Institute of Geographical Information Systems

CS-212 - Object Oriented Programming LAB

Semester: Fall 2025	**Class: SCEE-IGIS - 2024**
Name: Ali Nawaz	**CMS ID : 00000526123**
Submitted to: Ma'am Alvina Anjum	**Due Date: Dec 17, 2025**

Open Ended LAB

Polymorphism in a Hospital Emergency Response and Monitoring System (HERMS) with Event-Driven Programming

Table of Contents

Objective

This lab provides hands-on experience with compile-time and run-time polymorphism using a real-life healthcare scenario. You will build a simplified version of a hospital emergency response and monitoring system (HERMS), where different medical units respond to hospital events such as:

- Low patient vitals
- Emergency code activation
- Equipment malfunction
- Medicine refill alerts
- Fire alarm triggers

These interactions rely on virtual functions, overriding, abstract interfaces, operator overloading, and event dispatching, integrating OOP principles with event-driven programming (EDP).

Learning Outcomes

After completing this lab, you should be able to:

1. Implement function overloading and operator overloading to demonstrate compile-time polymorphism.
2. Apply run-time polymorphism using abstract classes and virtual functions.
3. Build an event-driven architecture where hospital units react to triggered events.
4. Use base-class pointers/references to achieve dynamic binding.
5. Resolve conflict in polymorphism when multiple classes override functions.
6. Design an extensible real-life system applying OOP and EDP principles together.
7. Ensure safe memory deallocation using destructors (preferably virtual).

REAL-LIFE SCENARIO: Hospital Emergency Response & Monitoring System (HERMS)

A large hospital uses an internal system to automatically monitor patient vitals, equipment status, medicine inventory, and building safety. Events are triggered in real-time and different hospital units must respond appropriately.

Key hospital responders include:

- Nursing Station
- ICU Response Team
- Biomedical Engineering Unit
- Pharmacy System
- Fire & Safety Control Room
- Patient Monitoring Console

Your HERMS system will simulate this environment using polymorphism and event-driven logic.

Task 1: Compile-Time Polymorphism (Function + Operator Overloading)

1. Create a class **HospitalEvent** with overloaded constructors:
   - `HospitalEvent(string source)`
   - `HospitalEvent(string source, string eventType)`
   - `HospitalEvent(string source, int severityLevel)`
   - `HospitalEvent(string source, double vitalReading)`
2. Overload operators:
   - `<<` to print the event
   - `==` to check if two events come from the same source
   - `>` to compare event severity levels
3. Write a demo program generating various events such as:
   - `"Vitals Monitor", 89.6` (Low O2 reading)
   - `"FireSensor", "Code Red"`
   - `"ICU Monitor", 3` (High severity)
4. Explain any overloading ambiguity you encountered.

Task 2: Run-Time Polymorphism (Hospital Units Responding to Events)

1. Create an **abstract base class** `HospitalUnit` with:

```
virtual void respond(const HospitalEvent &e) = 0;
virtual ~HospitalUnit() {}
```

2. Create **at least five derived classes**, such as:
   - `NursingStation`
   - `ICUTeam`
   - `PharmacySystem`
   - `BiomedicalUnit`
   - `FireSafetyUnit`
3. Each class must override `respond()` differently.
   Example behaviours:
   - **NursingStation** → Sends nurse to patient room
   - **ICUTeam** → Dispatches critical care specialists
   - **PharmacySystem** → Auto-generate refill order
   - **BiomedicalUnit** → Schedules equipment maintenance
   - **FireSafetyUnit** → Locks fire doors + activates sprinklers
4. Demonstrate run-time polymorphism:

```
HospitalUnit* responders[5];
responders[0] = new NursingStation();
...
responders[i]->respond(event);
```

5. Show how dynamic binding allows different units to respond to the same event.

Task 3: Event-Driven Architecture (Core of HERMS)

1. Create an **EventDispatcher** class that stores registered responders:

```
void registerUnit(HospitalUnit* unit);
void dispatch(const HospitalEvent &e);
```

