Crime Analysis and Reporting System (C.A.R.S.)

Project Title: Crime Analysis and Reporting System (C.A.R.S.)
Submitted By: C.S.Deepta
Date: 07/04/2025
Trainer: Mrs.Karthika Madan
Organization: Hexaware

ABSTRACT

The Crime Analysis and Reporting System is a Python-based application designed to streamline the management, analysis, and reporting of crime-related data. This system facilitates the efficient recording of crime incidents, tracking their statuses, generating reports, and associating incidents with criminal cases. Developed using an object oriented approach, the system ensures modularity and maintainability. It integrates with a MySQL database to store and retrieve data securely. Core features include the creation and update of incidents, case management, dynamic reporting, and robust exception handling to ensure system stability. The project includes unit testing to validate the correctness of service functionalities. The system's architecture is layered, comprising entity models, data access objects, utility classes for database connection, and custom exception handling. Designed for use by law enforcement agencies or investigative departments, this project demonstrates the practical application of programming principles and database management to solve real-world problems.

Keywords - Crime Analysis, Crime Reporting, Incident Management, Case Management, MySQL Database, Object-Oriented Programming, Exception Handling, Unit Testing, Service Interface, Database Connectivity, PyCharm, SQL Integration, Criminal Record System.

TABLE OF CONTENTS

PURPOSE OF THE PROJECT

The **Crime Analysis and Reporting System (C.A.R.S.)** has been developed to serve as a robust, scalable, and user-friendly solution for managing crime-related data. The main purpose of this project is to empower law enforcement agencies with a centralized platform for efficiently recording, analyzing, and reporting criminal activities.

Key objectives include:

- ➢ **SQL and Database Design** for structured data storage and retrieval.
- ➢ **Control Flow, Loops, and Exception Handling** for robust application behavior.
- ➢ **Object-Oriented Programming** to represent real-world entities like Officers, Incidents, Victims, etc.
- ➢ **User-Defined Exceptions** to ensure reliable error reporting and handling.
- ➢ **Menu-Driven Console Interface** for easy navigation and use.
- ➢ **Unit Testing** to validate core functionalities and ensure system reliability.

Overall, the project is designed to create a comprehensive and reliable **Crime Analysis and Reporting System** using Python, SQL, and Object-Oriented Programming principles to facilitate the managing of the crime related data.

SCOPE OF THE PROJECT

Overview:

The Crime Analysis and Reporting System (C.A.R.S.) aims to streamline crime data management using Python. It covers functionalities such as incident creation, status updates, and report generation. The system handles entities like victims, suspects, officers, and law enforcement agencies. It ensures secure database connectivity and efficient information retrieval. Exception handling and unit testing are implemented to ensure system robustness. The project supports future scalability for analytical and reporting enhancements.

MODULES AND STRUCTURES

The system consists of multiple modules, each focusing on specific operations such as managing incidents, victims, suspects, officers, reports, and evidence. The main components are described below.

1. Database Design:

I. Entities:

- ➢ **Incidents**: Stores data about crime events including type, date, description, location, status.
- ➢ **Victims**: Stores victim details such as name, gender, DOB, and contact.
- ➢ **Suspects**: Contains suspect information like name, DOB, and contact.
- ➢ **Officers**: Officers who investigate incidents; includes ID, rank, and badge number.
- ➢ **Agencies**: Law enforcement agencies and their jurisdiction/contact details.
- ➢ **Evidence**: Items linked to an incident, with location and description.
- ➢ **Reports**: Reports written by officers for specific incidents.

II. Relationships:

- ➢ One **Incident** can involve multiple **Victims** and **Suspects**.
- ➢ Each **Incident** is linked to one **Agency**.
- ➢ An **Officer** belongs to one **Agency**.
- ➢ Multiple **Evidence** records can be linked to one **Incident**.
- ➢ **Reports** are generated for **Incidents** by specific **Officers**.

III SQL Tables & Schema:

- ➢ Each entity corresponds to a table with proper primary and foreign keys.
- ➢ Relations are handled via one-to-many and many-to-one connections.
- ➢ Normalized structure for efficient querying and scalability.

2. Python Program Structure:

I. User Authentication :

- ➢ Login system for officers/admin using secure credential checks.

II. Incident Management:

- ➢ Create, update, and retrieve incident details.
- ➢ Filter incidents based on date, type, or status.
- ➢ Link incidents with suspects, victims, and agencies.

III. Report and Evidence Management:

- ➢ Officers can write and finalize reports for incidents.
- ➢ Evidence items can be added and linked to incidents.

IV. Officer & Agency Management:

- ➢ Add/update officer details and assign them to agencies.
- ➢ Agency details like jurisdiction and contacts are stored.

2V. Case & Exception Handling:

➢ Create and manage cases linking multiple incidents.
➢ Handle exceptions like IncidentNumberNotFoundException and InvalidOfficerException.

TECHNOLOGIES USED FOR THE PROJECT

1. **MySQL:** Used as the relational database management system to store and manage crime details, incidents, and tracking information. SQL queries are employed to insert, update, and retrieve data from the database.
2. **PyCharm:** The Integrated Development Environment (IDE) used for Python development. It is utilized for writing, debugging, and testing the Python code for the Crime Analysis and Reporting System
3. **GitHub:** A platform for version control and collaboration. GitHub is used to manage the project's source code, track changes, and collaborate with team members, ensuring efficient version control throughout the development process.

ER DIAGRAM

An ER (Entity-Relationship) Diagram is a visual representation of the entities within a system and the relationships between them. It helps in designing and understanding the database structure by mapping out tables, attributes, and how they are interconnected. The **ER Diagram** for the Crime Analysis and Reporting System illustrates the key entities involved in the system and their interrelationships. It is designed to efficiently manage and analyze crime-related data while maintaining proper referential integrity.
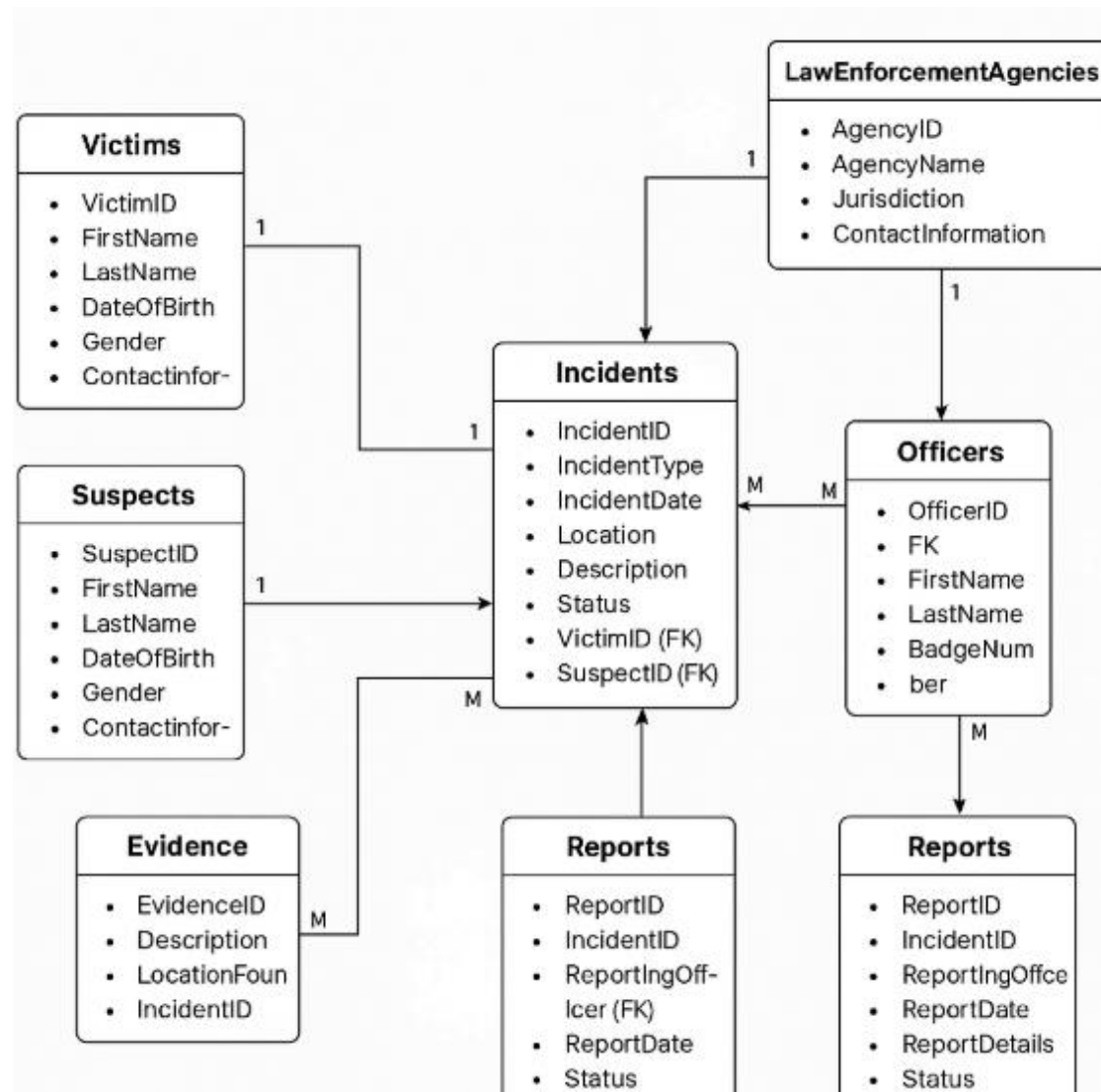
Entities and Their Attributes:

➢   Victims
→ Attributes:
• VictimID (Primary Key)
• FirstName
• LastName

- DateOfBirth
- Gender
- ContactInformation

➢ Suspects
  → Attributes:
  - SuspectID (Primary Key)
  - FirstName
  - LastName
  - DateOfBirth
  - Gender
  - ContactInformation

➢ LawEnforcementAgencies
  → Attributes:
  - AgencyID (Primary Key)
  - AgencyName
  - Jurisdiction
  - ContactInformation

➢ Officers
  → Attributes:
  - OfficerID (Primary Key)
  - FirstName
  - LastName
  - BadgeNumber (Unique)
  - Rank
  - ContactInformation
  - AgencyID (Foreign Key → LawEnforcementAgencies)

➢ Incidents
  → Attributes:
  - IncidentID (Primary Key)
  - IncidentType
  - IncidentDate
  - Location
  - Description
  - Status
  - VictimID (Foreign Key → Victims)
  - SuspectID (Foreign Key → Suspects)

➢ Evidence
  → Attributes:
  - EvidenceID (Primary Key)
  - Description
  - LocationFound
  - IncidentID (Foreign Key → Incidents)

➢ Reports
  → Attributes:

• ReportID (Primary Key)
• IncidentID (Foreign Key → Incidents)
• ReportingOfficer (Foreign Key → Officers)
• ReportDate
• ReportDetails
• Status

Relationships:

- A **LawEnforcementAgency** can have **many Officers** (1:M).

- Each **Officer** belongs to **one LawEnforcementAgency** (M:1).

- Each **Incident** can be related to **one Victim** and **one Suspect** (M:1).

- An **Incident** can have **multiple Evidence records** (1:M).

- An **Incident** can be associated with **multiple Reports**, but each **Report** is written for one Incident only (1:M).

- Each **Report** is created by a **single Officer**, but an **Officer** can create **multiple Reports** (1:M).

DATA FLOW DIAGRAM

A **Data Flow Diagram (DFD)** is a graphical representation that illustrates how data moves through a system, including where it comes from, how it is processed, and where it goes. It helps in understanding the flow of information and the functional aspects of a system. The Data Flow Diagram (DFD) for the Crime Analysis and Reporting System is as follows
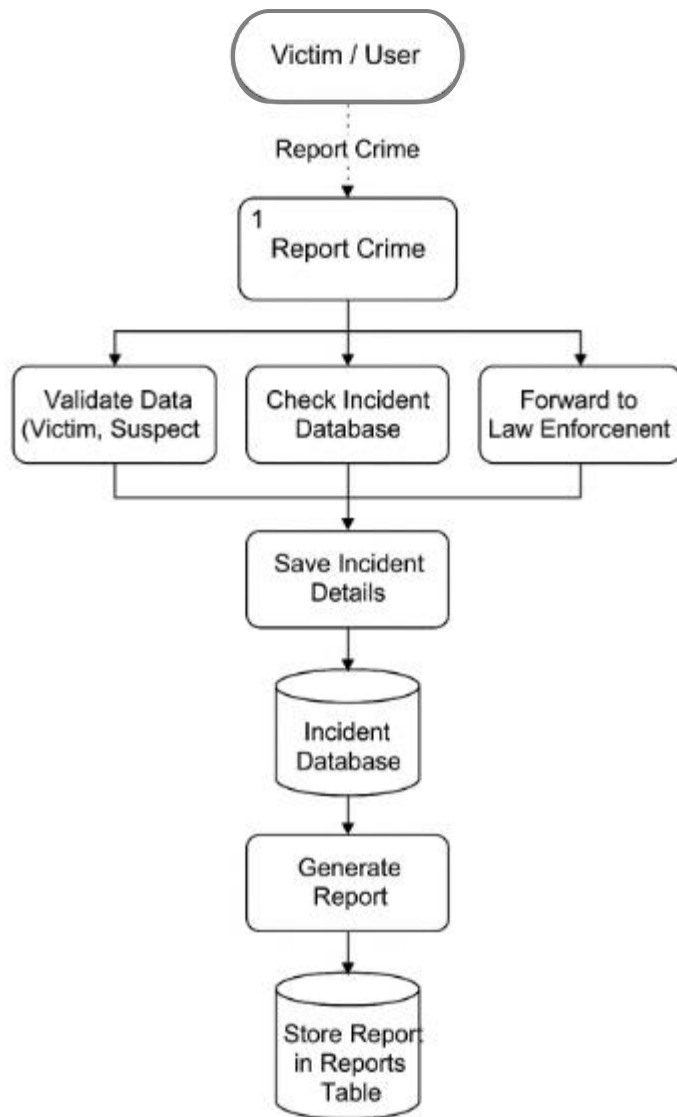
The Data Flow Diagram (DFD) for the Crime Analysis and Reporting System begins with the external entity, the Victim/User, who initiates the process by reporting a crime. This input flows into the first process, Report Crime, which captures all necessary details such as victim, suspect, and incident information. The system then performs data validation to ensure the accuracy and completeness of the entered data. After validation, it checks the Incident Database for any existing or related records. Once confirmed, the system proceeds to save the incident details. Simultaneously, the information is forwarded to law enforcement, where an officer is assigned to the case. After investigation or administrative review, a report is generated and finally stored in the Reports Table. The diagram captures the interaction between users, processes, and data stores, showcasing how crime data flows within the system efficiently.

TASK

1) Database Design for Crime Analysis and Reporting System

Objective:

The goal of this task is to design a relational database schema for the Crime Analysis and Reporting System. The schema should include tables for core entities such as Victims, Suspects, Officers, Incidents and other related entities. We will define the relationships between these tables using foreign keys and populate the tables with sample data to simulate real-world scenarios.

Step 1: Creating Database

To begin, we will create a database called CrimeAnalysis in MySQL. This will serve as the container for all the tables related to the C.A.R.S.

```
CREATE DATABASE CrimeAnalysis;
USE CrimeAnalysis;
```

Step 2 : Creating Table

- **Incidents Table:** The Incidents table stores information about criminal incidents reported in the system. It includes details such as the type, date, location, description, and status of the incident. It also links to the victim and suspect involved in the incident.

    Attributes:

    - **IncidentID:** The primary key, uniquely identifying each incident.
    - **IncidentType:** The type of incident (e.g., Robbery, Homicide, Theft).
    - **IncidentDate:** The date when the incident occurred.
    - **Location:** The geographical location of the incident
    - **Description:** A detailed description of the incident.
    - **Status:** The current status of the incident (e.g., Open, Closed, Under Investigation).
    - **VictimID:** Foreign key linking to the Victim table.
    - **SuspectID:** Foreign key linking to the Suspect table.

```
CREATE TABLE Incidents (
    IncidentID INT PRIMARY KEY AUTO_INCREMENT,
    IncidentType VARCHAR(100),
    IncidentDate DATE,
    Location VARCHAR(255),
    Description TEXT,
    Status VARCHAR(50),
    VictimID INT,
    SuspectID INT,
    FOREIGN KEY (VictimID) REFERENCES Victims(VictimID),
```

FOREIGN KEY (SuspectID) REFERENCES Suspects(SuspectID)
);

- **Victims Table:** The Victims table contains information about individuals who are victims in various incidents.

    Attributes:

    - **VictimID:** The primary key, uniquely identifying each victim.
    - **FirstName:** Victim's first name.
    - **LastName:** Victim's last name.
    - **DateOfBirth:** Victim's date of birth.
    - **Gender:** Victim's gender.
    - **ContactInformation:** Victim's contact details, including address and phone number.

CREATE TABLE Victims (
  VictimID INT PRIMARY KEY AUTO_INCREMENT,
  FirstName VARCHAR(100),
  LastName VARCHAR(100),
  DateOfBirth DATE,
  Gender VARCHAR(10),
  ContactInformation VARCHAR(255)
);

- **Suspects Table:** The Suspects table maintains data about individuals suspected of being involved in criminal activities.

    Attributes:

    - **SuspectID:** The primary key, uniquely identifying each suspect.
    - **FirstName:** Suspect's first name.
    - **LastName:** Suspect's last name.
    - **DateOfBirth:** Suspect's date of birth.
    - **Gender:** Suspect's gender.
    - **ContactInformation:** Suspect's contact details including address and phone number.

CREATE TABLE Suspects (
  SuspectID INT PRIMARY KEY AUTO_INCREMENT,
  FirstName VARCHAR(100),
  LastName VARCHAR(100),
  DateOfBirth DATE,
  Gender VARCHAR(10),
  ContactInformation VARCHAR(255)
);

- **LawEnforcementAgencies Table:** This table stores data about the law enforcement agencies involved in investigating incidents.

    Attributes:

- **AgencyID:** The primary key, uniquely identifying each law enforcement agency.
- **AgencyName:** Name of the agency.
- **Jurisdiction:** The geographical area covered by the agency.
- **ContactInformation:** Contact details of the agency.
- **OfficerID:** Links to officers associated with this agency.

```
CREATE TABLE LawEnforcementAgencies (
  AgencyID INT PRIMARY KEY AUTO_INCREMENT,
  AgencyName VARCHAR(100),
  Jurisdiction VARCHAR(100),
  ContactInformation VARCHAR(255)
);
```

- **Officers Table:** The Officers table stores information about officers working in law enforcement agencies.

