EXPERIMENT-1 (ENTITY RELATIONSHIP DIAGRAMS)

AIM: To demonstrate Entity Relationship Diagrams.

DESCRIPTION: An Entity-Relationship Diagram (ERD) is a visual representation of the logical structure of a database. It is used to model the entities or objects within a system and the relationships between them. ERDs are crucial tools in database design and help in understanding the database's architecture, ensuring data integrity, and optimizing queries.
Here's a breakdown of the key components you'll find in an ER Diagrams:

1. Entities: An entity represents a real-world object or concept that can be easily identifiable. In a business context, entities can be things like customers, products, or orders.
2. Attributes: Attributes are the properties or characteristics of an entity. For example, a 'Customer' entity might have attributes like 'CustomerID', 'Name', and 'Email'.
3. Relationships: Relationships define how two entities relate to each other within the database. They can be one-to-one, one-to-many, or many-to-many. Relationships are represented by lines connecting the related entities and are labeled to indicate the nature of the relationship.
4. Cardinality: Cardinality describes the numerical relationships between entities in a relationship. It specifies how many instances of one entity are associated with instances of another entity.
5. Primary Key: A primary key is an attribute (or combination of attributes) that uniquely identifies each record in an entity. It ensures that each record within an entity is distinct.
6. Foreign Key: A foreign key is an attribute in a database table that references the primary key of another table. It establishes a link between the data in two tables, enforcing referential integrity.

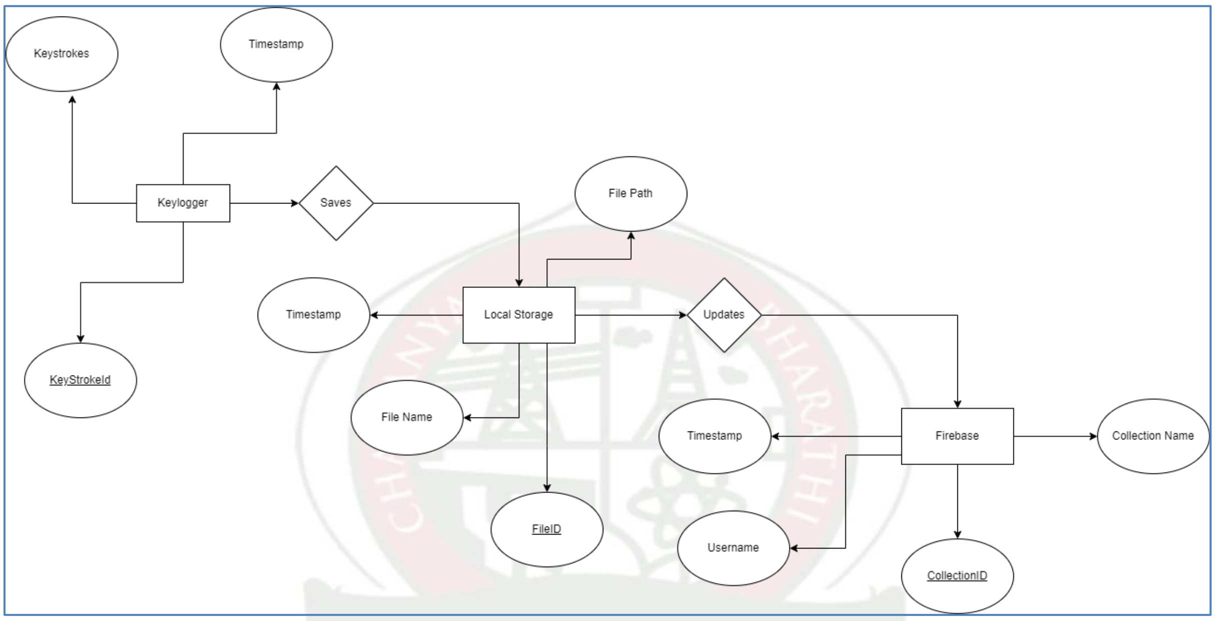
When creating an ER Diagrams, designers use symbols to represent these components:

1. Rectangles: Represent entities
2. Ovals: Represent attributes
3. Diamonds: Represent relationships
4. Lines: Connect entities and show relationships
5. Double lines: Indicate total participation (e.g., every instance of one entity must participate in the relationship

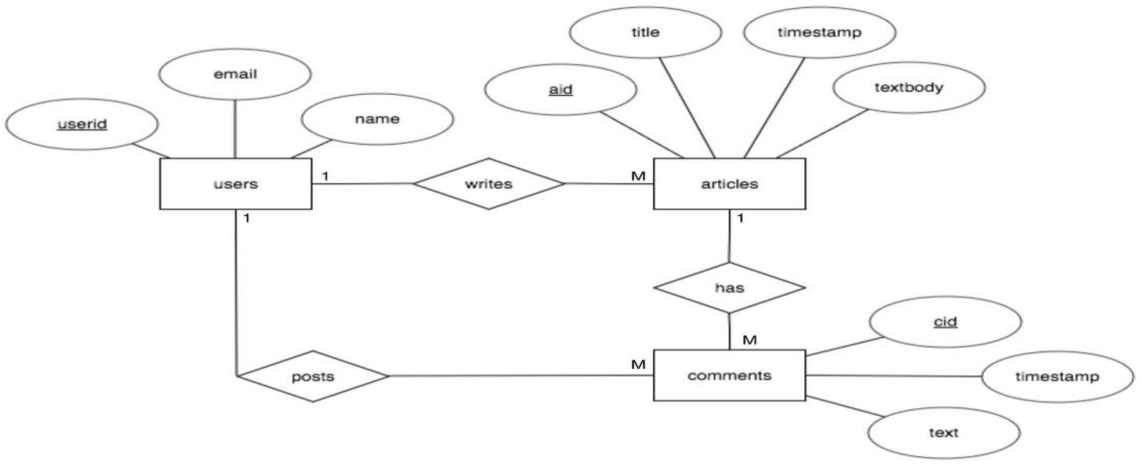
ER Diagrams can be created using various tools and methodologies, such as Crow’s Foot notation, Chen notation, or UML (Unified Modeling Language). The choice of notation often depends on the specific requirements of the project and personal or organizational preferences. In summary, an ERD provides a clear and concise visual representation of a database's structure, helping database designers, developers, and stakeholders understand and communicate the database design effectively.

DIAGRAMS:

ER Diagram for Advanced Keylogger



ER Diagram for Blog Website



EXPERIMENT-2 (USE CASE DIAGRAMS)

AIM: To demonstrate Use Case Diagrams in Unified Modelling Language (UML).

DESCRIPTION: A Use Case Diagram is a type of behavioral diagram in the Unified Modeling Language (UML) that illustrates the interactions between users (actors) and a system to achieve specific goals. It provides a high-level overview of the functionalities provided by a system and the actors involved in those functionalities.
Here are the key elements of a Use Case Diagram:

1. Actors: Actors represent the users or external systems interacting with the system. An actor can be a human user, another system, or even a hardware device. Actors are usually depicted as stick figures or blocks on the edges of the diagram.
2. Use Cases: Use cases represent the specific functionalities or tasks that the system performs to achieve the goals of the actors. They describe the interaction between the system and its users to accomplish a particular goal. Use cases are typically represented as ovals or ellipses within the diagram.
3. Relationships: Relationships between actors and use cases show which actors are involved in each use case. A solid line connecting an actor to a use case indicates that the actor participates in that use case.
4. System Boundary: The system boundary, often represented as a rectangle, encloses all the use cases of the system. It defines the scope of the system being modeled.

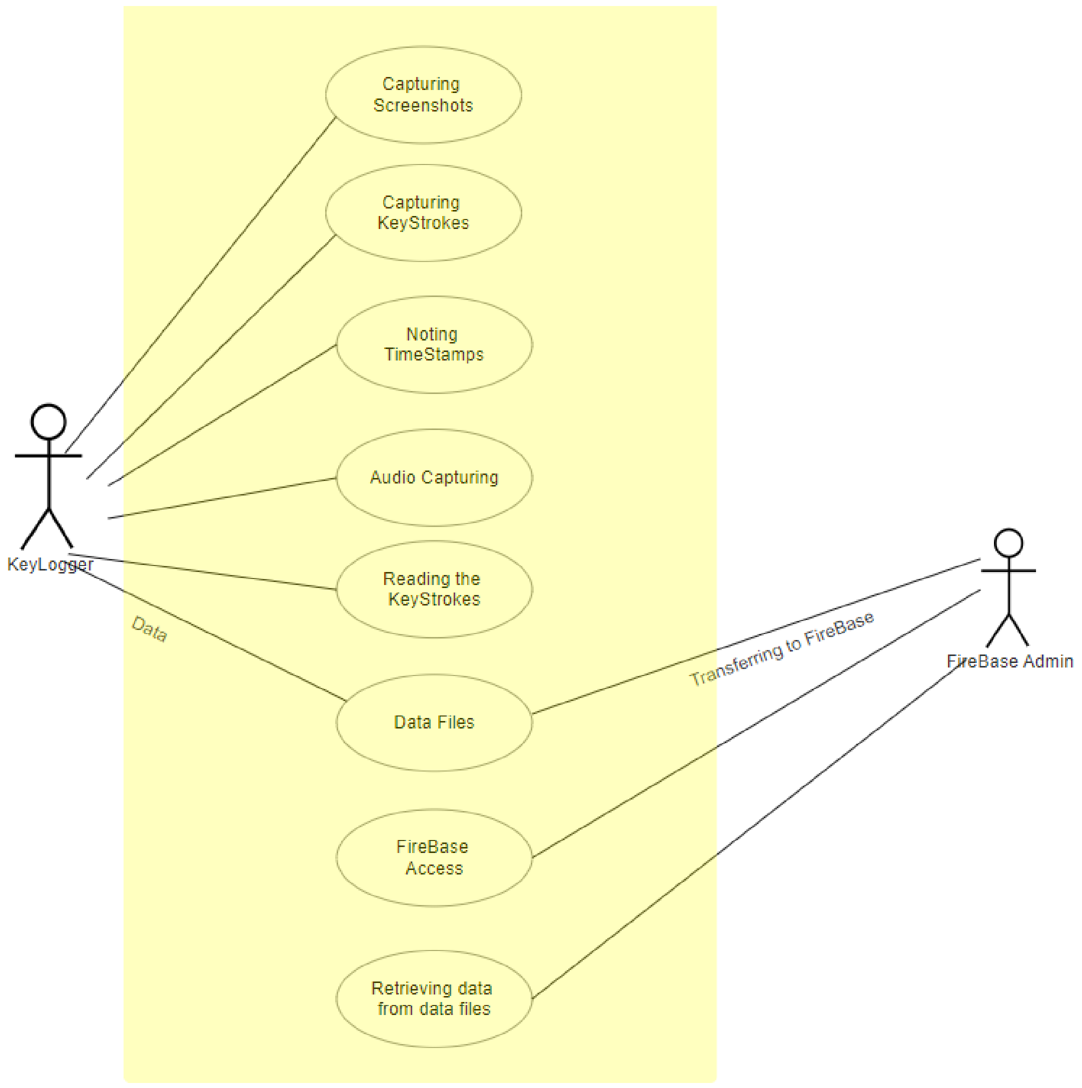
Use Case Diagrams are beneficial for various reasons:

- Understanding System Requirements: They help stakeholders, including developers, designers, and clients, understand the system's functionalities and requirements from a user's perspective.
- Communication: Use Case Diagrams serve as a visual communication tool that bridges the gap between technical and non-technical stakeholders, ensuring everyone has a shared understanding of the system.
- Validation: They aid in validating system requirements by visualizing the interactions between users and the system, helping to identify potential gaps or ambiguities in the requirements.
- System Design: Use Case Diagrams can guide system design by identifying the key functionalities and interactions that need to be implemented, providing a foundation for more detailed design and development activities.

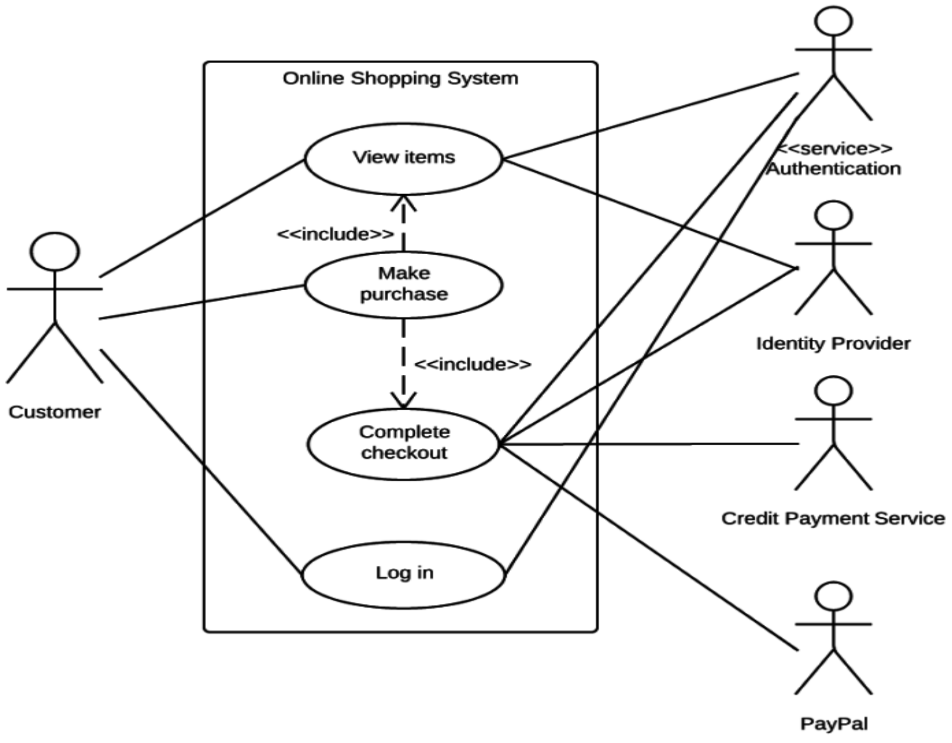
In summary, a Use Case Diagram is a valuable tool in the software development process, helping to visualize and clarify the system's functionalities, interactions, and requirements from a user-centric perspective.

DIAGRAMS:

Use Case Diagram for Advanced KeyLogger



Use Case Diagram for Online Shopping System



EXPERIMENT-3 (DATA FLOW DIAGRAMS)

AIM: To demonstrate Data Flow Diagrams (DFD).

DESCRIPTION: A Data Flow Diagram (DFD) is a visual representation that illustrates the flow of data within a system. It depicts how data moves between different components of a system, including external entities, processes, and data stores. DFDs provide a clear and concise overview of the system's data flow and interactions. They help in understanding the system's functionalities, requirements, and architecture. DFDs are essential tools in systems analysis and design, aiding in requirement analysis, system design, implementation, and testing phases of the Software Development Lifecycle (SDLC). DFDs come in different levels, such as Level 0 (Context Diagram), Level 1, Level 2, etc., each providing varying levels of detail. The components of a DFD include External Entities, Processes, Data Stores, and Data Flows. External Entities represent sources or destinations of data outside the system. Processes depict the activities or functions performed by the system. Data Stores are repositories where data is stored, and Data Flows show the movement of data between components. DFDs assist in identifying data sources, transformations, and sinks in a system. They help in identifying potential bottlenecks, redundancies, or inefficiencies in data flow and processes. DFDs are used across various industries and domains, including software development, business analysis, and system architecture. They aid in designing scalable, efficient, and robust systems by providing insights into data flow patterns and interactions. DFDs can be manually drawn or created using specialized software tools, making them versatile and adaptable to different project requirements.
Here are the key components of a Data Flow Diagram:

1. External Entities: External entities represent external sources or destinations of data, such as users, systems, or organizations that interact with the system. They are usually depicted as squares on the boundaries of the diagram.
2. Processes: Processes represent the activities or transformations that occur within the system. They describe how data is input, manipulated, and output by the system. Processes are typically depicted as circles or ovals within the diagram.
3. Data Stores: Data stores represent repositories where data is stored within the system. They can be databases, files, or any other storage medium. Data stores are usually depicted as two parallel lines.
4. Data Flows: Data flows represent the movement of data between external entities, processes, and data stores. They indicate the direction in which data is transferred and are depicted as arrows.

Types of Data Flow Diagrams:

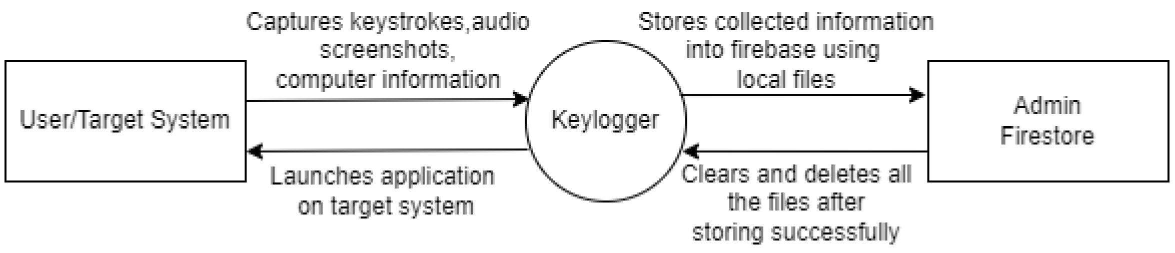
- Level 0 DFD (Context Diagram): This is the highest level of a DFD and provides a broad overview of the system. It shows the interactions between the system and external entities without detailing the internal processes.
- Level 1 DFD: This level provides a more detailed view of the system by breaking down the processes of the context diagram into sub-processes. It helps in understanding the internal workings of the system.
- Level 2 DFD, Level 3 DFD, etc: These levels further decompose the processes of the previous level into more detailed sub-processes, providing a deeper understanding of the system's functionalities and data flow.