2. In `dispatch()`, loop through all units and call:

```
unit->respond(e);
```

3. Simulate real-time event flow:
   - Patient heartbeat drops
   - Room temperature rises
   - Ventilator malfunction
   - Fire alarm triggered
   - Medicine inventory below threshold

Task 4: Medication and Vital Monitoring

1. Create a base abstract class **VitalSensor**:

```
virtual HospitalEvent generateEvent() = 0;
```

2. Derive:
   - `HeartRateSensor`
   - `OxygenSensor`
   - `TemperatureSensor`
3. Override `generateEvent()` so each sensor produces different event types.
4. Show polymorphism by passing sensors through a single function:

```
void processSensor(VitalSensor* s) {
dispatcher.dispatch(s->generateEvent());
}
```

Task 5: Real-Life Problem Solving

Question 1:

How does polymorphism allow HERMS to scale when new hospital units or devices are added?

Answer:

Polymorphism allows us to add new types of hospital units (like a "SecurityUnit" or "DietaryUnit") without changing the core code of the EventDispatcher. The dispatcher only knows about the base class HospitalUnit. As long as the new unit inherits from the base class, the system accepts it immediately. This makes the system easy to expand.

Question 2:

Why would operator overloading be useful for event comparison or sorting?

Answer:

It makes the code more readable and intuitive. Instead of writing a function like event1.isMoreSevereThan(event2), we can simply write event1 > event2. This makes the code look like natural logic, which is easier for developers to maintain.

Question 3:

How could you integrate **Observer Pattern** into HERMS?

Answer:

The current EventDispatcher is actually already using a variation of the Observer Pattern. The EventDispatcher acts as the **Subject** (Publisher), and the HospitalUnits act as the **Observers** (Subscribers). When an event occurs (state change), the dispatcher notifies all registered units automatically.

Question 4:

How does EDP reduce complexity in emergency response systems?

Answer:

EDP decouples the system. The sensor that detects a fire doesn't need to know *who* puts out the fire or *how* to call them. It just broadcasts a "Fire Event." The logic for responding is handled separately by the listeners. This separation means if we change how the fire team responds, we don't have to touch the sensor code.

Question 5:

What would happen if virtual destructors were not used?

Answer:

If we delete a derived class object (like NursingStation) through a base class pointer (HospitalUnit*) without a virtual destructor, only the base class destructor is called. The derived class's destructor is skipped. This leads to **memory leaks** because resources allocated specifically by the derived class are never cleaned up.

Challenges Faced & Solutions

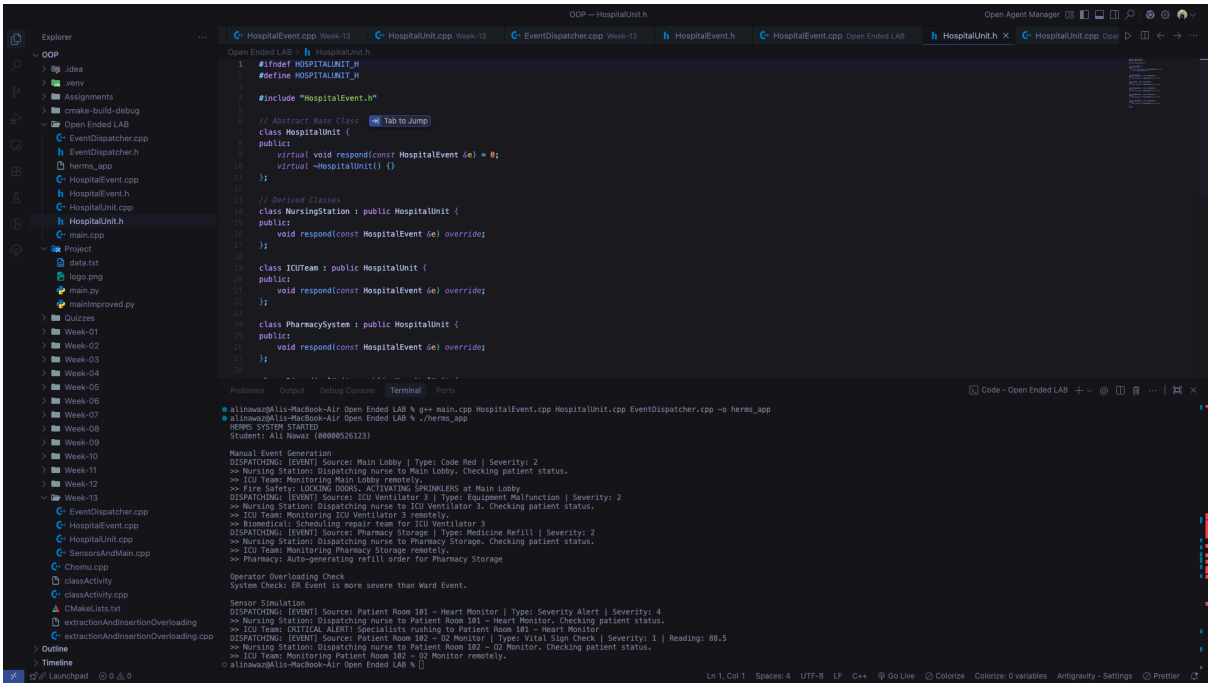
Ambiguity in Overloading: I encountered ambiguity when creating the HospitalEvent constructors. Specifically, distinguishing between int severity and double vitalReading.