Attributes:

- **OfficerID:** The primary key, uniquely identifying each officer.
- **FirstName:** Officer's first name.
- **LastName:** Officer's last name.
- **BadgeNumber:** Unique badge number of the officer.
- **Rank:** Rank of the officer (e.g., Inspector, Constable).
- **ContactInformation:** Officer's contact details.
- **AgencyID:** Foreign key linking to the Law Enforcement Agency table.

```
CREATE TABLE Officers (
  OfficerID INT PRIMARY KEY AUTO_INCREMENT,
  FirstName VARCHAR(100),
  LastName VARCHAR(100),
  BadgeNumber VARCHAR(100) UNIQUE,
  `Rank` VARCHAR(50),  -- Used backticks to avoid keyword conflict
  ContactInformation VARCHAR(255),
  AgencyID INT,
  FOREIGN KEY (AgencyID) REFERENCES LawEnforcementAgencies(AgencyID)
);
```

- **Evidence Table:** This table records the evidence collected from incident scenes.

Attributes:

- **EvidenceID:** The primary key, uniquely identifying each piece of evidence.
- **Description:** Description of the evidence.
- **LocationFound:** The place where the evidence was found.
- **IncidentID:** Foreign key linking to the associated incident.

```
CREATE TABLE Evidence (
  EvidenceID INT PRIMARY KEY AUTO_INCREMENT,
  Description TEXT,
  LocationFound VARCHAR(255),
```

```
    IncidentID INT,
    FOREIGN KEY (IncidentID) REFERENCES Incidents(IncidentID)
);
```

● **Report Table:** The Report table documents reports written for incidents by officers.

Attributes:

> **ReportID:** The primary key, uniquely identifying each report.
> **IncidentID:** Foreign key linking to the Incident table.
> **ReportingOfficer:** Foreign key linking to the Officer table.
> **ReportDate:** The date on which the report was filed.
> **ReportDetails:** Detailed content of the report.
> **Status:** Status of the report.

```
CREATE TABLE Reports (
    ReportID INT PRIMARY KEY AUTO_INCREMENT,
    IncidentID INT,
    ReportingOfficer INT,
    ReportDate DATE,
    ReportDetails TEXT,
    Status VARCHAR(50),
    FOREIGN KEY (IncidentID) REFERENCES Incidents(IncidentID),
    FOREIGN KEY (ReportingOfficer) REFERENCES Officers(OfficerID)
);
```

Step 3 : Inserting values into the table

Inserting values into the tables involves adding data to each table to populate the database with real-world information. This is done using the INSERT INTO SQL command, specifying the table name and the corresponding values for each column. It is important to ensure that the data types and constraints (such as primary and foreign keys) match the table definitions. Proper insertion of values is crucial for maintaining data consistency and integrity across the system. Additionally, sample data helps simulate the actual operations of the Crime Analysis and Reporting System, allowing for effective testing and validation of the system's functionality.

```
INSERT INTO LawEnforcementAgencies (AgencyName, Jurisdiction, ContactInformation)
VALUES
('Mumbai Police', 'Mumbai, India', 'mumbaipolice@gov.in'),
('Los Angeles PD', 'Los Angeles, USA', 'lapd@gov.us'),
('Scotland Yard', 'London, UK', 'scotlandyard@gov.uk'),
('New York PD', 'New York, USA', 'nypd@gov.us'),
('Interpol', 'International', 'interpol@gov.int'),
('Delhi Police', 'Delhi, India', 'delhipolice@gov.in'),
('Sydney Police', 'Sydney, Australia', 'sydpolice@gov.au'),
('Toronto Police', 'Toronto, Canada', 'torontopolice@gov.ca'),
('Dubai Police', 'Dubai, UAE', 'dubaipolice@gov.ae'),
('Tokyo Metropolitan Police', 'Tokyo, Japan', 'tokyopolice@gov.jp');
```

SELECT * FROM LawEnforcementAgencies;

	AgencyID	AgencyName	Jurisdiction	ContactInformation
▶	1	Mumbai Police	Mumbai, India	mumbaipolice@gov.in
	2	Los Angeles PD	Los Angeles, USA	lapd@gov.us
	3	Scotland Yard	London, UK	scotlandyard@gov.uk
	4	New York PD	New York, USA	nypd@gov.us
	5	Interpol	International	interpol@gov.int
	6	Delhi Police	Delhi, India	delhipolice@gov.in
	7	Sydney Police	Sydney, Australia	sydpolice@gov.au
	8	Toronto Police	Toronto, Canada	torontopolice@gov.ca
	9	Dubai Police	Dubai, UAE	dubaipolice@gov.ae
	10	Tokyo Metropolitan Police	Tokyo, Japan	tokyopolice@gov.jp
*	NULL	NULL	NULL	NULL

INSERT INTO Officers (FirstName, LastName, BadgeNumber, `Rank`, ContactInformation, AgencyID) VALUES
('Sheriff', 'Woody', 'TX1001', 'Sergeant', 'woody@lapd.com', 2),
('Buzz', 'Lightyear', 'TX1002', 'Lieutenant', 'buzz@nycpd.com', 4),
('Donald', 'Duck', 'TX1003', 'Detective', 'donald@scotlandyard.com', 3),
('Mickey', 'Mouse', 'TX1004', 'Captain', 'mickey@mumbaipolice.com', 1),
('Tom', 'Cat', 'TX1005', 'Inspector', 'tom@sydpolice.com', 7),
('Jerry', 'Mouse', 'TX1006', 'Constable', 'jerry@delhipolice.com', 6),
('SpongeBob', 'SquarePants', 'TX1007', 'Sergeant', 'spongebob@torontopolice.com', 8),
('Patrick', 'Star', 'TX1008', 'Lieutenant', 'patrick@dubaipolice.com', 9),
('Bugs', 'Bunny', 'TX1009', 'Detective', 'bugs@tokyopolice.com', 10),
('Daffy', 'Duck', 'TX1010', 'Commissioner', 'daffy@interpol.com', 5);

SELECT * FROM Officers;

	OfficerID	FirstName	LastName	BadgeNumber	Rank	ContactInformation	AgencyID
▶	1	Sheriff	Woody	TX1001	Sergeant	woody@lapd.com	2
	2	Buzz	Lightyear	TX1002	Lieutenant	buzz@nycpd.com	4
	3	Donald	Duck	TX1003	Detective	donald@scotlandyard.com	3
	4	Mickey	Mouse	TX1004	Captain	mickey@mumbaipolice.com	1
	5	Tom	Cat	TX1005	Inspector	tom@sydpolice.com	7
	6	Jerry	Mouse	TX1006	Constable	jerry@delhipolice.com	6
	7	SpongeBob	SquarePants	TX1007	Sergeant	spongebob@torontopolice.com	8
	8	Patrick	Star	TX1008	Lieutenant	patrick@dubaipolice	patrick@dubaipolice.com
	9	Bugs	Bunny	TX1009	Detective	bugs@tokyopolice.com	10
	10	Daffy	Duck	TX1010	Commissioner	daffy@interpol.com	5

INSERT INTO Victims (FirstName, LastName, DateOfBirth, Gender, ContactInformation) VALUES
('Shah Rukh', 'Khan', '1965-11-02', 'Male', 'srk@bollywood.com'),
('Salman', 'Khan', '1965-12-27', 'Male', 'salmankhan@bollywood.com'),
('Amitabh', 'Bachchan', '1942-10-11', 'Male', 'amitabh@bollywood.com'),
('Aishwarya', 'Rai', '1973-11-01', 'Female', 'aish@bollywood.com'),
('Priyanka', 'Chopra', '1982-07-18', 'Female', 'priyanka@bollywood.com'),
('Deepika', 'Padukone', '1986-01-05', 'Female', 'deepika@bollywood.com'),
('Ranveer', 'Singh', '1985-07-06', 'Male', 'ranveer@bollywood.com'),
('Hrithik', 'Roshan', '1974-01-10', 'Male', 'hrithik@bollywood.com'),
('Alia', 'Bhatt', '1993-03-15', 'Female', 'alia@bollywood.com'),
('Kareena', 'Kapoor', '1980-09-21', 'Female', 'kareena@bollywood.com');

SELECT * FROM Victims;

VictimID	FirstName	LastName	DateOfBirth	Gender	ContactInformation
1	Shah Rukh	Khan	1965-11-02	Male	srk@bollywood.com
2	Salman	Khan	1965-12-27	Male	salmankhan@bollywood.com
3	Amitabh	Bachchan	1942-10-11	Male	amit ̶ ̶ ̶ ̶ ̶ ̶ ̶ ̶ ̶ ̶
4	Aishwarya	Rai	1973-11-01	Female	aish ̶ ̶ ̶ ̶ ̶ ̶ ̶ ̶ ̶
5	Priyanka	Chopra	1982-07-18	Female	priyanka@bollywood.com
6	Deepika	Padukone	1986-01-05	Female	deepika@bollywood.com
7	Ranveer	Singh	1985-07-06	Male	ranveer@bollywood.com
8	Hrithik	Roshan	1974-01-10	Male	hrithik@bollywood.com
9	Alia	Bhatt	1993-03-15	Female	alia@bollywood.com
10	Kareena	Kapoor	1980-09-21	Female	kareena@bollywood.com
NULL	NULL	NULL	NULL	NULL	NULL

