. These data flows can be mapped to communication channels or interfaces in the software architecture.
For example, a data flow representing the transfer of order information from the user interface to the backend processing module may be mapped to a RESTful API or messaging queue.
External Entities to Interfaces or External Systems:

External entities in the DFD represent sources or destinations of data outside the system. These external entities can be mapped to interfaces or external systems integrated with the software architecture.

For instance, an external entity representing a payment gateway may be mapped to an API interface for payment processing within the software architecture.

Architectural mapping using Data flow

Structured design

Transform mapping

To map these data flow diagrams into a software architecture,you would initiate the following design steps:

1.Review the fundamental system model.

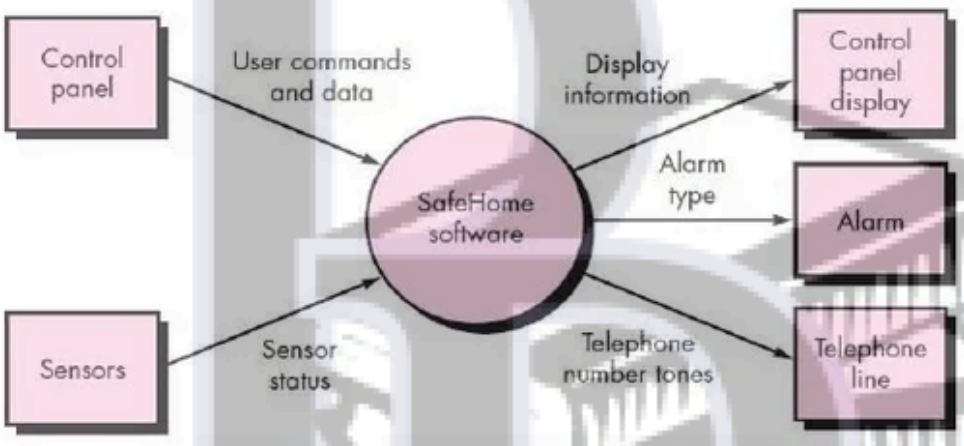


Fig 9.10 Context-level DFD for safeHome security function