Benefits of Data Flow Diagrams:

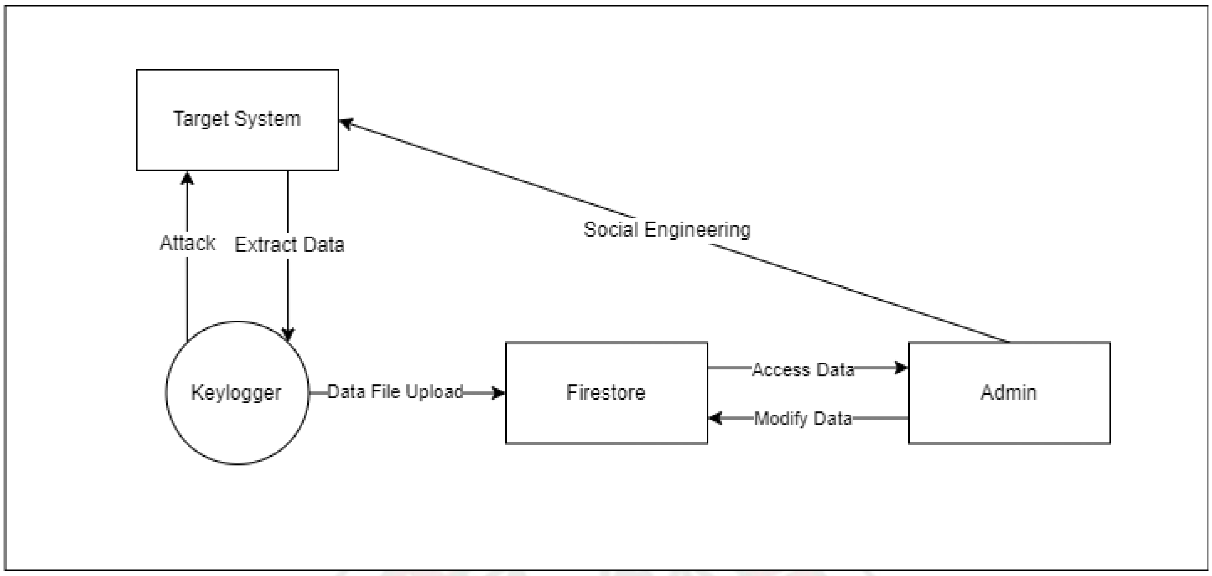
- Understanding and Analysis: DFDs help stakeholders understand the system's data flow and processes, facilitating analysis, and identifying areas for improvement.
- Communication: DFDs serve as a visual communication tool that helps in conveying complex system information in a clear and concise manner to both technical and non-technical stakeholders.
- System Design: DFDs aid in designing the system by providing a blueprint of the data flow and processes, guiding the development team in implementing the system.

DIAGRAM:

DFD Level 0 for Advanced Keylogger


Captures keystrokes,audio screenshots, computer information
Stores collected information into firebase using local files
User/Target System
Keylogger
Admin Firestore
Launches application on target system
Clears and deletes all the files after storing successfully

DFD Level 1 for Advanced Keylogger



EXPERIMENT-4 (CLASS DIAGRAMS)

AIM: To demonstrate Class Diagrams in Unified Modelling Language (UML).

DESCRIPTION: Class Diagrams are one of the most widely used types of diagrams in the Unified Modeling Language (UML) to visualize the structure of a system by modeling its classes, their attributes, operations or methods, and the relationships between classes.
Key Components of Class Diagrams:

1. Classes:
   - Definition: Classes represent the blueprint or template for creating objects in object-oriented programming. They encapsulate data (attributes) and behaviors (methods) related to a particular entity or concept.
   - Representation: In a class diagram, classes are depicted as rectangles with three compartments: the top compartment contains the class name, the middle compartment contains attributes, and the bottom compartment contains methods.

2. Attributes:
   - Definition: Attributes represent the properties or characteristics of a class. They describe the state of an object and are defined within a class.
   - Representation: Attributes are listed in the middle compartment of a class rectangle in a class diagram.

3. Methods (Operations):
   - Definition: Methods, also known as operations, represent the behaviors or actions that a class can perform. They define how objects interact and manipulate their data.
   - Representation: Methods are listed in the bottom compartment of a class rectangle in a class diagram.

4. Relationships:
   - Association: Represents a bi-directional relationship between two classes, indicating that instances of one class are linked to instances of another class.
   - Aggregation: Represents a "whole-part" relationship between classes, where one class is a part of another class but can exist independently.
   - Composition: Represents a stronger form of aggregation, where the part cannot exist without the whole.

- Generalization (Inheritance): Represents an "is-a" relationship between a superclass (parent) and subclass (child), indicating that the subclass inherits attributes and behaviors from the superclass.
- Dependency: Represents a weaker relationship where one class depends on another class, usually through method parameters or return types.

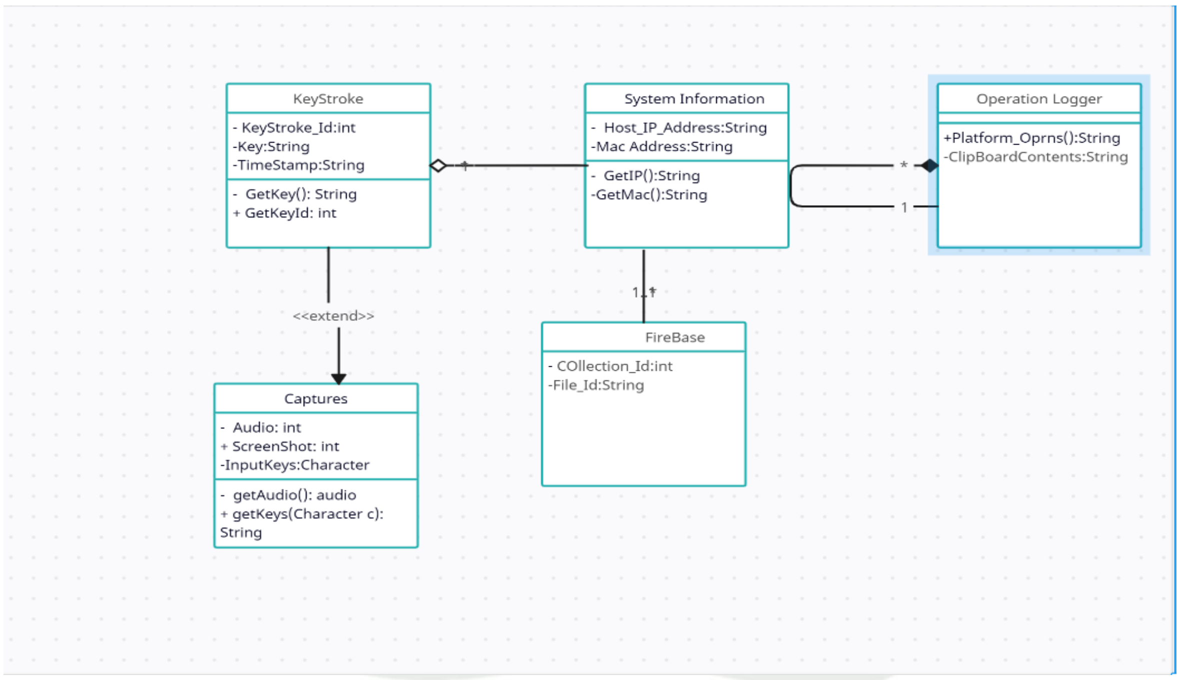
5. Multiplicity:
- Definition: Multiplicity defines the number of instances of one class that can be associated with instances of another class in a relationship.
- Representation: Multiplicity is indicated near the end of an association line using numbers, such as 1, 0..1, 0..*, 1..*, etc., specifying the minimum and maximum number of instances.

Benefits of Class Diagrams:
- System Design: Class Diagrams serve as a blueprint for system design, helping designers and developers visualize and organize the system's structure, relationships, and interactions.
- Code Generation: They assist in generating code from the model, ensuring consistency and reducing manual coding errors.
- Documentation: Class Diagrams act as documentation tools, capturing the system's architecture, relationships, and design decisions, facilitating future maintenance and enhancements.
- Communication: They serve as a communication tool between stakeholders, including developers, designers, testers, and business analysts, ensuring a shared understanding of the system's design and requirements.

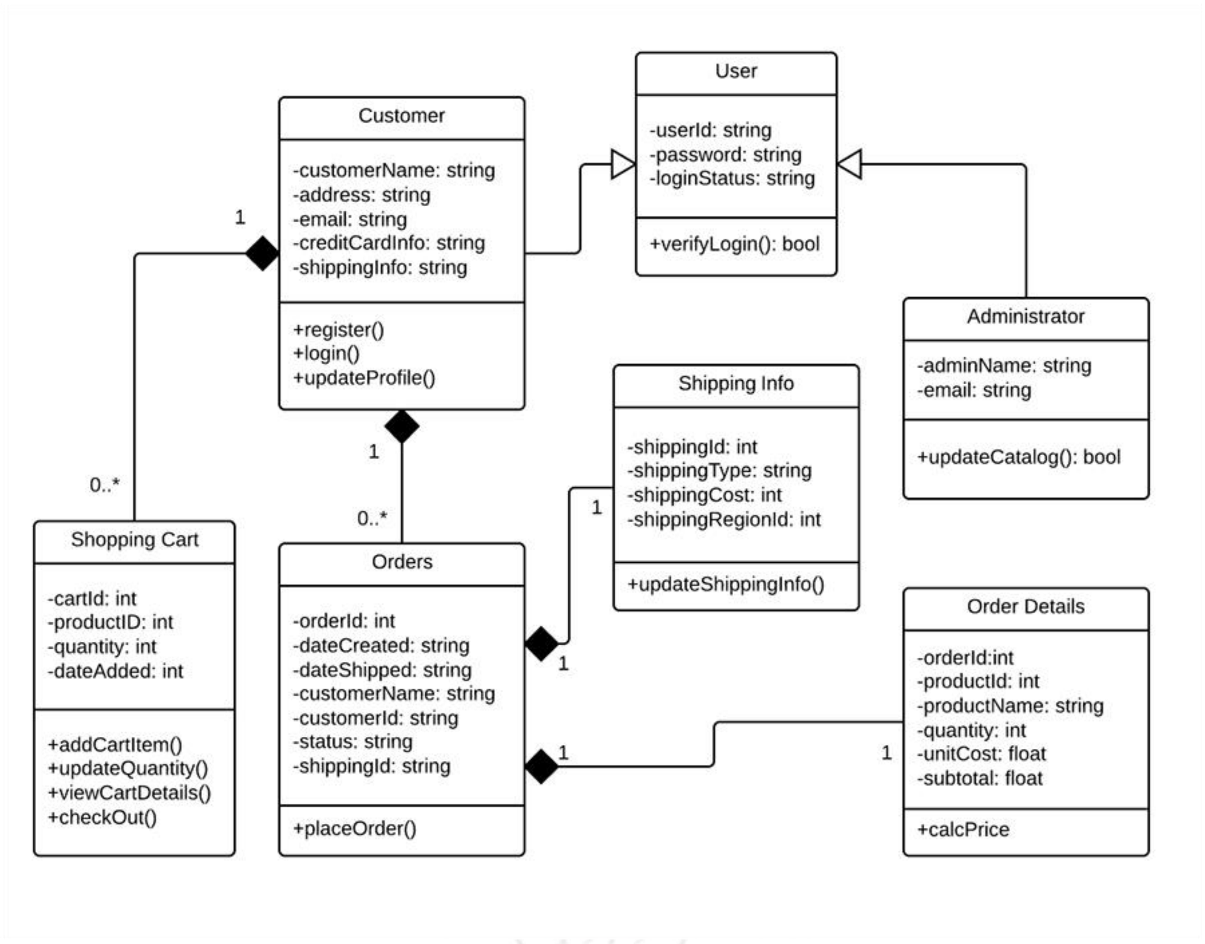
Usage in Software Development:
- Requirement Analysis: Class Diagrams aid in capturing and analyzing system requirements, identifying classes, attributes, and relationships based on the user's needs and functionalities.
- Implementation: They guide the implementation phase by providing a clear structure and design for developers to follow, ensuring the system's consistency and maintainability.
- Testing: Class Diagrams help testers understand the system's structure, relationships, and interactions, enabling effective test case design and validation.