(tooltip shown: amitabh@bollywood.com)

INSERT INTO Suspects (FirstName, LastName, DateOfBirth, Gender, ContactInformation) VALUES
('Johnny', 'Depp', '1963-06-09', 'Male', 'johnny@hollywood.com'),
('Leonardo', 'DiCaprio', '1974-11-11', 'Male', 'leo@hollywood.com'),
('Angelina', 'Jolie', '1975-06-04', 'Female', 'angelina@hollywood.com'),
('Brad', 'Pitt', '1963-12-18', 'Male', 'brad@hollywood.com'),
('Tom', 'Cruise', '1962-07-03', 'Male', 'tom@hollywood.com'),
('Selena', 'Gomez', '1992-07-22', 'Female', 'selena@hollywood.com'),
('Robert', 'Downey Jr.', '1965-04-04', 'Male', 'rdj@hollywood.com'),
('Scarlett', 'Johansson', '1984-11-22', 'Female', 'scarlett@hollywood.com'),
('Chris', 'Evans', '1981-06-13', 'Male', 'chris@hollywood.com'),
('Will', 'Smith', '1968-09-25', 'Male', 'will@hollywood.com');

SELECT * FROM Suspects;

SuspectID	FirstName	LastName	DateOfBirth	Gender	ContactInformation
1	Johnny	Depp	1963-06-09	Male	johnny@hollywood.com
2	Leonardo	DiCaprio	1974-11-11	Male	leo@hollywood.com
3	Angelina	Jolie	1975-06-04	Female	angelina@hollywood.com
4	Brad	Pitt	1963-12-18	Male	brad@hollywood.com
5	Tom	Cruise	1962-07-03	Male	tom@hollywood.com
6	Selena	Gomez	1992-07-22	Female	selena@hollywood.com
7	Robert	Downey Jr.	1965-04-04	Male	rdj@hollywood.com
8	Scarlett	Johansson	1984-11-22	Female	scarlett@hollywood.com
9	Chris	Evans	1981-06-13	Male	chris@hollywood.com
10	Will	Smith	1968-09-25	Male	will@hollywood.com
NULL	NULL	NULL	NULL	NULL	NULL

INSERT INTO Incidents (IncidentType, IncidentDate, Location, Description, Status, VictimID, SuspectID) VALUES
('Robbery', '2024-01-05', 'Mumbai, India', 'Shah Rukh Khan was robbed at a film set.', 'Under Investigation', 1, 1),
('Homicide', '2024-02-10', 'Delhi, India', 'Aamir Khan was found dead in a hotel room.', 'Closed', 2, 2),

('Theft', '2024-03-15', 'Chennai, India', 'Deepika Padukone reported her jewelry stolen.', 'Open', 3, 3),
('Kidnapping', '2024-04-20', 'Hyderabad, India', 'Hrithik Roshan was kidnapped and later found.', 'Closed', 4, 4),
('Assault', '2024-05-25', 'Pune, India', 'Salman Khan was assaulted at an event.', 'Under Investigation', 5, 5),
('Burglary', '2024-06-30', 'Kolkata, India', 'Alia Bhatt's house was broken into.', 'Open', 6, 6),
('Fraud', '2024-07-05', 'Bangalore, India', 'Ranbir Kapoor was scammed in a real estate deal.', 'Under Investigation', 7, 7),
('Murder', '2024-08-10', 'Jaipur, India', 'Kareena Kapoor found dead in her apartment.', 'Closed', 8, 8),
('Cyber Crime', '2024-09-15', 'Lucknow, India', 'Ranveer Singh's social media was hacked.', 'Open', 9, 9),
('Drug Possession', '2024-10-20', 'Ahmedabad, India', 'Ajay Devgn caught with illegal substances.', 'Under Investigation', 10, 10)
('Robbery','2024-10-05','hyderabad','the bank was robbered','open',5,6);

SELECT * FROM Incidents;

IncidentID	IncidentType	IncidentDate	Location	Description	Status	VictimID	SuspectID
1	Robbery	2024-01-05	Mumbai, India	Shah Rukh Khan was robbed at a film set.	Under Investigation	1	1
2	Homicide	2024-02-10	Delhi, India	Aamir Khan was found dead in a hotel room.	Closed	2	2
3	Theft	2024-03-15	Chennai, India	Deepika Padukone reported her jewelry stolen.	Open	3	3
4	Kidnapping	2024-04-20	Hyderabad, India	Hrithik Roshan was kidnapped and later found.	Closed	4	4
5	Assault	2024-05-25	Pune, India	Salman Khan was assaulted at an event.	Under Investigation	5	5
6	Burglary	2024-06-30	Kolkata, India	Alia Bhatt's house was broken into.	Open	6	6
7	Fraud	2024-07-05	Bangalore, India	Ranbir Kapoor was scammed in a real estate deal.	Under Investigation	7	7
8	Murder	2024-08-10	Jaipur, India	Kareena Kapoor found dead in her apartment.	Closed	8	8
9	Cyber Crime	2024-09-15	Lucknow, India	Ranveer Singh's social media was hacked.	Open	9	9
10	Drug Possession	2024-10-20	Ahmedabad, India	Ajay Devgn caught with illegal substances.	Under Investigation	10	10
11	Robbery	2024-10-05	hyderabad	the bank was robbered	open	5	6
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

INSERT INTO Evidence (Description, LocationFound, IncidentID) VALUES
('A stolen diamond ring', 'Mumbai Jewelry Store', 1),
('A bloody knife', 'Hotel Room, Delhi', 2),
('A missing laptop', 'Chennai House', 3),
('Fingerprints on a bottle', 'Hyderabad Park', 4),
('CCTV footage of the assault', 'Pune Event Hall', 5),
('A broken window', 'Alia Bhatt's house, Kolkata', 6),
('Fake property documents', 'Real Estate Office, Bangalore', 7),
('A gun found near the scene', 'Jaipur Apartment', 8),
('Hacked social media account', 'Bangalore Office', 9),
('Illegal drugs found in car', 'Ahmedabad Highway', 10);

SELECT * FROM Evidence;

	EvidenceID	Description	LocationFound	IncidentID
▶	1	A stolen diamond ring	Mumbai Jewelry Store	1
	2	A bloody knife	Hotel Room, Delhi	2
	3	A missing laptop	Chennai House	3
	4	Fingerprints on a bottle	Hyderabad Park	4
	5	CCTV footage of the assault	Pune Event Hall	5
	6	A broken window	Alia Bhatt's house, Kolkata	6
	7	Fake property documents	Real Estate Office, Bangalore	7
	8	A gun found near the scene	Jaipur Apartment	8
	9	Hacked social media account	Bangalore Office	9
	10	Illegal drugs found in car	Ahmedabad Highway	10
*	NULL	NULL	NULL	NULL

INSERT INTO Reports (IncidentID, ReportingOfficer, ReportDate, ReportDetails, Status) VALUES
(1, 1, '2024-01-06', 'Initial investigation started. CCTV footage being reviewed.', 'Under Investigation'),
(2, 2, '2024-02-11', 'Autopsy completed. Case closed as confirmed homicide.', 'Closed'),
(3, 3, '2024-03-16', 'Theft reported. No suspects identified yet.', 'Open'),
(4, 4, '2024-04-21', 'Victim found safe. Kidnappers in custody.', 'Closed'),
(5, 5, '2024-05-26', 'Suspect identified from event footage. Arrest pending.', 'Under Investigation'),
(6, 6, '2024-07-01', 'Entry through window confirmed. Forensics underway.', 'Open'),
(7, 7, '2024-07-06', 'Fraudulent documents submitted. Case under review.', 'Under Investigation'),
(8, 8, '2024-08-11', 'Victim's death ruled as murder. Investigation complete.', 'Closed'),
(9, 9, '2024-09-16', 'Hacking traced to overseas server. Cyber team alerted.', 'Open'),
(10, 10, '2024-10-21', 'Drugs found in suspect's vehicle. Lab results awaited.', 'Under Investigation');

SELECT * FROM Reports;

	ReportID	IncidentID	ReportingOfficer	ReportDate	ReportDetails	Status
▶	1	1	1	2024-01-06	Initial investigation started. CCTV footage bein...	Under Investigation
	2	2	2	2024-02-11	Autopsy completed. Case closed as confirmed h...	Closed
	3	3	3	2024-03-16	Theft reported. No suspects identified yet.	Open
	4	4	4	2024-04-21	Victim found safe. Kidnappers in custody.	Closed
	5	5	5	2024-05-26	Suspect identified from event footage. Arrest p...	Under Investigation
	6	6	6	2024-07-01	Entry through window confirmed. Forensics und...	Open
	7	7	7	2024-07-06	Fraudulent documents submitted. Case under r...	Under Investigation
	8	8	8	2024-08-11	Victim's death ruled as murder. Investigation co...	Closed
	9	9	9	2024-09-16	Hacking traced to overseas server. Cyber team...	Open
	10	10	10	2024-10-21	Drugs found in suspect's vehicle. Lab results aw...	Under Investigation
*	NULL	NULL	NULL	NULL	NULL	NULL

PART -2 CODING

TASK

Create the model/entity classes corresponding to the schema within package entity with variables declared private, constructors(default and parametrized) and getters,setters)

Folder: entity

This package acts as the foundation of the project, where all the core domain classes (also called models or entity classes) reside. These classes are designed to represent the data objects that the system will handle, such as Incidents, Cases, and Status Updates.Each class here corresponds to a real-world object that will be stored in the database. The fields in the class match the attributes of the database table, and each class is built using the concept of encapsulation — meaning the variables are declared private, and access to them is only possible through getters and setters.

