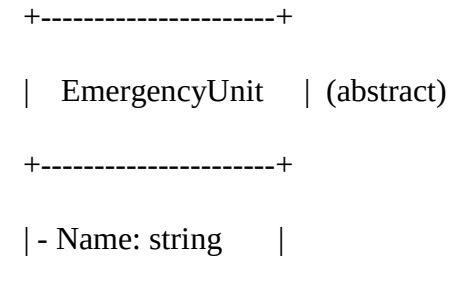
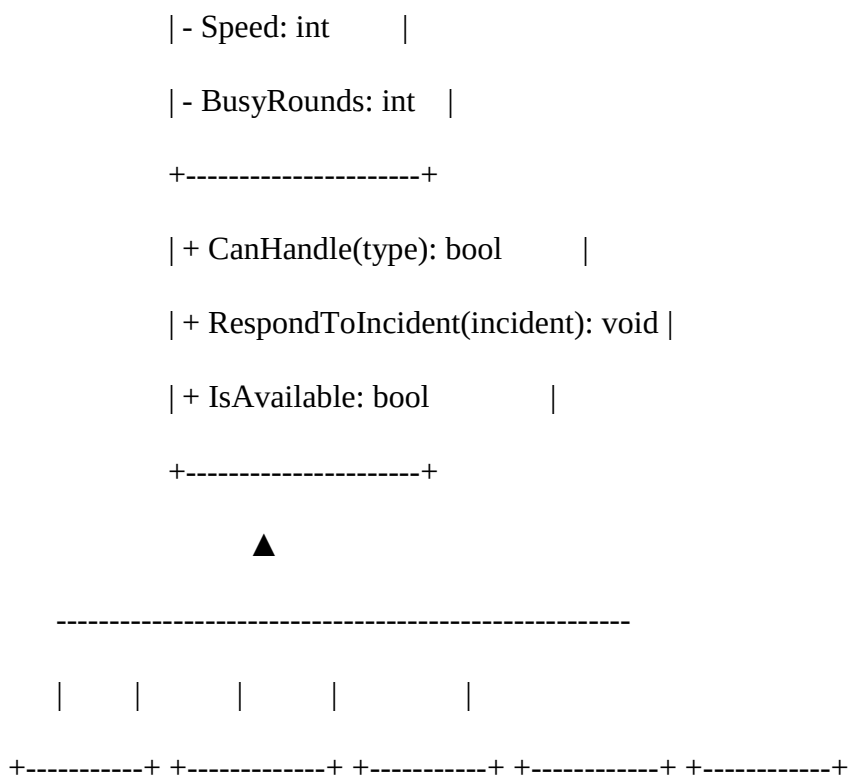
OOP Concepts Applied:

1. **Abstraction**:
    - The **`EmergencyUnit`** class is abstract and provides a general blueprint for all types of emergency units. It hides the implementation details of how each unit responds to an incident but exposes essential methods like `CanHandle()` and `RespondToIncident()` which are overridden by each specific unit type (Police, Firefighter, Ambulance, etc.).
2. **Inheritance**:
    - The **`Police`**, **`Firefighter`**, **`Ambulance`**, **`Hazmat`**, and **`CyberUnit`** classes **inherit** from the abstract **`EmergencyUnit`** class. This allows them to share common properties and methods (like `Name`, `Speed`, and `RespondToIncident()`) while also enabling each subclass to implement its own specific behavior (such as handling different types of incidents).
3. **Polymorphism**:
    - Polymorphism is demonstrated by the **`CanHandle()`** and **`RespondToIncident()`** methods, which are overridden in each subclass. The base class provides a common interface, but each unit type can handle incidents in its own way (e.g., the Police unit handles only "Crime", while the Firefighter unit handles only "Fire"). This allows the program to interact with different unit types in a uniform way without needing to know their exact details.
4. **Encapsulation**:
    - Properties such as `Speed`, `Name`, and `BusyRounds` are **encapsulated** within each unit class. These properties are protected from direct modification by other classes, ensuring that units can only change their internal state through controlled methods. For example, the **`IsAvailable`** property tracks whether a unit is currently responding to an incident, encapsulating its availability status.