In summary, Class Diagrams are essential tools in object-oriented analysis and design, providing a visual representation of a system's structure and relationships. They facilitate understanding, communication, design, implementation, and documentation of complex software systems, ensuring clarity, consistency, and scalability throughout the software development lifecycle.

DIAGRAM:
Class Diagram for Advanced Keylogger



Class Diagram for Online Shopping System



EXPERIMENT-5 (SEQUENCE DIAGRAMS)

AIM: To demonstrate Sequence Diagram in Unified Modelling Language (UML).

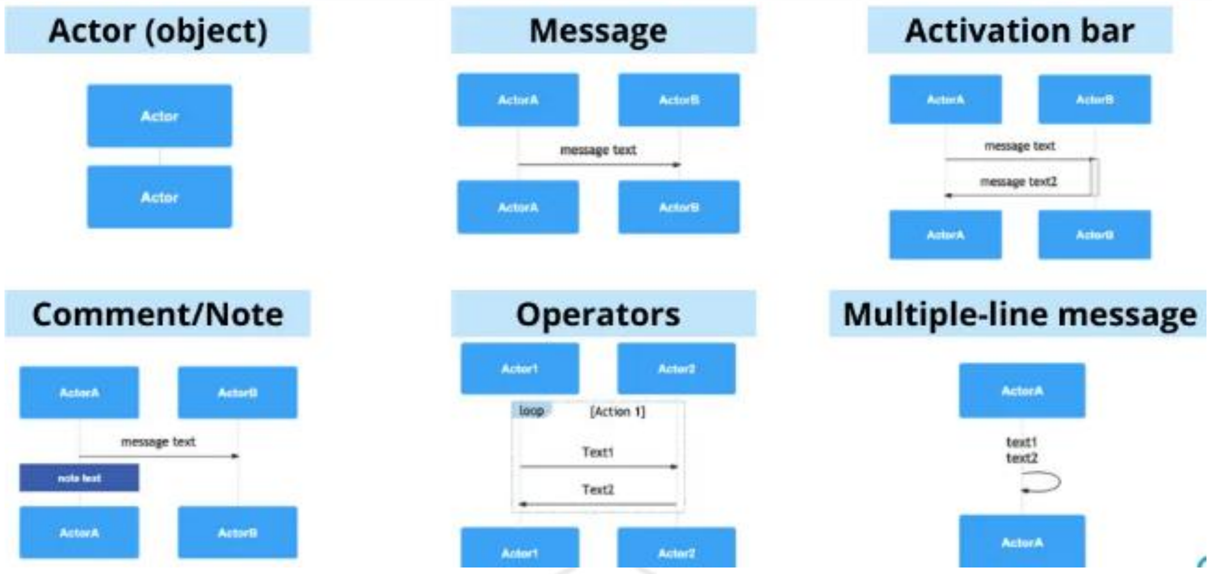
DESCRIPTION: Sequence diagrams are a type of interaction diagram that describe how objects interact in a particular scenario of a business or a software system. They are a key component of the Unified Modeling Language (UML), a standardized modeling language used in software engineering and systems design. Here are the key aspects of sequence diagrams:

Purpose
Sequence diagrams are used to visualize the sequence of messages exchanged between objects in order to carry out a specific functionality or a process. They are particularly useful for:

- Understanding the flow of a process or operation within the system.
- Designing the interactions between components.
- Documenting the existing systems or software architecture.
- Clarifying complex sequences of events and ensuring all possible scenarios are accounted for.

Components

- Actors: Represent external entities that interact with the system (e.g., users, other systems).
- Objects: Instances of classes or other entities that participate in the interaction.
- Lifelines: Represent the existence of an object over time. Lifelines are depicted as vertical dashed lines.
- Activation Bars: Show the time period during which an object is performing an action.
- Messages: Horizontal arrows that indicate communication between objects. They can represent method calls, returns, or signals.
- Synchronous Messages: Indicate that the sender waits for the recipient to process the message and return control.
- Asynchronous Messages: Indicate that the sender does not wait for the recipient to process the message.
- Frames: Enclose a section of the sequence diagram to represent conditional or iterative behavior.



Benefits

- Clarity: Provides a clear visual representation of interactions over time.
- Communication: Helps in communication between stakeholders, such as developers, analysts, and business users.
- Error Detection: Aids in identifying potential issues and inefficiencies in the process.
- Documentation: Serves as a part of comprehensive system documentation.

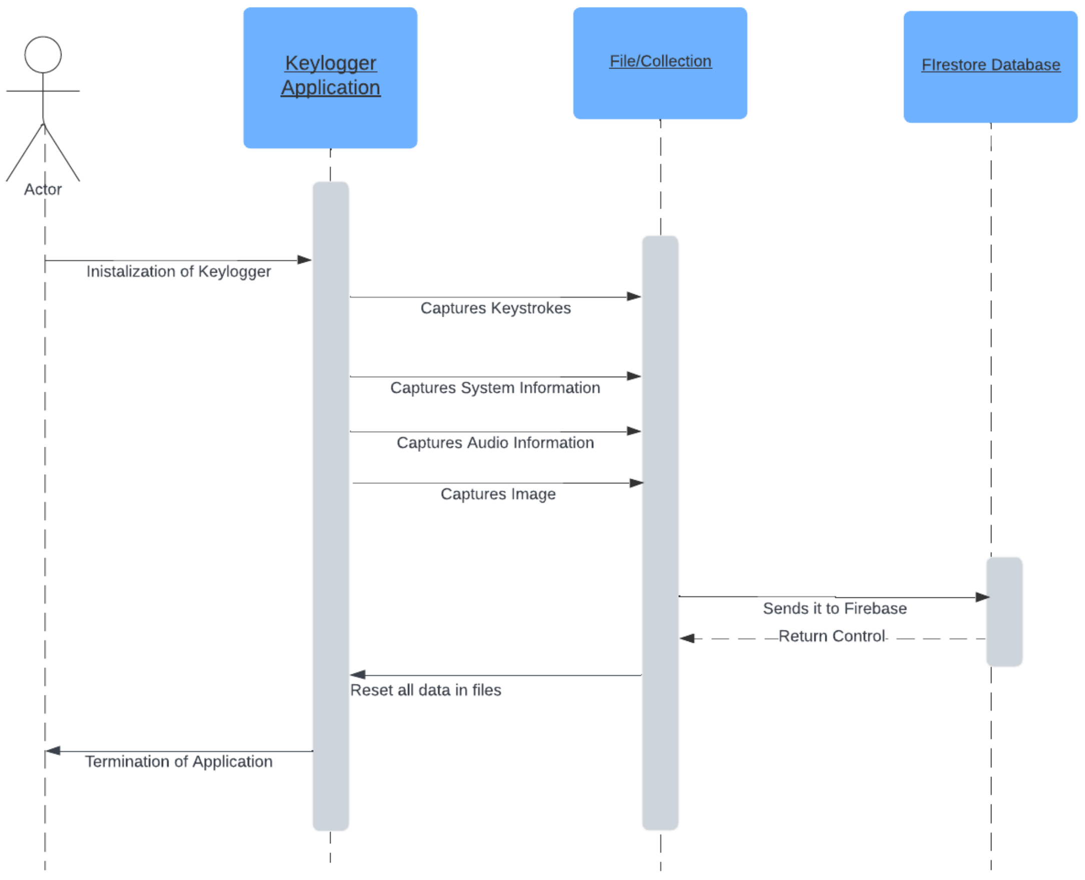
Limitations

- Complexity: Can become complex and difficult to manage for large systems with many interactions.
- Maintenance: Requires updating along with the system to remain accurate and useful.