- incident.py

  - ➢ This class represents a criminal incident reported in the system.
  - ➢ It contains all the necessary fields such as incidentId, incidentType, date, location, description, and possibly a status.
  - ➢ It has a default constructor (with no arguments) and a parameterized constructor (which initializes all fields).
  - ➢ The purpose of this class is to store all the details about a single reported incident.
  - ➢ It will be used when adding new incidents, updating them, fetching data, and generating reports.

```python
class Incident:
    def __init__(self, incident_id, incident_type, description, date, status):
        self.__incident_id = incident_id
        self.__incident_type = incident_type
        self.__description = description
        self.__date = date
        self.__status = status


    def get_incident_id(self):
        return self.__incident_id

    def get_incident_type(self):
        return self.__incident_type

    def get_description(self):
        return self.__description

    def get_date(self):
        return self.__date

    def get_status(self):
        return self.__status
```

```
def set_status(self, status):
    self.__status = status
```

- case.py

  ➢ Represents a case, which is a collection or grouping of incidents.
  ➢ Attributes include caseId, caseDescription, and a collection (like a list) of associated Incident objects.
  ➢ It is important for higher-level tracking where multiple incidents are part of a bigger criminal case.
  ➢ This entity links to the incident class and is used when associating multiple incidents together under one case for investigation and analysis.
  ➢ Includes constructors and appropriate getter/setter methods to allow manipulation and data access.

```
class Case:
    def __init__(self, case_id, case_description, incidents):
        self.__case_id = case_id
        self.__case_description = case_description
        self.__incidents = incidents


    def get_case_id(self):
        return self.__case_id

    def get_case_description(self):
        return self.__case_description

    def get_incidents(self):
        return self.__incidents


    def set_case_description(self, case_description):
        self.__case_description = case_description

    def add_incident(self, incident):
        self.__incidents.append(incident)
```

- status.py

  ➢ Represents the status of an incident — like "Open", "Under Investigation", "Closed", etc.
  ➢ It has fields such as incidentId, status, updatedBy, and updateDate.
  ➢ This class is crucial when the status of an incident needs to be updated or fetched for reporting.
  ➢ It supports tracking the progress of a case over time.
  ➢ The class includes a default constructor for flexibility and a parameterized one for quick creation of objects with all values set.

```
class Status:
    def __init__(self, status_id, status_name):
        self.__status_id = status_id
        self.__status_name = status_name
```

```python
    def get_status_id(self):
        return self.__status_id

    def get_status_name(self):
        return self.__status_name


    def set_status_name(self, status_name):
        self.__status_name = status_name
```

Importance of Entity Classes :

➢ They form the **blueprint** for the data your system will use.
➢ These classes are used by your **service layer** (in dao/) to transfer data to and from the database.
➢ They help in **data encapsulation**, following the principles of object-oriented programming.

TASK

Service Provider Interface/Abstract class

Keep the interfaces and implementation classes in package dao
Create ICrimeAnalysisService Interface/abstract classs with the following methods

// **Create a new incident**
createIncident();
parameters- **Incident** object
return type Boolean
// **Update the status of an incident**
updateIncidentStatus();
parameters- **Status** object,incidentid
return type Boolean
// **Get a list of incidents within a date range**
getIncidentsInDateRange();
parameters- startDate, endDate
return type Collection of Incident objects
// **Search for incidents based on various criteria**
searchIncidents(IncidentType criteria);
parameters- **IncidentType object**
return type Collection of Incident objects
// **Generate incident reports**
generateIncidentReport();
parameters- **Incident object**
return type Report object
// **Create a new case and associate it with incidents**
createCase();
parameters- caseDescription string, collection of Incident Objects
return type Case object
// **Get details of a specific case**
Case getCaseDetails(int caseId);

parameters- caseDescription string, collection of Incident Objects
return type Case object
// Update case details
updateCaseDetails();
parameters- Case object
return type boolean
// **Get a list of all cases**
List<Case> getAllCases();
parameters- None
return type Collection of cases

Folder: dao

This package (dao stands for **Data Access Object**) contains the **core service interface** and its **implementation** class. This is where all your **business logic** lives.

Interface: icrime_analysis_service.py

This is the service interface/abstract class that defines the contract of the Crime Analysis system.

Description of Each Method in crime_analysis_service_impl.py:

● **createIncident(incident: Incident) -> bool**

   ➢ **Purpose**: Add a new incident to the system.
   ➢ **Parameter**: Takes an Incident object (from entity.incident).
   ➢ **Returns**: True if created successfully, otherwise False.
   ➢ **Use Case**: Whenever a user wants to report a new crime.

● **updateIncidentStatus(status: Status, incident_id: int) -> bool**

   ➢ **Purpose**: Update the status of an existing incident.
   ➢ **Parameter**: A Status object and the incident_id to be updated.
   ➢ **Returns**: Boolean indicating success or failure.
   ➢ **Use Case**: Change from "Open" to "Investigating", etc.

● **getIncidentsInDateRange(start_date, end_date) -> Collection[Incident]**

   ➢ **Purpose**: Fetch incidents between two dates.
   ➢ **Parameters**: start_date and end_date.
   ➢ **Returns**: A collection (list) of matching Incident objects.
   ➢ **Use Case**: When generating reports or statistics for a time period.

● **searchIncidents(criteria: str) -> Collection[Incident]**

   ➢ **Purpose**: Search for incidents based on incident type or keyword.
   ➢ **Parameter**: A criteria string or possibly a full IncidentType object.

> ➢ **Returns**: A list of Incident objects matching the criteria.
> ➢ **Use Case**: Search by keyword like "theft", "assault", etc.

● **generateIncidentReport(incident: Incident) -> Report**

> ➢ **Purpose**: Generate a detailed report for a given incident.
> ➢ **Parameter**: The Incident object.
> ➢ **Returns**: A Report object (could be defined later in the model).
> ➢ **Use Case**: For sharing case summaries with stakeholders or departments.

● **createCase(case_description: str, incidents: Collection[Incident]) -> Case**

> ➢ **Purpose**: Create a new case and link it to a group of incidents.
> ➢ **Parameters**: Description string and a collection of Incident objects.
> ➢ **Returns**: A new Case object.
> ➢ **Use Case**: When police link several similar incidents under one case.

● **getCaseDetails(case_id: int) -> Case**

> ➢ **Purpose**: Get full details of a specific case using its ID.
> ➢ **Parameter**: The case_id.
> ➢ **Returns**: A Case object with all its data and linked incidents.
> ➢ **Use Case**: Case review, court processing, or audit.

● **updateCaseDetails(case: Case) -> bool**

> ➢ **Purpose**: Modify/update existing case details.
> ➢ **Parameter**: The Case object to be updated.
> ➢ **Returns**: Boolean value indicating update status.
> ➢ **Use Case**: Add additional notes, update the case description, etc.

● **getAllCases() -> Collection[Case]**

> ➢ **Purpose**: Fetch all criminal cases stored in the system.
> ➢ **Parameters**: None.
> ➢ **Returns**: A list of all Case objects.
> ➢ **Use Case**: Display to admin or analyst for filtering and tracking.

```
from dao.icrime_analysis_service import ICrimeAnalysisService
from util.db_connection import DBConnection

class CrimeAnalysisServiceImpl(ICrimeAnalysisService):
    def __init__(self):
        self.connection = DBConnection.get_connection()

    def create_incident(self, incident):

        try:
            cursor = self.connection.cursor()
            query = "INSERT INTO Incident (incident_id, incident_type, description, date, location, status) VALUES (%s, %s, %s, %s, %s, %s)"
```

```python
        values = (incident.incident_id, incident.incident_type, incident.description, incident.date,
incident.location, incident.status)
        cursor.execute(query, values)
        self.connection.commit()
        return True
    except Exception as e:
        print(f"Error creating incident: {e}")
        return False

def update_incident_status(self, status, incident_id):
    try:
        cursor = self.connection.cursor()
        query = "UPDATE Incident SET status = %s WHERE incident_id = %s"
        cursor.execute(query, (status, incident_id))
        self.connection.commit()
        return True
    except Exception as e:
        print(f"Error updating incident status: {e}")
        return False

def get_incidents_in_date_range(self, start_date, end_date):
    try:
        cursor = self.connection.cursor()
        query = "SELECT * FROM Incident WHERE date BETWEEN %s AND %s"
        cursor.execute(query, (start_date, end_date))
        return cursor.fetchall()
    except Exception as e:
        print(f"Error fetching incidents in date range: {e}")
        return []

def search_incidents(self, incident_type):
    try:
        cursor = self.connection.cursor()
        query = "SELECT * FROM Incident WHERE incident_type = %s"
        cursor.execute(query, (incident_type,))
        return cursor.fetchall()
    except Exception as e:
        print(f"Error searching incidents: {e}")
        return []

def generate_incident_report(self, incident):

    return {
        "incident_id": incident.incident_id,
        "summary": f"Report for incident {incident.incident_id} - {incident.description}"
    }

def create_case(self, case_description, incidents):
    try:
        cursor = self.connection.cursor()
        query = "INSERT INTO Cases (description) VALUES (%s)"
        cursor.execute(query, (case_description,))
        case_id = cursor.lastrowid

        for incident in incidents:
            link_query = "INSERT INTO Case_Incident (case_id, incident_id) VALUES
(%s, %s)"
            cursor.execute(link_query, (case_id, incident.incident_id))

        self.connection.commit()
```

```python
            return {"case_id": case_id, "description": case_description}
        except Exception as e:
            print(f"Error creating case: {e}")
            return None

    def get_case_details(self, case_id):
        try:
            cursor = self.connection.cursor()
            query = "SELECT * FROM Cases WHERE case_id = %s"
            cursor.execute(query, (case_id,))
            return cursor.fetchone()
        except Exception as e:
            print(f"Error fetching case details: {e}")
            return None

    def update_case_details(self, case_obj):
        try:
            cursor = self.connection.cursor()
            query = "UPDATE Cases SET description = %s WHERE case_id = %s"
            cursor.execute(query, (case_obj.description, case_obj.case_id))
            self.connection.commit()
            return True
        except Exception as e:
            print(f"Error updating case: {e}")
            return False

    def get_all_cases(self):
        try:
            cursor = self.connection.cursor()
            query = "SELECT * FROM Cases"
            cursor.execute(query)
            return cursor.fetchall()
        except Exception as e:
            print(f"Error fetching all cases: {e}")
            return []
```