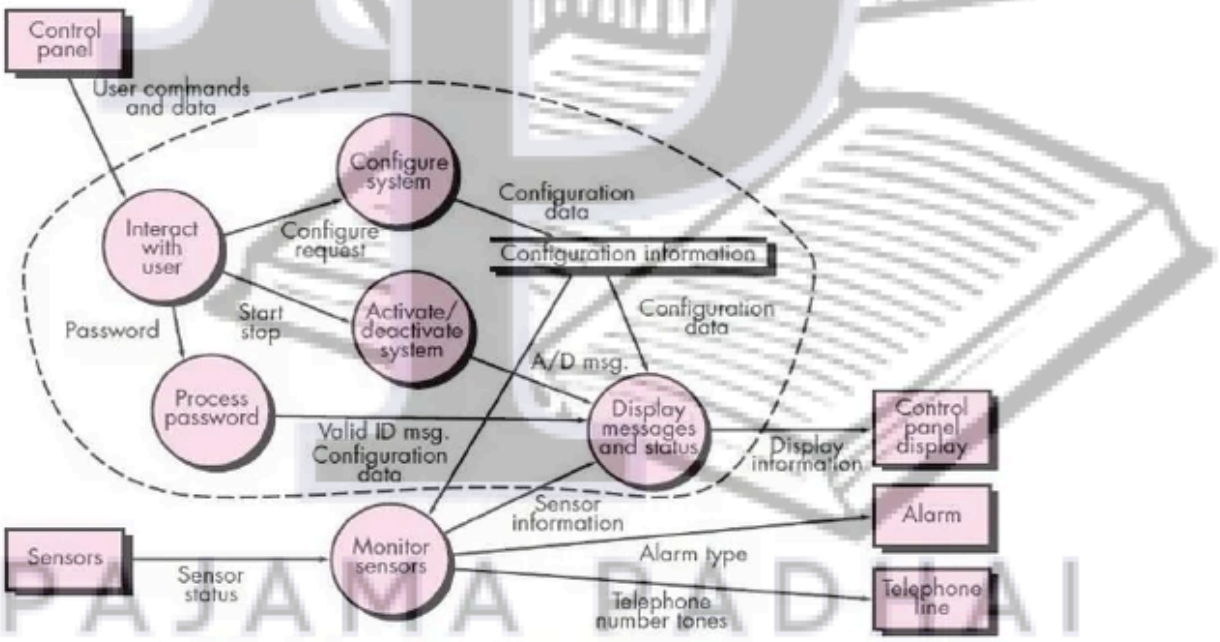
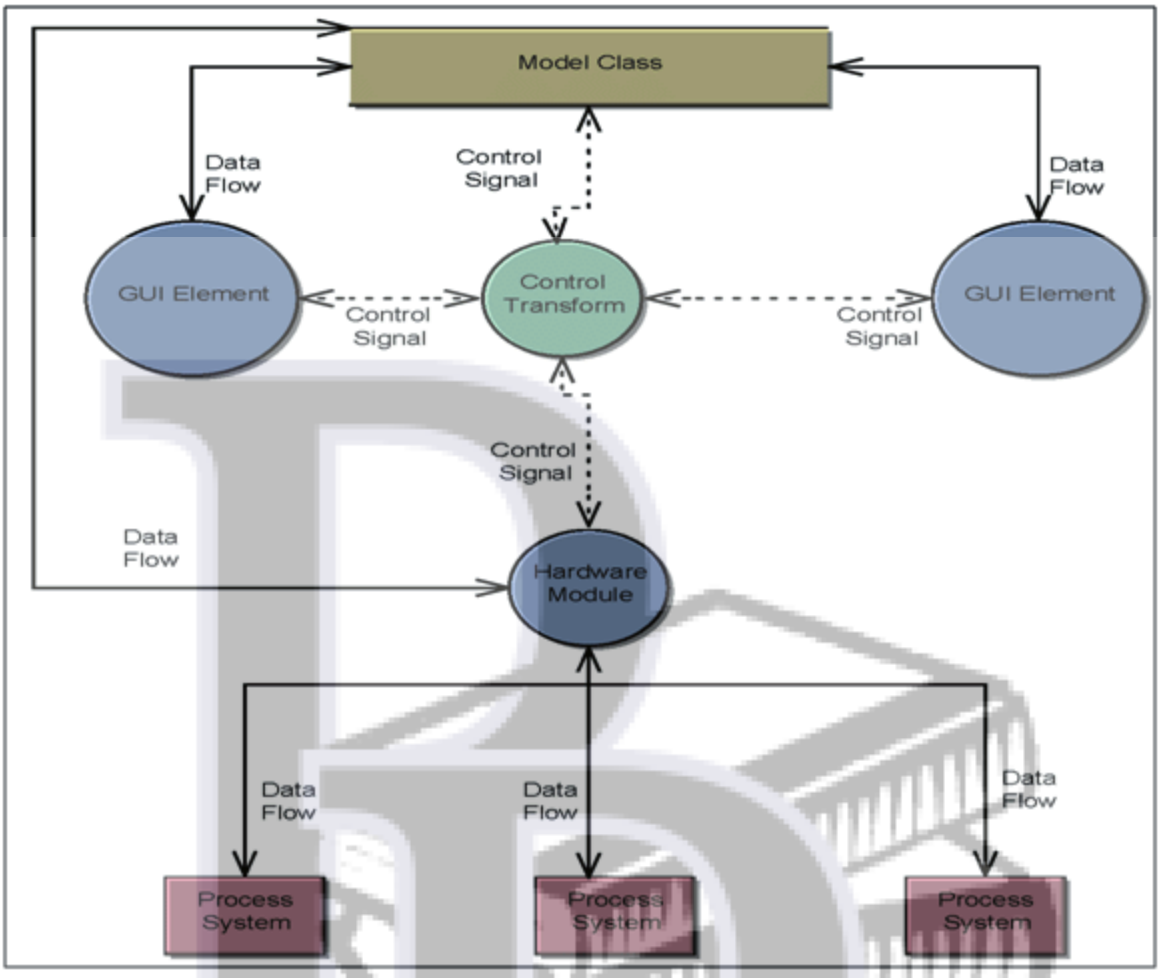


Fig 9.11: Level 1 DFD for safeHome security function



Example Mapping:

In a web-based e-commerce system:

The process of user authentication and authorization could be mapped to a security/authentication module.
Data stores containing product catalog information may be mapped to a product database layer.
Data flows representing user interactions with the shopping cart may be mapped to RESTful API endpoints handling cart management.
External entities representing payment gateways or shipping services may be mapped to corresponding external system interfaces.
By mapping DFD elements into the software architecture, developers can create a structured and organized system design that aligns with the system's functional requirements and facilitates efficient implementation and maintenance.

Now, let's create an example based on these principles:

System Description:
We'll consider a simple system for processing orders in an online bookstore.