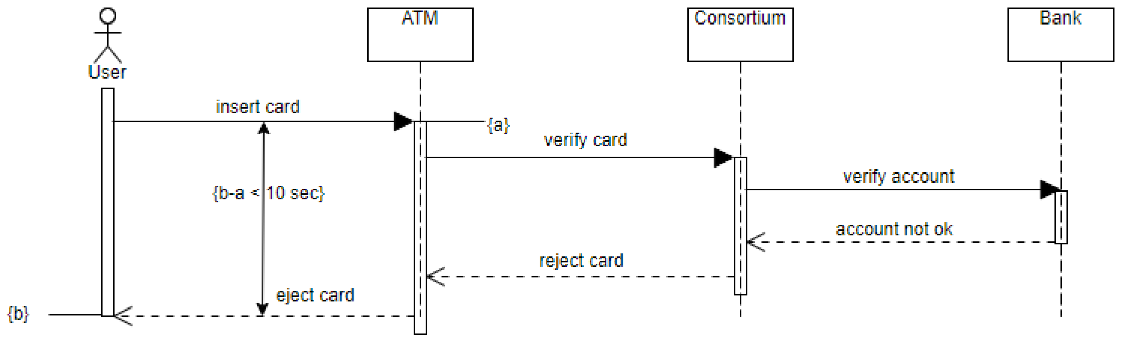
In summary, Sequence diagrams are an essential tool in the toolkit of system designers and software engineers, enabling them to design, analyze, and communicate the dynamic aspects of systems effectively.

DIAGRAMS:

Sequence Diagram for Advanced Keylogger



Sequence Diagram for ATM transfer



EXPERIMENT-6 (STATE DIAGRAMS)

AIM: To demonstrate State Diagrams in Unified Modelling Diagram (UML).

DESCRIPTION: State diagrams, also known as state machine diagrams, are powerful tools in UML for visualizing the dynamic behavior of an object or system. They depict the various states an entity can exist in and the events that trigger transitions between those states. Here's a detailed breakdown:

Components:

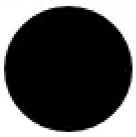
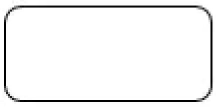
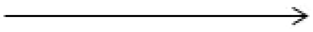
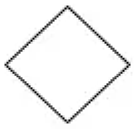
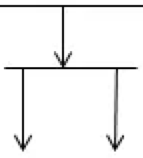
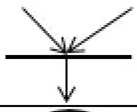
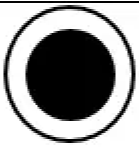
- States: Represented by rounded rectangles, states signify distinct conditions or situations the system can be in. Each state has a unique name that reflects its purpose.
- Transitions: Transitions are directed arrows connecting states and indicate the events that cause a state change. They are labeled with the triggering event and optionally, actions performed during the transition. These actions can modify the system's attributes or perform specific operations.
- Initial State: A solid black circle denotes the starting point of the system's execution. There can only be one initial state per diagram.
- Final State (Optional): A state marked with a bull's-eye symbol represents the system's termination point. Final states are optional, and a diagram can have zero or more final states.
- Guards (Optional): Guards are Boolean expressions attached to transitions that determine whether a transition can be triggered based on a specific condition. They are written within square brackets `[guard_condition]` alongside the event name on the transition label.
- Activities (Optional): Activities are actions performed within a state, typically for a specific duration. They are denoted by text enclosed in curly braces `{activity_name}` within the state rectangle.

Benefits of using State Diagrams:

- Clarity and Communication: State diagrams offer a visual representation of the system's behavior, simplifying communication and understanding between developers, designers, and stakeholders.

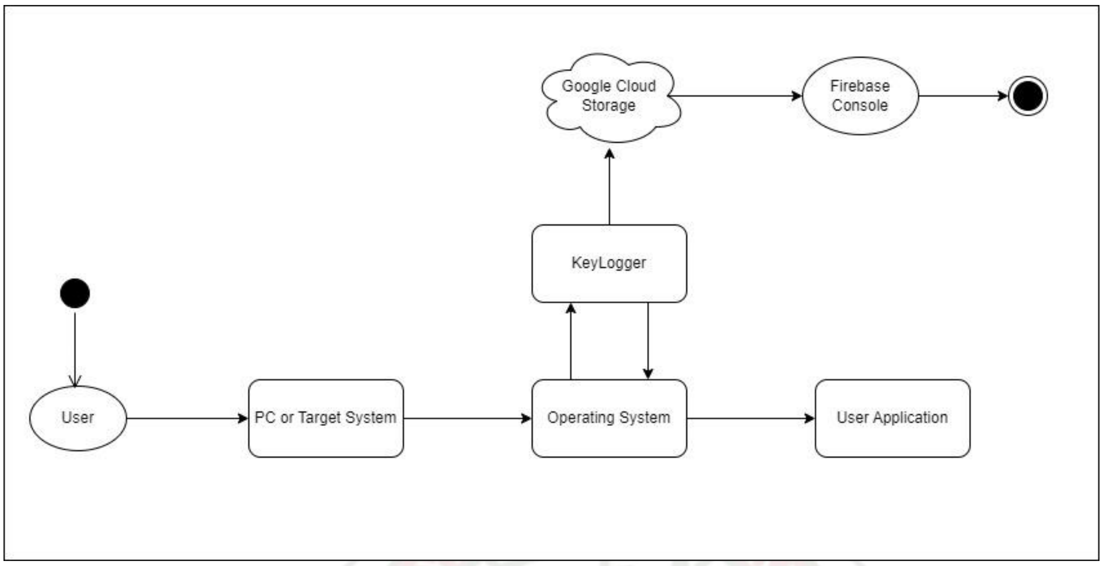
- Early Error Detection: By modeling different states and transitions, you can identify potential issues like unreachable states, unintended state changes, or missing transitions, leading to earlier error detection and improved system design.
- Improved Maintainability: State diagrams serve as a form of documentation, making it easier to understand existing code and maintain the system over time.

State Diagrams can get complex with many states and nesting, have limited error handling focus, and might not capture all system aspects. Techniques like decomposition and using other UML diagrams can help address these limitations.

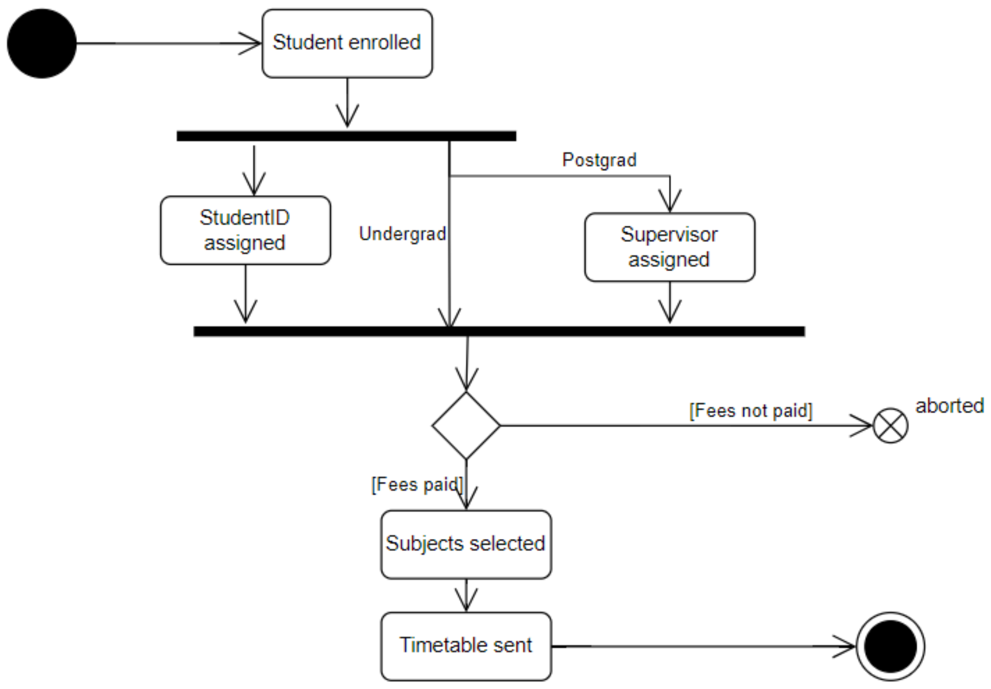
Name	Symbol
Start Node	
Action State	
Control Flow	
Decision Node	
Fork	
Join	
End State	

DIAGRAMS:

State Diagram for Advanced Keylogger



State Diagram for Student Enrollment System



EXPERIMENT-7 (ACTIVITY DIAGRAMS)

AIM: To demonstrate Activity Diagrams in Unified Modelling Diagram (UML).

DESCRIPTION: Activity diagram in UML is a type of behavioural diagram that visually portrays the flow of actions within a system. It's essentially a flowchart on steroids, offering a more powerful way to depict the system's dynamic behaviour.

Purpose:

- To model the sequential or concurrent activities performed by a system in response to a stimulus.

Elements:

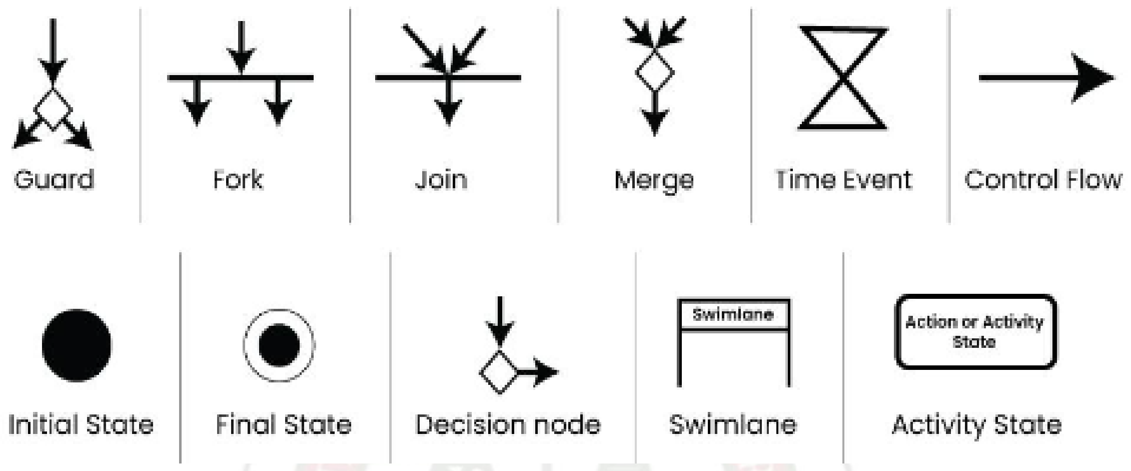
- Activities: Represented by rounded rectangles, they signify actions or processes the system performs.
- Transitions: Arrows indicating the flow of control between activities, often triggered by events or completion of the previous activity.
- Additional elements:
    - Swimlanes: Divide the diagram horizontally to group activities based on the performers (e.g., actors, objects).
    - Decision/Merge Nodes: Diamond shapes representing decision points with conditional branching and rejoining of execution paths.
    - Forks/Joins: Depict splitting or merging of concurrent activities.
    - Object Flows (Optional): Represent the flow of objects between activities, providing an object-oriented perspective.