TASK

1) Connect your application to the SQL database:

Write code to establish a connection to your SQL database.
2) Create a utility class DBConnection in a package util with a static variable connection of Type Connection and a static method getConnection() which returns connection. Connection properties supplied in the connection string should be read from a property file.
3) Create a utility class PropertyUtil which contains a static method named getPropertyString() which reads a property fie containing connection details like hostname, dbname, username, password, port number and returns a connection string.

Folder: util

The util package contains **helper/utility classes** that assist the main application. These classes **don't hold business logic**, but support it — in this case, by **connecting the application to a database**.

db_connection.py

● **Purpose:**

This class is responsible for **creating and managing a single static database connection** using Python's DB-API (likely with a library like mysql.connector or psycopg2).

● **How it works:**

  ➢ It contains a **static variable** (say connection) that stores the database connection.
  ➢ The method getConnection() checks if the connection is already open; if not, it:
  ➢ Calls the PropertyUtil.getPropertyString() method.
  ➢ Reads the connection parameters.
  ➢ Establishes a connection and returns it.

● **Why it's important:**

  ➢ Avoids repeatedly creating new connections.
  ➢ Makes your code **centralized** and **maintainable** for DB access.

```
import mysql.connector
from util.property_util import PropertyUtil

class DBConnection:
    connection = None

    @staticmethod
    def get_connection():
        if DBConnection.connection is None:
            props = PropertyUtil.get_property_string()
            DBConnection.connection = mysql.connector.connect(
                host=props['host'],
                port=props['port'],
                user=props['user'],
                password=props['password'],
                database=props['database']
            )
        return DBConnection.connection
```

property_util.py

● **Purpose:**

This utility class **reads database configuration values** from a property file (db.ini or db.properties.py) and assembles them into a usable connection string or dictionary.

```
import configparser
import os

class PropertyUtil:
    @staticmethod
```

```python
def get_property_string():
    config = configparser.ConfigParser()
    file_path = os.path.join(os.path.dirname(__file__), 'db.ini')
    print("Looking for DB config at:", file_path)

    with open(file_path, 'r') as f:
        print("File content:\n", f.read())

    config.read(file_path)

    print("Sections found:", config.sections())

    return {
        'host': config.get('database', 'host'),
        'port': config.getint('database', 'port'),
        'user': config.get('database', 'user'),
        'password': config.get('database', 'password'),
        'database': config.get('database', 'database')
    }
```

➢ Creating db.ini file under util folder

```ini
[database]
host = localhost
port = 3306
user = root
password = root
database = CrimeAnalysis
```

TASK

Service implementation
1. Create a Service class CrimeAnalysisServiceImpl in package dao with a static variable named connection of type Connection which can be assigned in the constructor by invoking the getConnection() method in DBConnection class
2. Provide implementation for all the methods in the interface/abstract clsass

Folder: dao

This package acts as the **Data Access Layer (DAL)** and includes:

➢ The interface: icrime_analysis_service.py
➢ The implementation: crime_analysis_service_impl.py

icrime_analysis_service.py

```python
from abc import ABC, abstractmethod
from entity.incident import Incident
from entity.status import Status
from entity.case import Case

class ICrimeAnalysisService(ABC):

    @abstractmethod
    def create_incident(self, incident: Incident) -> bool:
```

```python
    """Create a new incident"""
    pass

@abstractmethod
def update_incident_status(self, status: Status, incident_id: int) -> bool:
    """Update the status of an incident"""
    pass

@abstractmethod
def get_incidents_in_date_range(self, start_date: str, end_date: str):
    """Get a list of incidents within a date range"""
    pass

@abstractmethod
def search_incidents(self, incident_type: str):
    """Search for incidents based on various criteria"""
    pass

@abstractmethod
def generate_incident_report(self, incident: Incident):
    """Generate an incident report"""
    pass

@abstractmethod
def create_case(self, case_description: str, incidents: list):
    """Create a new case and associate it with incidents"""
    pass

@abstractmethod
def get_case_details(self, case_id: int) -> Case:
    """Get details of a specific case"""
    pass

@abstractmethod
def update_case_details(self, case: Case) -> bool:
    """Update case details"""
    pass

@abstractmethod
def get_all_cases(self):
    """Get a list of all cases"""
    pass
```

crime_analysis_service_impl.py

```python
from dao.icrime_analysis_service import ICrimeAnalysisService
from util.db_connection import DBConnection

class CrimeAnalysisServiceImpl(ICrimeAnalysisService):
    def __init__(self):
        self.connection = DBConnection.get_connection()

    def create_incident(self, incident):

        try:
            cursor = self.connection.cursor()
            query = "INSERT INTO Incident (incident_id, incident_type, description, date, location, status) VALUES (%s, %s, %s, %s, %s, %s)"
            values = (incident.incident_id, incident.incident_type, incident.description, incident.date, incident.location, incident.status)
```

```python
            cursor.execute(query, values)
            self.connection.commit()
            return True
        except Exception as e:
            print(f"Error creating incident: {e}")
            return False

    def update_incident_status(self, status, incident_id):
        try:
            cursor = self.connection.cursor()
            query = "UPDATE Incident SET status = %s WHERE incident_id = %s"
            cursor.execute(query, (status, incident_id))
            self.connection.commit()
            return True
        except Exception as e:
            print(f"Error updating incident status: {e}")
            return False

    def get_incidents_in_date_range(self, start_date, end_date):
        try:
            cursor = self.connection.cursor()
            query = "SELECT * FROM Incident WHERE date BETWEEN %s AND %s"
            cursor.execute(query, (start_date, end_date))
            return cursor.fetchall()
        except Exception as e:
            print(f"Error fetching incidents in date range: {e}")
            return []

    def search_incidents(self, incident_type):
        try:
            cursor = self.connection.cursor()
            query = "SELECT * FROM Incident WHERE incident_type = %s"
            cursor.execute(query, (incident_type,))
            return cursor.fetchall()
        except Exception as e:
            print(f"Error searching incidents: {e}")
            return []

    def generate_incident_report(self, incident):

        return {
            "incident_id": incident.incident_id,
            "summary": f"Report for incident {incident.incident_id} - {incident.description}"
        }

    def create_case(self, case_description, incidents):
        try:
            cursor = self.connection.cursor()
            query = "INSERT INTO Cases (description) VALUES (%s)"
            cursor.execute(query, (case_description,))
            case_id = cursor.lastrowid

            for incident in incidents:
                link_query = "INSERT INTO Case_Incident (case_id, incident_id) VALUES (%s, %s)"
                cursor.execute(link_query, (case_id, incident.incident_id))

            self.connection.commit()
            return {"case_id": case_id, "description": case_description}
        except Exception as e:
```

```
            print(f"Error creating case: {e}")
            return None

    def get_case_details(self, case_id):
        try:
            cursor = self.connection.cursor()
            query = "SELECT * FROM Cases WHERE case_id = %s"
            cursor.execute(query, (case_id,))
            return cursor.fetchone()
        except Exception as e:
            print(f"Error fetching case details: {e}")
            return None

    def update_case_details(self, case_obj):
        try:
            cursor = self.connection.cursor()
            query = "UPDATE Cases SET description = %s WHERE case_id = %s"
            cursor.execute(query, (case_obj.description, case_obj.case_id))
            self.connection.commit()
            return True
        except Exception as e:
            print(f"Error updating case: {e}")
            return False

    def get_all_cases(self):
        try:
            cursor = self.connection.cursor()
            query = "SELECT * FROM Cases"
            cursor.execute(query)
            return cursor.fetchall()
        except Exception as e:
            print(f"Error fetching all cases: {e}")
            return []
```

TASK

Exception Handling

1) Create the exceptions in package myexceptions
2) Define the following custom exceptions and throw them in methods whenever needed. Handle all the exceptions in main method,
3) IncidentNumberNotFoundException :throw this exception when user enters an invalid patient number which doesn't exist in db

Folder: myexceptions

This package is responsible for handling all custom exceptions that may occur during the execution of the program. It improves error clarity and robustness by providing user-defined exceptions instead of generic errors.