These OOP concepts come together to create a flexible, scalable simulation of an emergency response system that models the interaction between different types of units and their response to various incidents.

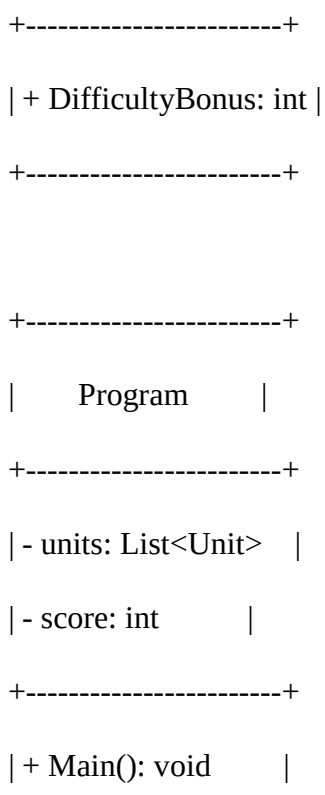
A Simple Class Diagram






```
                | - Speed: int        |
                | - BusyRounds: int   |
                +---------------------+
                | + CanHandle(type): bool         |
                | + RespondToIncident(incident): void |
                | + IsAvailable: bool             |
                +---------------------+
                          ▲
    ------------------------------------------------------
    |       |         |         |             |
+-----------+ +-------------+ +-----------+ +------------+ +------------+
|  Police   | | Firefighter | | Ambulance | |   Hazmat   | |  CyberUnit |
+-----------+ +-------------+ +-----------+ +------------+ +------------+
| Handles:  | | Handles:    | | Handles:  | | Handles:   | | Handles:   |
| "Crime"   | | "Fire"      | | "Medical" | | "Chemical" | | "Cyber"    |
+-----------+ +-------------+ +-----------+ +------------+ +------------+


                +-----------------------+
                |      Incident         |
                +-----------------------+
                | - Type: string        |
                | - Location: string    |
                | - Difficulty: string  |
```



🧠 Notes:

- All specific units (like Police, Hazmat, etc.) inherit from `EmergencyUnit`.
- The `Incident` class represents emergencies with a type, location, and difficulty.
- The `Program` contains and manages all the logic for simulating the rounds.

📓 Lessons Learned & Challenges Faced

☑ Lessons Learned

1. **The Power of Polymorphism**
   Using polymorphism made the system extremely flexible. We could loop through all emergency units and call `CanHandle()` or `RespondToIncident()` without worrying about the exact type of unit.
2. **Designing with Abstraction Simplifies Code**
   Abstracting common behavior in the `EmergencyUnit` base class reduced redundancy and made the system easier to extend (e.g., adding new unit types like Hazmat or CyberUnit).

3. **Scalability Through OOP**
   The use of OOP principles made it easy to scale the project—adding new unit types or incident categories required very minimal code changes.
4. **Simulation Design is a Real-World Fit for OOP**
   Modeling real-world systems (like emergency response) maps naturally to object-oriented structures, reinforcing the relevance of these concepts.

⚠ Challenges Faced

1. **Choosing a Fair Scoring System**
   Designing a balanced scoring mechanism based on response time, difficulty, and availability was tricky. It required tuning so that the game wasn't too easy or too punishing.

2. **Handling Unit Availability**
   Managing unit "cooldowns" or `BusyRounds` involved careful tracking to make sure units didn’t respond while already busy.

3. **Randomness vs. Control**
   Balancing the randomness of incident generation with predictable simulation flow took some thought. Too much randomness made testing harder; too little made the simulation feel static.

4. **Avoiding Code Duplication**
   Without good abstraction, code duplication crept in when adding more unit types. Inheritance helped solve this, but it highlighted the importance of designing the base class thoughtfully.