Benefits:

- Clear Workflow Visualization: Provides a step-by-step view of the system's actions, making it easier to understand complex processes.
- Improved Communication: Facilitates communication between stakeholders regarding the system's functionality.
- Modelling Parallelism: Can represent concurrent activities happening simultaneously.

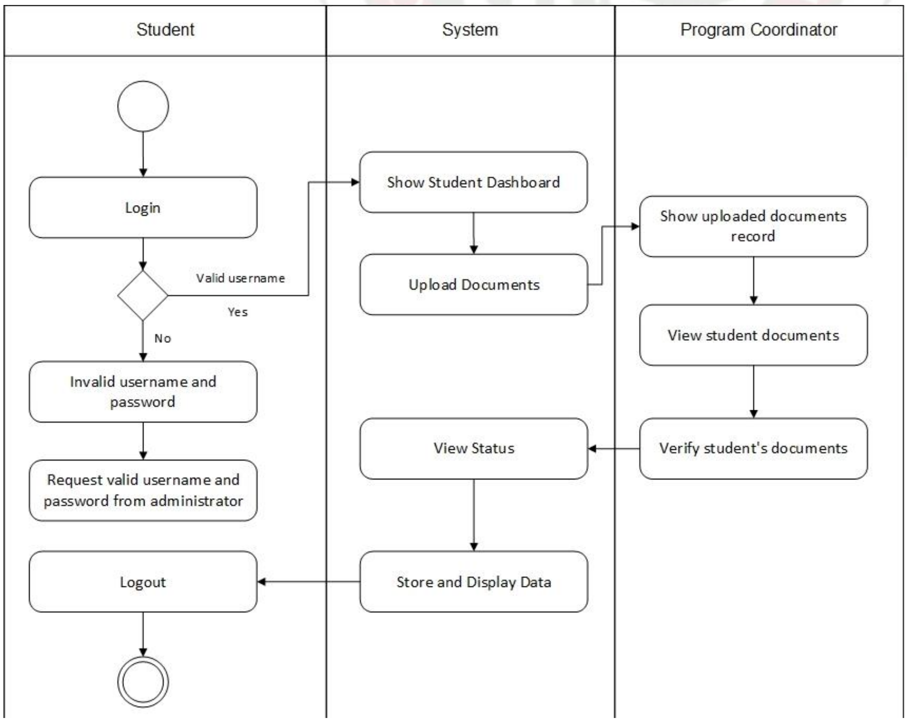
Use Cases include modelling workflows, business processes, use cases, and algorithms.

In summary, activity diagrams are a valuable tool for understanding and documenting the flow of actions within a system, promoting clear communication and effective system design.



DIAGRAMS:

Activity Diagram for Student Learning Management System



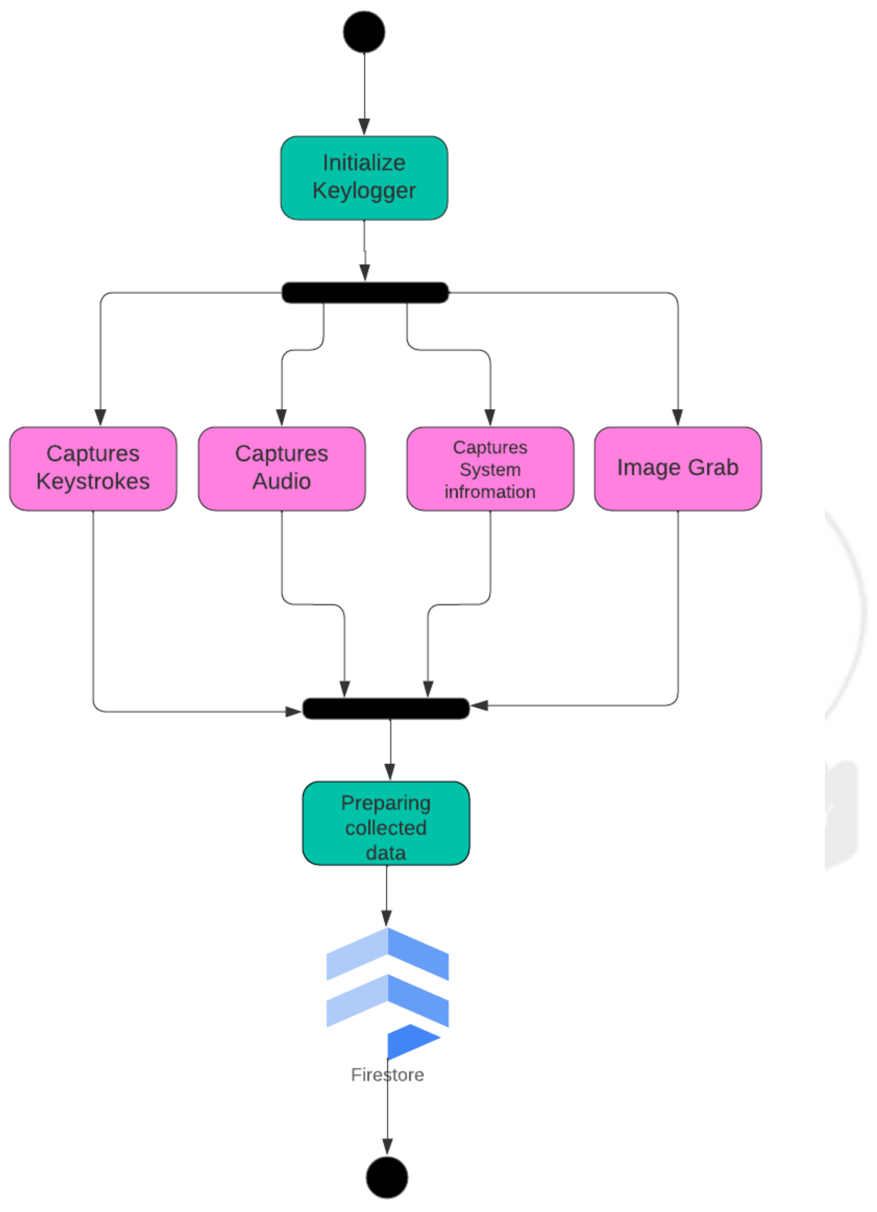
Laboratory Record **Roll No: 160121749301**

Of : UML **Experiment No:**

Sheet No:

Date:

Activity Diagram for Advanced Keylogger



CBIT

EXPERIMENT-8 (COMPONENT DIAGRAMS)

AIM: To demonstrate Component Diagrams in Unified Modelling Diagram (UML).

DESCRIPTION: Component diagrams in UML (Unified Modeling Language) are like blueprints for a complex system. They offer a static view, focusing on the individual components, their interactions, and dependencies. Unlike deployment diagrams that showcase the physical layout, component diagrams delve into the modular building blocks and how they work together.

Core Components:

- Components: These are the independent, reusable modules that make up the system. They can be physical entities like libraries or executables, or logical groupings of functionalities. Each component is depicted as a rectangle in the diagram.
- Interfaces: These act as contracts between components, defining the services offered (provided interfaces) and the services required (required interfaces). Imagine them as agreements that components use to communicate. Interfaces are typically shown as circles or lollypop shapes connected to components by lines.
- Dependencies: These represent the relationships between components, signifying how they rely on each other's functionalities. An arrow pointing from component A to component B indicates that A depends on B to function.

Benefits of Component Diagrams:

- Enhanced Understanding: They provide a clear view of the system's modular structure, making it easier to grasp how components collaborate to achieve the overall goals.
- Improved Communication: These diagrams act as a common language for developers and stakeholders to discuss system architecture, fostering clear communication about component interactions.
- Promoted Reusability: By focusing on independent components, they encourage code reusability across different projects. Each component can potentially be used in various systems with minimal modification.
- Simplified Maintenance: Component diagrams help visualize dependencies. This allows developers to predict the impact of changes in one component on other parts of the system, streamlining maintenance efforts.

Advanced Concepts:

- Internal Compartments (Optional): You can add compartments within a component rectangle to show internal details or stereotypes (like «file» or «database»).