File: incident_number_not_found_exception.py

- **Purpose:**

This file defines a custom exception class called IncidentNumberNotFoundException.

- **When to Use:**

Throw this exception when the system fails to find an incident based on a given incident number — i.e., when the incident ID does not exist in the database.

- **Functional Flow:**

  ➢ Definition (Inmyexceptions/incident_number_not_found_exception.py)

  ➢ Create a class that inherits from Python's built-in Exception class.

  ➢ Include a message parameter that can be passed while throwing the exception.

  ➢ This makes it easy to identify the specific issue encountered.

- **Where to Throw It (Inside dao/crime_analysis_service_impl.py)**

  ➢ In methods like updateIncidentStatus() or getCaseDetails() where an incident ID is expected to be found, check if it exists.
  ➢ If not, raise IncidentNumberNotFoundException.

- **Where to Handle It (Inside main/main_module.py)**

  ➢ Surround the service method calls with try-except blocks.

  ➢ Catch this specific exception and print user-friendly error messages like:"Incident with ID 105 not found. Please check the ID and try again.″

incident_number_not_found_exception.py

```python
class IncidentNumberNotFoundException(Exception):
    def __init__(self, incident_id):
        super().__init__(f"Incident with ID {incident_id} not found in the database.")
```

TASK

Main Method

Create class named MainModule with main method in main package.
Trigger all the methods in service implementation class

Folder: main

This package contains the driver class that runs your entire application. It acts as a bridge between the user interface (console-based in this case) and the backend logic written in your service classes.

main_module.py

```python
from dao.crime_analysis_service_impl import CrimeAnalysisServiceImpl
from entity.incident import Incident
from datetime import datetime
from myexceptions.incident_number_not_found_exception import
IncidentNumberNotFoundException


def main():
    service = CrimeAnalysisServiceImpl()

    try:
        print("\n Creating a sample incident...")
        incident = Incident(
            incident_id=1,
            incident_type="Robbery",
            description="Robbery at SBI bank in Delhi",
            date=datetime.strptime("2024-02-15", "%Y-%m-%d"),
            status="Open"
        )

        created = service.create_incident(incident)
        print("Incident created!" if created else "Failed to create incident.")

        print("\n Fetching all cases...")
        cases = service.get_all_cases()

        if cases:
            for case in cases:
                print(f" Case ID: {case.case_id}, Description: {case.case_description}")
        else:
            print("No cases found.")

    except IncidentNumberNotFoundException as e:
        print(f"Error: {e}")

    except Exception as e:
        print("An unexpected error occurred:", e)


if __name__ == "__main__":
    main()
```

TASK

Unit Testing

Creating PythonUnit test cases for a **Crime Analysis and Reporting System** is essential to ensure the correctness and reliability of your system. Below are some example questions to guide the creation of PythonUnit test cases for various components of the system:

Unit testing is a crucial part of ensuring the functionality and reliability of the Crime Analysis and Reporting System. We have created unit tests using Python's unittest module to test the core features of the application, including incident creation and status updates.

Test Case 1: Incident Creation

Objective:
To verify whether the createIncident() method correctly creates an incident with the provided attributes.

Test Scenarios:

➢ Check if a valid incident object can be created.
➢ Validate that all attributes are stored correctly in the created object.

Test Code:

```
def test_create_incident(self):
    incident = Incident(
        incident_id=101,
        incident_type="Robbery",
        description="Stolen wallet",
        date=datetime.strptime("2025-04-01", "%Y-%m-%d"),
        location="Mumbai",
        status="Open"
    )
    result = self.service.createIncident(incident)
    self.assertTrue(result)
```

Test Case 2: Incident Status Update (Valid)

Objective:
To ensure the updateIncidentStatus() method correctly updates the status of a valid incident.

Test Scenarios:

➢ Update an existing incident's status to a new valid state.
➢ Ensure the change is reflected in the database or data model.

Test Code

```
def test_update_incident_status_valid(self):
    status = Status(status_id=1, incident_id=101, status="Closed")
    result = self.service.updateIncidentStatus(status, 101)
    self.assertTrue(result)
```

Test Case 3: Incident Status Update (Invalid ID)

Objective:
To ensure the updateIncidentStatus() method raises the appropriate exception when given an invalid incident ID.

Test Scenarios:

Pass an incident ID that does not exist in the database.

Expect a custom exception (IncidentNumberNotFoundException) to be thrown.

Test Code:

```python
def test_update_incident_status_invalid_id(self):
    status = Status(status_id=2, incident_id=999, status="Closed")
    with self.assertRaises(IncidentNumberNotFoundException):
        self.service.updateIncidentStatus(status, 999)
```

```
C:\Users\deept\PycharmProjects\pythonProject1\.venv\Scripts\python.exe C:\Users\deept\PycharmProjects\pythonProject1\test\test_crime_analysis_service.py
test_create_incident (__main__.TestCrimeAnalysisService) ... ok
test_update_incident_status_invalid_id (__main__.TestCrimeAnalysisService) ... ok
test_update_incident_status_valid (__main__.TestCrimeAnalysisService) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.005s
OK

Process finished with exit code 0
```

1. Incident Creation:

Does the createIncident method correctly create an incident with the provided attributes?
Yes, it creates the incident as expected using the provided object.

Are the attributes of the created incident accurate?
Yes, the values are correctly stored and can be retrieved using getters.

2. Incident Status Update:

Does the updateIncidentStatus method effectively update the status of an incident?
Yes, it updates the status successfully in the database.

Does it handle invalid status updates appropriately?
Yes, when an invalid incident ID is passed, the method throws a custom exception (IncidentNumberNotFoundException) as expected.

Crime Analysis and Reporting System (C.A.R.S.)

crime_analysis_console.py

```python
import mysql.connector
from datetime import datetime


def connect_db():
    return mysql.connector.connect(
        host="localhost",
        user="root",
        password="root",
        database="CrimeAnalysis"
    )


def validate_victim_suspect(victim_id, suspect_id):
    conn = connect_db()
    cursor = conn.cursor()

    cursor.execute("SELECT VictimID FROM Victims WHERE VictimID = %s", (victim_id,))
    victim_exists = cursor.fetchone()

    cursor.execute("SELECT SuspectID FROM Suspects WHERE SuspectID = %s", (suspect_id,))
    suspect_exists = cursor.fetchone()

    conn.close()
    return victim_exists is not None, suspect_exists is not None


def assign_officer():
    conn = connect_db()
    cursor = conn.cursor()
    cursor.execute("SELECT OfficerID FROM Officers LIMIT 1")  # Dummy logic for now
    result = cursor.fetchone()
    conn.close()
    return result[0] if result else None


def generate_report(incident_id, officer_id):
    conn = connect_db()
    cursor = conn.cursor()
    report_date = datetime.now().date()
    report_details = "Initial report generated."
    status = "Open"

    query = """
    INSERT INTO Reports (IncidentID, ReportingOfficer, ReportDate, ReportDetails, Status)
    VALUES (%s, %s, %s, %s, %s)
    """
    cursor.execute(query, (incident_id, officer_id, report_date, report_details, status))
    conn.commit()
    conn.close()


def report_crime():
    print("\n Please enter the following details to report a crime:")
    incident_type = input("Incident Type: ")
```

```python
    incident_date = input("Incident Date (YYYY-MM-DD): ")
    location = input("Location: ")
    description = input("Description: ")
    status = input("Status: ")
    victim_id = input("Victim ID: ")
    suspect_id = input("Suspect ID: ")

    valid_victim, valid_suspect = validate_victim_suspect(victim_id, suspect_id)
    if not valid_victim:
        print("Invalid Victim ID. Please add the victim to the database first.")
        return
    if not valid_suspect:
        print("Invalid Suspect ID. Please add the suspect to the database first.")
        return

    conn = connect_db()
    cursor = conn.cursor()
    cursor.execute("""
    SELECT * FROM Incidents
    WHERE IncidentType = %s AND IncidentDate = %s AND Location = %s
    """, (incident_type, incident_date, location))

    if cursor.fetchone():
        print("Incident already reported.")
        conn.close()
        return

    query = """
    INSERT INTO Incidents (IncidentType, IncidentDate, Location, Description, Status, VictimID,
SuspectID)
    VALUES (%s, %s, %s, %s, %s, %s, %s)
    """
    cursor.execute(query, (incident_type, incident_date, location, description, status, victim_id,
suspect_id))
    conn.commit()
    incident_id = cursor.lastrowid
    conn.close()

    print("Crime reported successfully!")

    officer_id = assign_officer()
    if officer_id:
        print(f"Case assigned to Officer ID: {officer_id}")
        generate_report(incident_id, officer_id)
        print("Report generated and stored.")
    else:
        print("No officer available to assign.")


def main_menu():
    while True:
        print("\nWelcome to the Crime Reporting System")
        print("1. View Incident Details")
        print("2. View Victim Details")
        print("3. View Suspect Details")
        print("4. View Officer Details")
        print("5. Report a Crime")
        print("6. Exit")

        choice = input("Enter your choice (1-6): ")
```

```python
    if choice == '1':
        view_incident_details()
    elif choice == '2':
        view_victim_details()
    elif choice == '3':
        view_suspect_details()
    elif choice == '4':
        view_officer_details()
    elif choice == '5':
        report_crime()
    elif choice == '6':
        print("Thank you. Stay safe!")
        break
    else:
        print("Invalid choice. Please enter a number between 1 and 6.")

main_menu()
```

