**Emergency Response System - Short Report**

**Object-Oriented Programming (OOP) Concepts Applied**

**Abstraction:** The EmergencyUnit class is an abstract class that defines the common properties and methods (such as Name, Speed, CanHandle, and RespondToIncident) for all emergency response units. This allows for hiding complex implementation details from the user.

**Inheritance:** Classes like Police, Firefighter, Ambulance, RescueTeam, and Hazmat inherit from the EmergencyUnit abstract class. Each subclass implements the abstract methods in a way specific to its responsibilities.

**Polymorphism:** The RespondToIncident method is implemented differently in each subclass, allowing the program to call the same method name on different types of units and get behavior specific to that unit.

**Encapsulation:** The properties (Name, Speed) and methods of the EmergencyUnit and Incident classes are kept together and managed internally, exposing only what is necessary for outside classes.

**Composition:** The Program class uses a list of EmergencyUnit objects (units list), allowing the system to manage multiple units dynamically.

**Lessons Learned**

The importance of clear class responsibilities: Keeping each class focused on a single task (e.g., Police handles crime, Firefighter handles fire) simplifies development and maintenance.

Validating user inputs is crucial: Implementing checks for incident types and difficulty values prevented runtime errors and made the system more user-friendly.

Small usability improvements make a big difference: Allowing users to retry after invalid login attempts and supporting continuous reporting in manual mode greatly improved the user experience.

Maintaining consistency in data formats (like converting incident types to lowercase) avoids subtle bugs when matching user inputs against predefined types.

**Challenges Faced**

**User Interaction Design:** Designing a clean flow where users could easily switch between automatic and manual incident reporting without restarting the program required careful structuring of loops and conditionals.

**Error Handling:** Managing invalid inputs (e.g., wrong usernames/passwords, invalid incident types, difficulty out of range) gracefully without crashing the program was initially tricky but improved after iterative testing.

**Balancing Simplicity and Flexibility:** Adding enough functionality (like multiple unit types, dynamic difficulty, scoring system) while keeping the system simple and readable was a challenge.