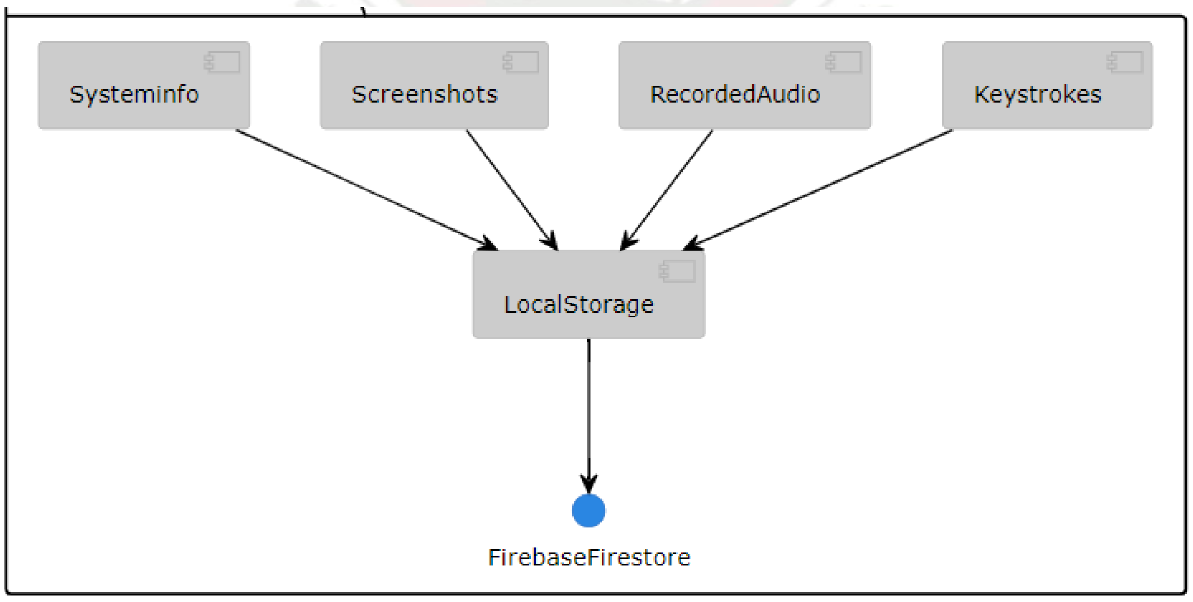
- Ports: These refine the interaction points within a component. Imagine them as specific doorways through which functionalities are provided or required.
- Provided vs. Required Interfaces: A component can have both types of interfaces. Provided interfaces expose its functionalities, while required interfaces specify the services it expects from other components.
- Assemblies: These represent groups of collaborating components that function as a single unit. They are useful for depicting complex subsystems within a larger system.

Things to Consider:

- Focus on Static Structure: Component diagrams capture a system's static view, not the dynamic behavior (how components interact over time). For that, consider using sequence diagrams or communication diagrams.
- Complexity with Large Systems: Diagrams can become cluttered for very large systems with numerous components and dependencies. Consider using packages or subsystems to manage complexity.

In essence, component diagrams are a powerful tool for visualizing and understanding how a system is built from independent, interacting components. They promote modularity, reusability, and clear communication during system design and development.

DIAGRAM:
Component Diagram for Advanced Keylogger



EXPERIMENT-9 (DEPLOYMENT DIAGRAMS)

AIM: To demonstrate Deployment Diagrams in Unified Modelling Diagram (UML).

DESCRIPTION: Deployment diagrams play a crucial role in visualizing the physical architecture of a system. They essentially depict how software components are deployed onto hardware nodes, providing a clear understanding of the system's real-world implementation.

Core Elements:

- Nodes: These represent the physical building blocks of the system, such as servers, workstations, databases, network devices, and any other hardware components involved. Nodes are depicted as rectangular boxes in the diagram.
- Artifacts: These embody the deployable software entities like executable files, libraries, configuration files, and databases. Artifacts are shown as small rectangles within the nodes, signifying their deployment on those specific hardware components.
- Deployment Specifications: These act as guidelines or configurations that dictate how artifacts are deployed onto nodes. They are typically not visualized in the diagram but can be documented alongside it.
- Dependencies: Deployment diagrams also depict relationships between nodes, established through communication paths. These dependencies can be physical connections (network cables) or logical interactions (data exchange).
- Components: These are the independent, reusable modules that make up the system. They can be physical entities like libraries or executables, or logical groupings of functionalities. Each component is depicted as a rectangle in the diagram.
- Interfaces: These act as contracts between components, defining the services offered (provided interfaces) and the services required (required interfaces). Imagine them as agreements that components use to communicate. Interfaces are typically shown as circles or lollypop shapes connected to components by lines.

Benefits of Deployment Diagrams:

- Clear Visualization: They provide a comprehensive picture of the system's physical layout, making it easier to understand how software components interact with hardware resources.
- Improved Communication: These diagrams act as a bridge between development teams and system administrators, fostering clear communication about deployment strategies.
- Enhanced Planning: By visualizing the physical architecture, deployment diagrams aid in capacity planning, resource allocation, and identifying potential bottlenecks.
- Documentation Value: They serve as valuable documentation for future reference, system maintenance, and understanding how the system functions in the real world.

Creating a Deployment Diagram:

1. Identify Nodes: Start by listing all the hardware components involved in the system.
2. Define Artifacts: Determine the software entities that need to be deployed onto the nodes.
3. Map Artifacts to Nodes: Specify which artifacts will be deployed on each node.
4. Establish Dependencies: Show the communication paths or interactions between nodes.
5. Document Deployment Specifications (Optional): If necessary, document any specific configurations or instructions for deployment.

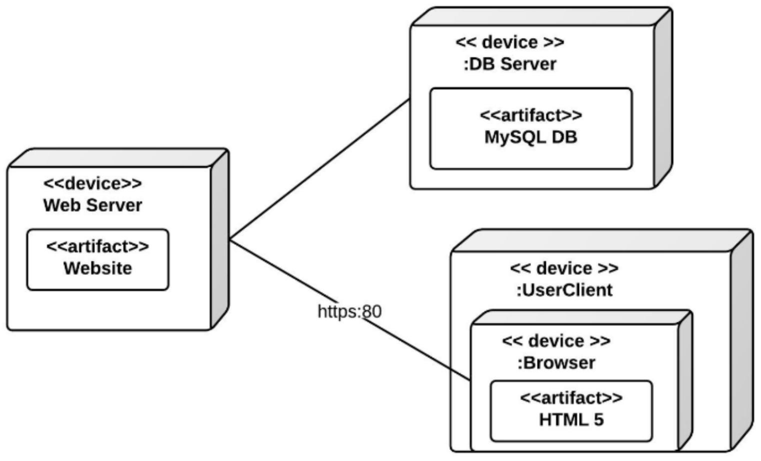
Beyond the Basics:

- Deployment Targets: These represent physical or logical locations where artifacts are deployed (e.g., development server, production environment).
- Deployment Environments: Diagrams can encompass different deployment environments like development, testing, and production.

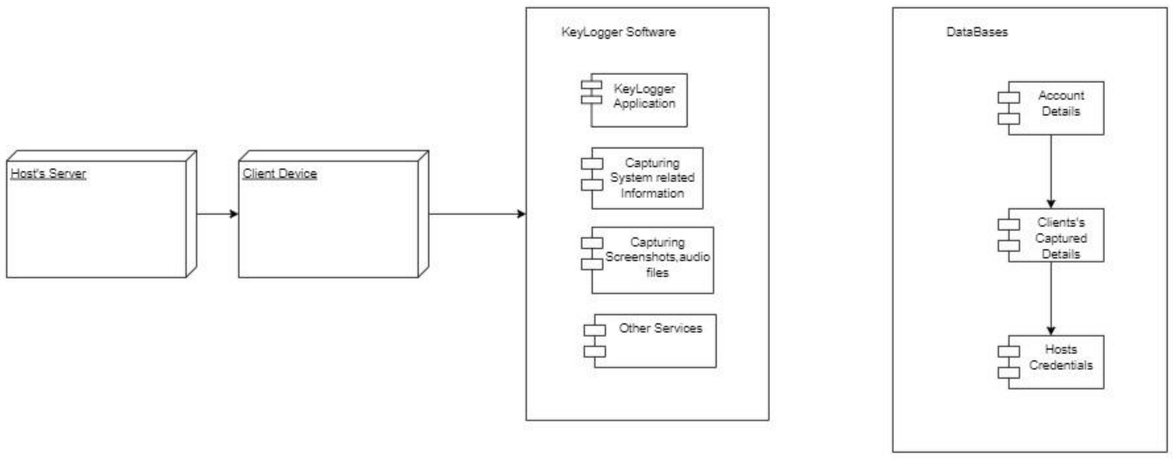
In Conclusion, deployment diagrams are a valuable asset in the UML toolbox. By providing a clear picture of the system's physical architecture, they promote better communication, planning, and understanding of how software interacts with the real world. So, the next time you're designing or deploying a system, leverage deployment diagrams to bridge the gap between the software and the hardware landscape.

DIAGRAM:

Deployment Diagram for a basic webpage



Deployment Diagram for Advanced Keylogger



EXPERIMENT-10 (COLLABORATION DIAGRAMS)

AIM: To demonstrate Collaboration Diagrams in Unified Modelling Diagram (UML).

DESCRIPTION: Collaboration diagrams, also known as communication diagrams in UML, provide a visual representation of how objects collaborate to fulfill a certain task or accomplish a specific behavior within a system.