Solution: The compiler distinguishes them by the type of number passed (e.g., 5 is int, 5.5 is double). To be safe, I ensured my test data used explicit types (integers for severity, decimals for readings).

Managing Multiple Responders: It was tricky to make sure the right unit responded to the right event.

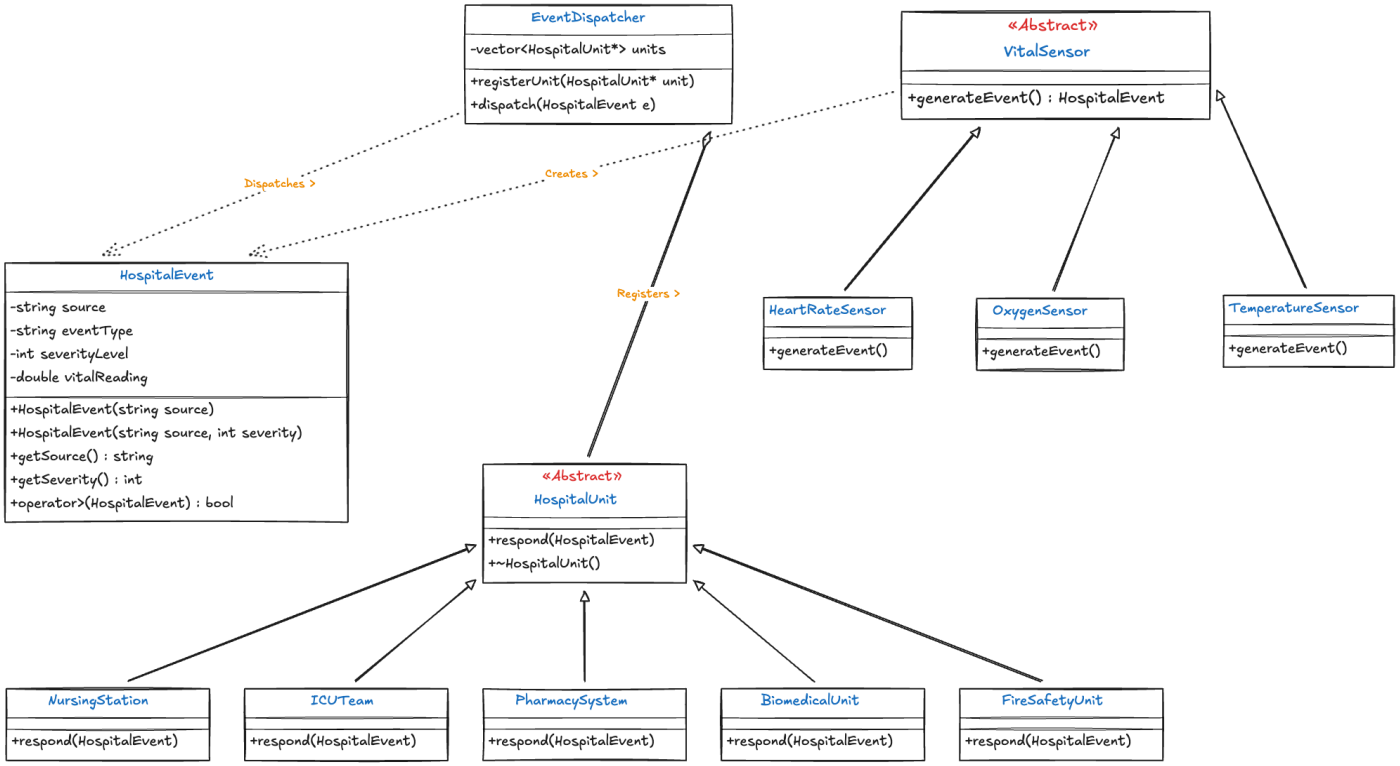
Solution: I implemented logic inside each respond() function (e.g., if (event.type == "Fire")). This ensures units ignore events that aren't relevant to them.