Data Flow Diagram:

External Entity: Customer
Process: Order Processing
Data Store: Order Database
Processing Narrative:

Customer places an order by submitting order details through the website.
Order details are received by the Order Processing system.
The system validates the order details.
If the order is valid, the system updates the Order Database with the new order.
The system calculates shipping costs and tax.
The system generates an invoice.
The invoice is sent to the customer.
Transform Flow Characteristics:

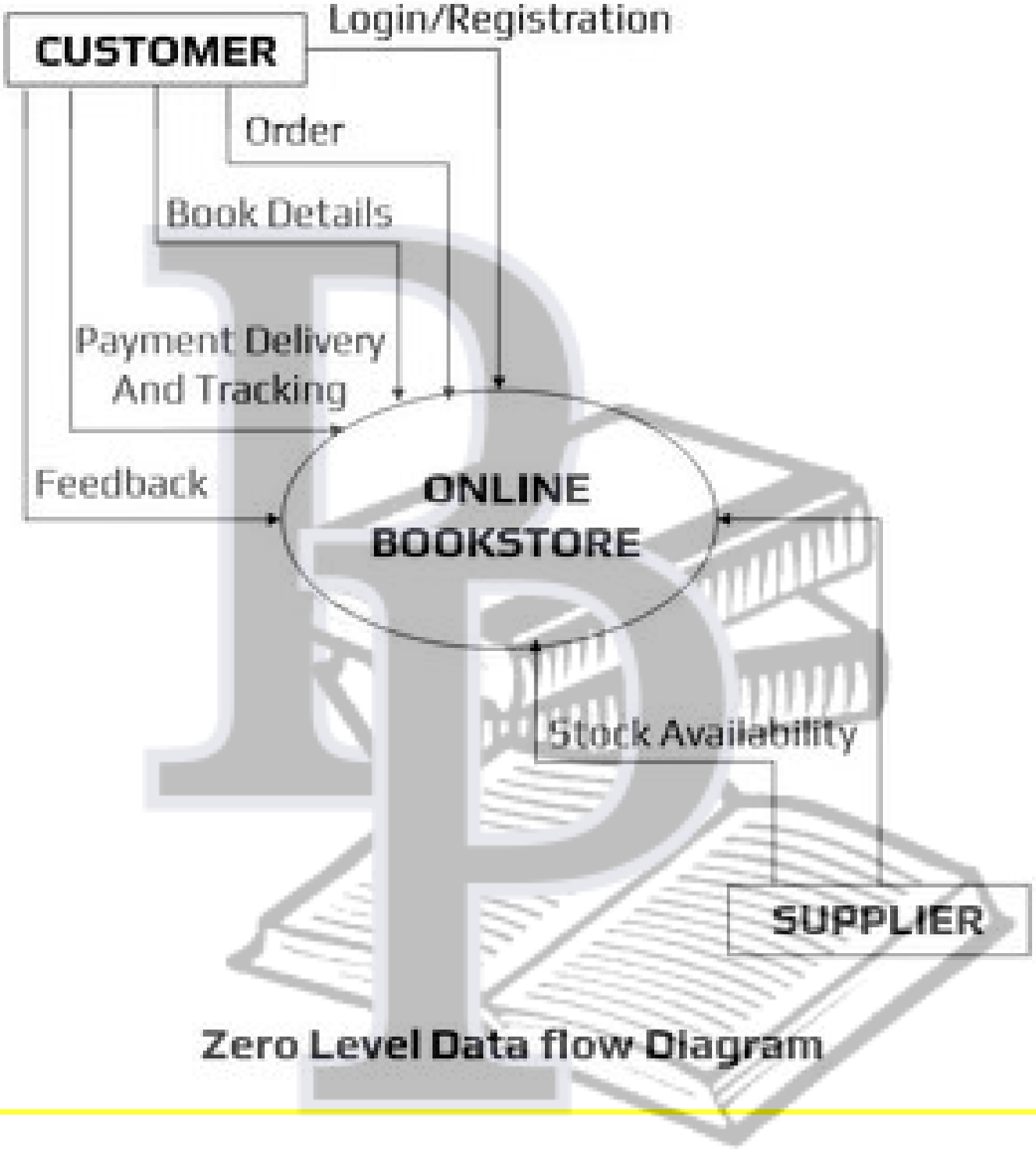
The system transforms raw order data into processed orders with calculated shipping costs and taxes, and generates an invoice for the customer.