Here's a deeper dive into their features:

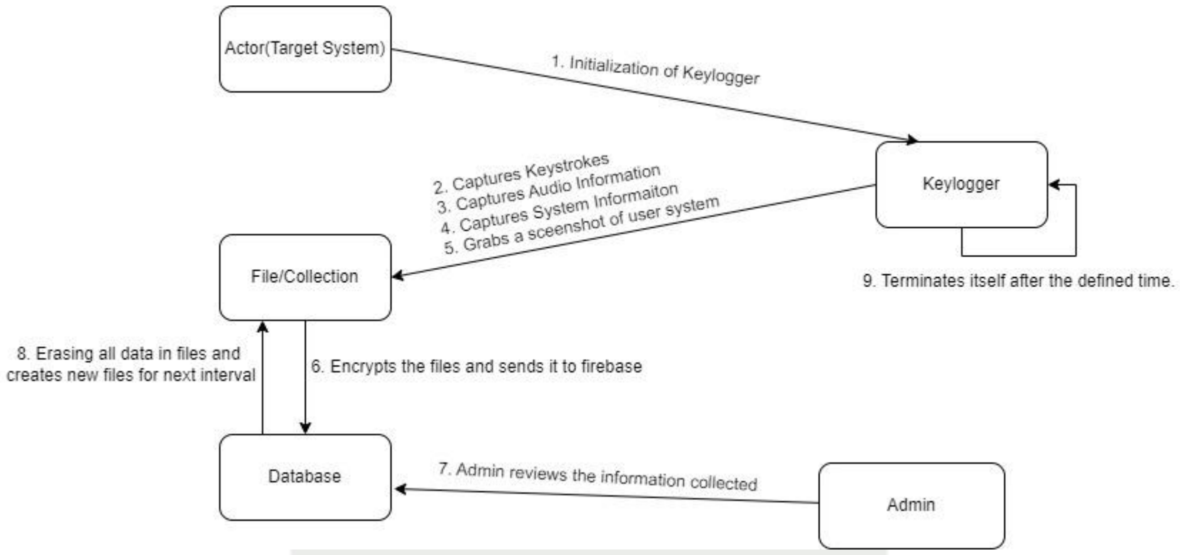
- Objects and Actors: Collaboration diagrams showcase the various objects or actors involved in the interaction. Objects, which are instances of classes, are depicted as rectangles with the object's name and its class type. Actors, representing entities external to the system, are also included if they play a role in the collaboration.
- Links and Associations: Links, depicted as lines connecting objects or actors, illustrate the communication pathways between them. These links often represent associations or relationships between the objects, indicating how they interact and exchange information.
- Messages and Interactions: Messages are the core elements of collaboration diagrams, representing the communication between objects or actors. Messages can take different forms, such as method calls, signals, or data exchanges, and they are typically labeled with the name of the message and any relevant parameters. The flow of messages demonstrates the sequence of interactions within the collaboration.
- Roles and Responsibilities: Collaboration diagrams may highlight the roles and responsibilities of each object or actor involved in the interaction. By assigning roles, the diagram clarifies the specific functions or behaviors that each participant contributes to the collaboration, facilitating a better understanding of the system's dynamics.
- Contextualization: Collaboration diagrams provide a contextual view of how objects collaborate within a particular scenario or use case. They focus on illustrating the interactions relevant to a specific behavior or task, allowing stakeholders to grasp the essential communication patterns without overwhelming details.
- Complementarity with Sequence Diagrams: While collaboration diagrams emphasize the structural aspects of interactions, sequence diagrams delve into the temporal sequence of messages exchanged between objects over time. Together, these diagram types offer complementary perspectives on system behavior, with collaboration diagrams providing a broader overview and sequence diagrams offering detailed insight into message sequencing.

- Simplicity and Abstraction: Collaboration diagrams aim to simplify complex interactions by abstracting away unnecessary details and focusing on the essential elements of the collaboration. This abstraction enables stakeholders to grasp the system's communication patterns more intuitively, fostering clearer communication and collaboration among team members.

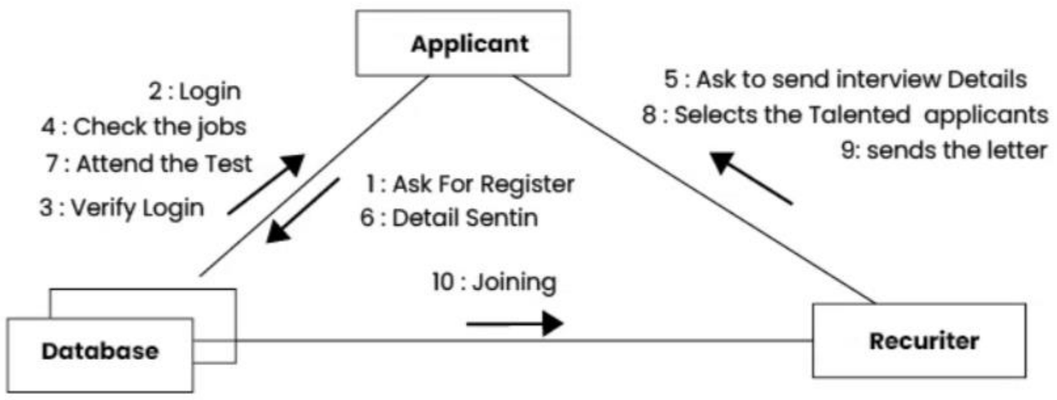
In essence, collaboration diagrams serve as powerful tools for modeling and analyzing the interactions between objects and actors within a system, enabling stakeholders to understand how components collaborate to achieve desired functionalities or behaviors.

DIAGRAMS:

Collaboration Diagram for Advanced Keylogger


Actor(Target System)
1. Initialization of Keylogger
Keylogger
2. Captures Keystrokes
3. Captures Audio Information
4. Captures System Informaiton
5. Grabs a sceenshot of user system
File/Collection
9. Terminates itself after the defined time.
8. Erasing all data in files and creates new files for next interval
6. Encrypts the files and sends it to firebase
Database
7. Admin reviews the information collected
Admin

Collaboration Diagram for basic Job Finding Platform



Advanced Keylogger

Software Requirements Specification (SRS) Document

1.Introduction

1.1 Purpose

The purpose of this document is to specify the requirements for the development of an advanced keylogger software capable of collecting system information and keystrokes for monitoring and administrative purposes.

1.2 Scope

The software will gather system information such as hostname, IP address, MAC address, username, public IP address, system details, and clipboard contents, while also functioning as a sophisticated keylogger. It will store the collected data securely in a Firebase Firestore database.

2. Overall Description

2.1 Product Perspective

The advanced keylogger will be a standalone Python script that utilizes platform-specific modules and APIs to collect system information and keystrokes. It will integrate with Firebase Firestore for secure data storage.

2.2 Product Features

•Collect comprehensive system information including hostname, IP address, MAC address, username, public IP address, system details, and clipboard contents.

•Capture keystrokes in real-time with support for special keys and key combinations.

•Integrate with Firebase Firestore for secure and scalable data storage.

2.3 User Classes and Characteristics

•Administrators: Responsible for monitoring system activities and analyzing collected data.

•Security Analysts: Interested in identifying anomalies and potential security threats through system monitoring.

3.System Features

3.1 Feature

1: System Information Collection Description

The software collects extensive system information and clipboard contents for analysis and monitoring purposes.

Packages Used:

- platform-specific modules and APIs: These will be used to gather system information such as hostname, IP address, MAC address, username, etc.
- getmac: This package will be used to retrieve the MAC address of the system.
- getpass: This module will be used to retrieve the username of the currently logged-in user.
- requests: This package may be used to interact with external APIs or services to gather additional system information.
- datetime: This module will be used to timestamp the collected system information.
- win32clipboard (specific to Windows): This package will be used to retrieve and store clipboard contents.
- firebase-admin: This package will be used to securely store the collected system information and clipboard contents in Firebase Firestore.

Requirements:

- Gather system information using platform-specific modules and APIs.
- Retrieve and store clipboard contents along with other system information.
- Ensure secure handling of sensitive information such as credentials and IP addresses.

3.2Feature 2: Keystroke Logging Description

The advanced keylogger captures keystrokes entered by the user in real-time and logs them securely.

Packages Used:

• pynput: This package will be used to capture keystrokes entered by the user in real-time, including special keys and key combinations.

• sounddevice and scipy: These packages may be used to capture audio recordings of keystrokes, depending on the implementation.

•firebase-admin: This package will be used to securely store the logged keystrokes in Firebase Firestore.

Requirements

• Capture individual keystrokes including special keys and key combinations.

• Log keystrokes in a secure manner and store them in the Firebase Firestore database.

• Implement efficient and reliable logging mechanisms to ensure data integrity.

3.3Feature 3: Firebase Firestore Integration Description

The software integrates seamlessly with Firebase Firestore for secure and scalable data storage.

Packages Used:

• firebase-admin: This package will be used to integrate seamlessly with Firebase Firestore for secure and scalable data storage.

• google-cloud-storage: This package may be used in conjunction with Firebase Firestore for additional storage capabilities, though it's not explicitly mentioned in the requirements.

Requirements

• Initialize Firebase Admin SDK with appropriate credentials and configuration.

• Store collected system information and keystrokes securely in a designated Firestore collection.

• Implement authentication and access controls to protect stored data from unauthorized access.

4. External Interface Requirements

4.1 User Interfaces

There is no direct user interface for this software. Configuration parameters may be specified within the script or through external configuration files.

4.2 Hardware Interfaces

The software interacts with the system's keyboard and clipboard functionality for keystroke logging and clipboard content retrieval.

4.3 Software Interfaces

The advanced keylogger relies on platform-specific modules and APIs for system information collection and keystroke logging. It also interacts with the Firebase Firestore API for secure data storage.

5. Non-functional Requirements

5.1 Performance Requirements

• The software should have minimal impact on system performance and resource utilization.

• Data collection and storage operations should be efficient and scalable to handle large volumes of data.

5.2 Security Requirements

• Collected data should be encrypted and stored securely in Firebase Firestore to prevent unauthorized access.

• Access to the software and stored data should be restricted to authorized personnel through proper authentication mechanisms.

5.3 Reliability Requirements

• The advanced keylogger should handle errors and exceptions gracefully to ensure uninterrupted operation.

• Implement robust error handling mechanisms to maintain data integrity and reliability.