Output Screenshots



```
Problems  Output  Debug Console  Terminal  Ports

● alinawaz@Alis-MacBook-Air Open Ended LAB % g++ main.cpp HospitalEvent.cpp HospitalUnit.cpp EventDispatcher.cpp -o herms_app
● alinawaz@Alis-MacBook-Air Open Ended LAB % ./herms_app
  HERMS SYSTEM STARTED
  Student: Ali Nawaz (00000526123)

  Manual Event Generation
  DISPATCHING: [EVENT] Source: Main Lobby | Type: Code Red | Severity: 2
  >> Nursing Station: Dispatching nurse to Main Lobby. Checking patient status.
  >> ICU Team: Monitoring Main Lobby remotely.
  >> Fire Safety: LOCKING DOORS. ACTIVATING SPRINKLERS at Main Lobby
  DISPATCHING: [EVENT] Source: ICU Ventilator 3 | Type: Equipment Malfunction | Severity: 2
  >> Nursing Station: Dispatching nurse to ICU Ventilator 3. Checking patient status.
  >> ICU Team: Monitoring ICU Ventilator 3 remotely.
  >> Biomedical: Scheduling repair team for ICU Ventilator 3
  DISPATCHING: [EVENT] Source: Pharmacy Storage | Type: Medicine Refill | Severity: 2
  >> Nursing Station: Dispatching nurse to Pharmacy Storage. Checking patient status.
  >> ICU Team: Monitoring Pharmacy Storage remotely.
  >> Pharmacy: Auto-generating refill order for Pharmacy Storage

  Operator Overloading Check
  System Check: ER Event is more severe than Ward Event.

  Sensor Simulation
  DISPATCHING: [EVENT] Source: Patient Room 101 - Heart Monitor | Type: Severity Alert | Severity: 4
  >> Nursing Station: Dispatching nurse to Patient Room 101 - Heart Monitor. Checking patient status.
  >> ICU Team: CRITICAL ALERT! Specialists rushing to Patient Room 101 - Heart Monitor
  DISPATCHING: [EVENT] Source: Patient Room 102 - O2 Monitor | Type: Vital Sign Check | Severity: 1 | Reading: 88.5
  >> Nursing Station: Dispatching nurse to Patient Room 102 - O2 Monitor. Checking patient status.
  >> ICU Team: Monitoring Patient Room 102 - O2 Monitor remotely.
○ alinawaz@Alis-MacBook-Air Open Ended LAB %
```

UML Diagram



Conclusion

This OEL mirrors real-world hospital software used for emergency monitoring, patient safety, and equipment management. By integrating **polymorphism + event-driven programming**, you will learn how complex, mission-critical systems are built in industry.