Flow Boundaries:

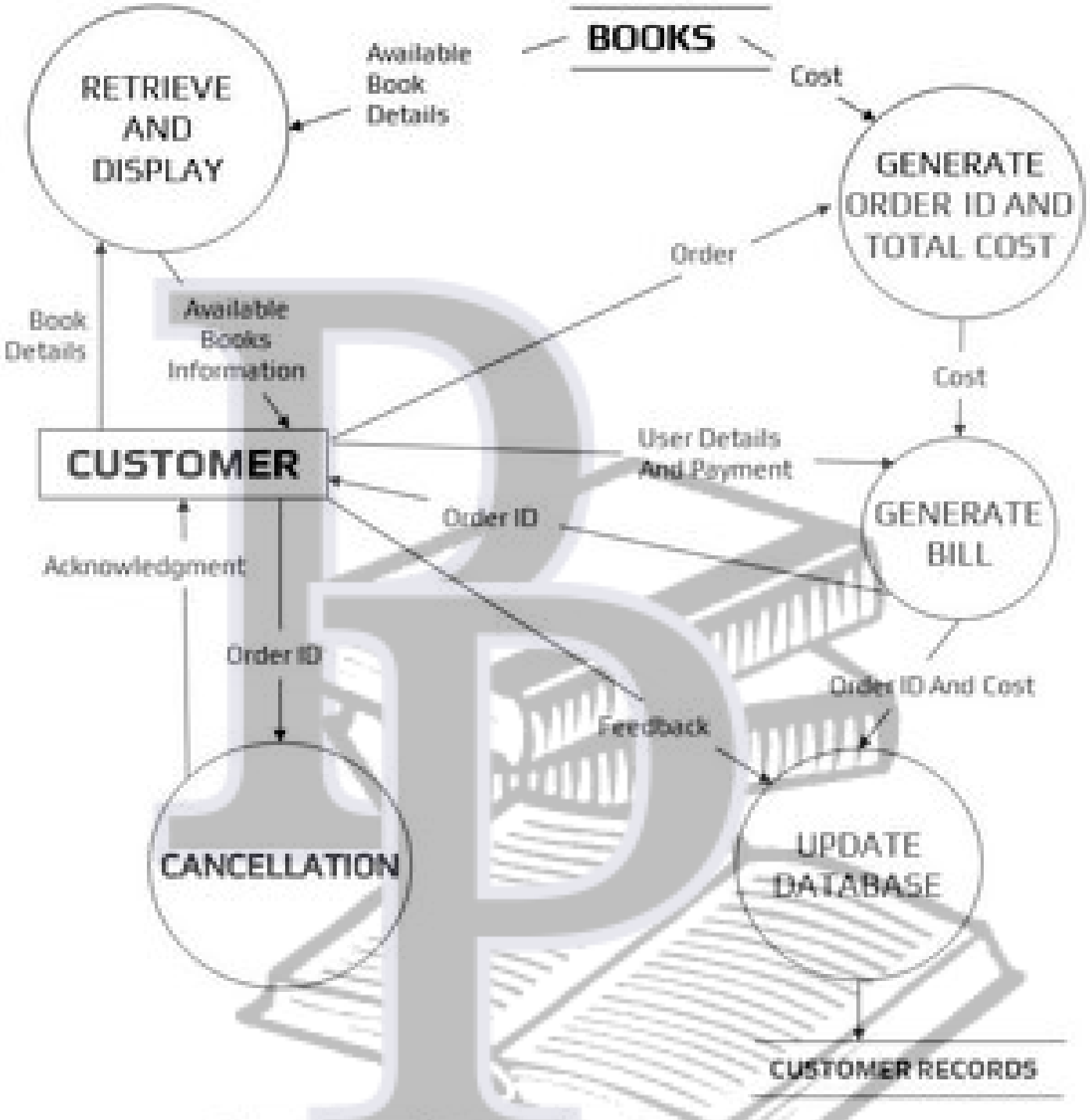
The system boundary includes all processes involved in order processing, the Order Database, and the interface with the customer. External to the system are any interactions with third-party services, such as payment processing.

Mapping DFD into Software Architecture:

The Order Processing process could be implemented as a module or component within the software architecture.
The Order Database could be represented by a database management system.
The validation, calculation, and invoice generation steps could be implemented as separate modules or functions within the system architecture.
Interfaces for customer interaction could be implemented as user interface components.
In conclusion, by creating a data flow diagram, processing narrative, defining transform flow characteristics, establishing flow boundaries, and mapping the DFD into software architecture, we've outlined a system for processing orders in an online bookstore.



Zero Level Data flow Diagram



First Level Data flow Diagram