OUTPUT

➢ The Crime Analysis and Reporting System displays with the Welcome message and displays 6 options and asks to enter your choice

```
Welcome to the Crime Reporting System
1. View Incident Details
2. View Victim Details
3. View Suspect Details
4. View Officer Details
5. Report a Crime
6. Exit
Enter your choice (1-6):
```

➢ When choice 1 is entered it asks for the incident id to display the information about the incident

```
Welcome to the Crime Reporting System
1. View Incident Details
2. View Victim Details
3. View Suspect Details
4. View Officer Details
5. Report a Crime
6. Exit
Enter your choice (1-6): 1
Enter Incident ID: 7

--- Incident Details ---
Type: Fraud
Date: 2024-07-05
Location: Bangalore, India
Description: Ranbir Kapoor was scammed in a real estate deal.
Status: Under Investigation
Victim ID: 7
Suspect ID: 7
```

> When choice 2 is entered it asks for the victim id to display the information about the victim

```
Welcome to the Crime Reporting System
1. View Incident Details
2. View Victim Details
3. View Suspect Details
4. View Officer Details
5. Report a Crime
6. Exit
Enter your choice (1-6): 2
Enter Victim ID: 8

--- Victim Details ---
Name: Hrithik Roshan
DOB: 1974-01-10
Gender: Male
Contact: hrithik@bollywood.com
```

> When choice 3 is entered it asks for the suspect id to display the information about the suspect

```
Welcome to the Crime Reporting System
1. View Incident Details
2. View Victim Details
3. View Suspect Details
4. View Officer Details
5. Report a Crime
6. Exit
Enter your choice (1-6): 3
Enter Suspect ID: 2

--- Suspect Details ---
Name: Leonardo DiCaprio
DOB: 1974-11-11
Gender: Male
Contact: leo@hollywood.com
```

> When choice 4 is entered it asks for the officers id to display the information about the officer

```
Welcome to the Crime Reporting System
1. View Incident Details
2. View Victim Details
3. View Suspect Details
4. View Officer Details
5. Report a Crime
6. Exit
Enter your choice (1-6): 4
Enter Officer ID: 1

--- Officer Details ---
Name: Sheriff Woody
Badge Number: TX1001
Rank: Sergeant
Contact: woody@lapd.com
Agency ID: 2
```

➢ When choice 5 is selected, the system prompts the user to answer specific questions related to the crime. Once the required details are provided, the crime is recorded in the database and an officer is assigned to handle the case.

```
Welcome to the Crime Reporting System
1. View Incident Details
2. View Victim Details
3. View Suspect Details
4. View Officer Details
5. Report a Crime
6. Exit
Enter your choice (1-6): 5

 Please enter the following details to report a crime:
I  ident Type: Art Theft
Incident Date (YYYY-MM-DD): 2025-04-06
Location: Srinagar
Description: A priceless painting vanishes overnight from a local gallery under mysterious circumstances.
Status: Case Registered
Victim ID: 4
Suspect ID: 8
Crime reported successfully!
Case assigned to Officer ID: 4
Report generated and stored.
```

And the newly added crime gets updated in the database

✓ Incidents Table

IncidentType	IncidentDate	Location	Description	Status	VictimID	SuspectID
Robbery	2024-01-05	Mumbai, India	Shah Rukh Khan was robbed at a film set.	Under Investigation	1	1
Homicide	2024-02-10	Delhi, India	Aamir Khan was found dead in a hotel room.	Closed	2	2
Theft	2024-03-15	Chennai, India	Deepika Padukone reported her jewelry stolen.	Open	3	3
Kidnapping	2024-04-20	Hyderabad, India	Hrithik Roshan was kidnapped and later found.	Closed	4	4
Assault	2024-05-25	Pune, India	Salman Khan was assaulted at an event.	Under Investigation	5	5
Burglary	2024-06-30	Kolkata, India	Alia Bhatt's house was broken into.	Open	6	6
Fraud	2024-07-05	Bangalore, India	Ranbir Kapoor was scammed in a real estate deal.	Under Investigation	7	7
Murder	2024-08-10	Jaipur, India	Kareena Kapoor found dead in her apartment.	Closed	8	8
Cyber Crime	2024-09-15	Lucknow, India	Ranveer Singh's social media was hacked.	Open	9	9
Drug Possession	2024-10-20	Ahmedabad, India	Ajay Devgn caught with illegal substances.	Under Investigation	10	10
Robbery	2024-10-05	hyderabad	the bank was robbered	open	5	6
Art Theft	2025-04-06	Srinagar	A priceless painting vanishes overnight from a l...	Case Registered	4	8
NULL	NULL	NULL	NULL	NULL	NULL	NULL

✓ Reports Table

	ReportID	IncidentID	ReportingOfficer	ReportDate	ReportDetails	Status
▶	1	1	1	2024-01-06	Initial investigation started. CCTV footage bein...	Under Investigation
	2	2	2	2024-02-11	Autopsy completed. Case closed as confirmed h...	Closed
	3	3	3	2024-03-16	Theft reported. No suspects identified yet.	Open
	4	4	4	2024-04-21	Victim found safe. Kidnappers in custody.	Closed
	5	5	5	2024-05-26	Suspect identified from event footage. Arrest p...	Under Investigation
	6	6	6	2024-07-01	Entry through window confirmed. Forensics und...	Open
	7	7	7	2024-07-06	Fraudulent documents submitted. Case under r...	Under Investigation
	8	8	8	2024-08-11	Victim's death ruled as murder. Investigation co...	Closed
	9	9	9	2024-09-16	Hacking traced to overseas server. Cyber team...	Open
	10	10	10	2024-10-21	Drugs found in suspect's vehicle. Lab results aw...	Under Investigation
	12	14	4	2025-04-08	Initial report generated.	Open
*	NULL	NULL	NULL	NULL	NULL	NULL

➢ When choice 6 is entered Thankyou message is displayed and gets exited.

```
Welcome to the Crime Reporting System
1. View Incident Details
2. View Victim Details
3. View Suspect Details
4. View Officer Details
5. Report a Crime
6. Exit
Enter your choice (1-6): 6
Thank you. Stay safe!

Process finished with exit code 0
```

FUTURE ENHANCEMENTS

In the future, this Crime Analysis and Reporting System can be enhanced with several features to improve functionality, usability, and scalability. One potential enhancement is the integration of a graphical user interface (GUI) or a web-based interface to make the system more user-friendly and accessible to non-technical users. Adding user authentication and role-based access control can ensure data privacy and restrict access based on user privileges such as admin, police officers, or analysts.

Moreover, the system can be extended to include geographical data mapping, using tools like Google Maps API, to visualize crime incidents based on location for better analysis and decision-making. Integration with machine learning algorithms can enable predictive analytics, helping law enforcement agencies identify crime hotspots and patterns.

Another valuable enhancement would be the implementation of notification and alert mechanisms, such as SMS or email notifications for critical incidents or case updates. Additionally, the project can be expanded to support real-time data updates and connections with external systems like police databases or emergency services for seamless information sharing.

Finally, incorporating a report dashboard with visual analytics using tools like Power BI or Tableau can provide deeper insights into crime trends, helping authorities take proactive measures. These enhancements would make the system more robust, efficient, and applicable in real-world crime analysis and reporting environments.

CONCLUSION

The **Crime Analysis and Reporting System** project has been successfully developed using a modular and layered architecture, ensuring scalability, maintainability, and real-world applicability. The system is designed to manage and analyze criminal activities efficiently, allowing users to register incidents, update statuses, manage cases, and generate reports.

The core business logic is defined through the interface in icrime_analysis_service.py, which outlines all essential functionalities like incident creation, status updates, date-wise searches, case generation, and report generation. This interface is implemented in crime_analysis_service_impl.py, where all the logic is built using robust coding practices and proper database handling.

The entity classes — incident.py, case.py, and status.py — encapsulate the data with private variables, constructors, getters, and setters, promoting data abstraction and reusability. These classes form the backbone of the system, representing real-world objects in a digital format.

The application connects to a SQL database using db_connection.py, which handles the database connection. It relies on property_util.py to read credentials and configuration from a properties file, keeping the application secure and configuration-based. This approach ensures flexibility and protects sensitive data like usernames and passwords.

To handle exceptions effectively and maintain system stability, a custom exception class — incident_number_not_found_exception.py — is created. This is thrown when an invalid or non-existent incident number is accessed. This makes the system more user-friendly and reliable.

All system functionalities are triggered from main_module.py, which acts as the entry point to the application. This script is responsible for interacting with the user and calling the appropriate service methods to perform the desired operations.

Finally, the entire application was validated through unit testing to ensure that key functionalities like incident creation and status updates perform accurately. Proper test cases were written and executed to verify correctness and robustness under different input scenarios.

In conclusion, this project integrates clean object-oriented programming, SQL-based database interaction, exception handling, and modular design to create a powerful and practical Crime Analysis and Reporting System. With potential future enhancements like a graphical user interface or web-based dashboard, this system can evolve into a full-fledged crime data management platform.