Tribhuvan University

Institute of Science and Technology

Central Department of Computer Science and Information Technology

Case Study Report :

Simulation of Queuing System for Patan Hospital

Submitted By:
Name: Rejina Dahal
Roll No: 15

Submitted To:
Prof. Dr. Subarna Shakya
Tribhuvan University, IOE

June 14, 2025

Contents

1 Introduction **2**

2 Objective of the Case Study **3**

3 Description About Your Case Study **4**
3.1 Admission Counter System 4
3.2 Billing Counter System 4
3.3 Observed System Behavior 5

4 The Requirement Model **6**
4.1 Functional Requirements 6
4.2 Non-Functional Requirements 6
4.3 Use Case Model 7
4.3.1 Use Case Diagram 7
4.3.2 Use Case Descriptions 7

5 The Analysis Model **9**
5.1 Analysis Class Diagram 9
5.2 Class Descriptions 9
5.3 Dynamic Modeling: Sequence Diagram 10

6 Design Model **12**
6.1 Architectural Design: Event-Driven Simulation 12
6.2 Design Patterns 12
6.3 Design Class Diagram 12
6.4 Refined Sequence Diagram 13

7 Implementation Model **15**
7.1 Technology Stack 15
7.2 Realization of Design Components 16
7.2.1 Core Simulation Structure 16
7.2.2 Patient and Factory Implementation 16
7.2.3 Queuing Strategy Implementation 16
7.2.4 Statistics Collection 17

8 Conclusion and Recommendation **18**
8.1 Conclusion 18
8.2 Recommendations 18

1 Introduction

Hospitals and healthcare facilities are complex systems where the efficient management of patient flow is paramount to ensuring both patient satisfaction and operational effectiveness. A common and significant challenge within these environments is the management of queues, which often form at various service points such as admission and billing. Long waiting times can lead to patient frustration, decreased perceived quality of care, and inefficient use of hospital resources. This case study focuses on the queuing system at Patan Hospital, a prominent healthcare institution facing such challenges. The hospital currently operates a multi-queue, multi-counter system at two critical stages: the Admission counter, which has five dedicated queues (two for normal patients, one for staff, and two for insurance holders), and the Billing counter, which has six queues (three for normal, one for staff, and two for insurance). The current segregation of queues, while intended to streamline processes, may inadvertently create bottlenecks and uneven wait times across different patient categories.

To address these challenges, this study proposes the development of a discrete-event simulation model using an Object-Oriented Software Engineering (OOSE) approach. Simulation provides a powerful, cost-effective, and risk-free method to analyze and experiment with complex systems. By creating a virtual representation of the hospital's queuing process, we can measure key performance indicators such as average wait time, queue length, and counter utilization under the current *as-is* configuration. The object-oriented paradigm is particularly well-suited for this task, as it allows for the intuitive modeling of real-world entities like `Patients`, `Counters`, and `Queues` as software objects with distinct properties and behaviors. This simulation will serve as a foundation for exploring *what-if* scenarios, such as reallocating counters or modifying queuing disciplines, to identify an optimized *to-be* model that enhances efficiency and reduces patient wait times.

However, this project anticipates several key challenges that must be addressed to ensure the validity and usefulness of the simulation. The primary challenge lies in the acquisition of accurate and comprehensive data, including patient arrival rates for each category at different times of the day, and the distribution of service times at each counter. The model's accuracy is fundamentally dependent on the quality of this input data. A second challenge involves accurately modeling the complexities of human behavior, such as patients potentially choosing a shorter queue (*jockeying*) or leaving without service if the wait is too long (*balking*), which may require simplifying assumptions. Finally, validating the *as-is* model against real-world observations to ensure it is a credible representation of the Patan Hospital system will be a critical and meticulous step. Overcoming these challenges will be essential to delivering a robust analysis and providing actionable recommendations for improving the patient experience at Patan Hospital.

2 Objective of the Case Study

The primary goal of this case study is to apply Object-Oriented Software Engineering (OOSE) principles to model, simulate, and optimize the patient queuing system at Patan Hospital. The project aims to provide data-driven recommendations for improving operational efficiency and patient satisfaction.

The specific objectives to achieve this goal are as follows:

- **Analyze the Existing System:** To conduct a thorough analysis of the current (*as-is*) multi-queue, multi-counter system at both the Admission and Billing departments of Patan Hospital. This involves understanding the patient flow, the rules governing each queue, and identifying existing bottlenecks.

- **Develop a Simulation Model:** To design and develop a flexible discrete-event simulation model using an OOSE approach. The model must accurately represent the key entities of the system, including different patient types (`Normal`, `Staff`, `Insurance`), the dedicated `Queues`, and the service `Counters`.

- **Validate the Model:** To validate the simulation model to ensure its credibility. This will be achieved by comparing the model's output (e.g., average wait times, throughput) against real-world data or established benchmarks from Patan Hospital, ensuring the model is a reliable representation of the actual system.

- **Quantify System Performance:** To use the validated model to quantitatively measure the performance of the current *as-is* system. Key performance indicators (KPIs) to be measured include average waiting time, average queue length, and counter utilization for each patient category.

- **Evaluate Alternative Scenarios:** To propose and simulate various alternative queuing strategies (*what-if* scenarios) to improve the system's performance. These scenarios may include reallocating counters between patient types, merging queues of the same type (e.g., a single queue for all normal patients), or implementing a pooled queuing discipline.

- **Recommend an Optimized Model:** To recommend an optimized (*to-be*) system configuration based on the comparative results of the simulation scenarios. The final recommendation will aim to provide a practical solution that balances reduced patient wait times with efficient resource utilization for Patan Hospital.

3 Description About Your Case Study

Patan Hospital is a major healthcare provider that serves a diverse population of patients daily. To manage the high volume of visitors, the hospital has implemented a structured administrative workflow. This case study focuses on two critical choke-points in this workflow: the **Admission** process and the **Billing** process. The current system is designed to categorize patients into three distinct types—General Patients, Hospital Staff, and Insurance Patients—and direct them to dedicated service counters. The objective of this categorization is to streamline service, but it also introduces complexities into the queuing dynamics.

The typical patient journey through the administrative system begins upon arrival at the hospital. A patient must first register at an Admission counter. Following their medical consultation or procedure (the details of which are outside the scope of this simulation), the patient proceeds to a Billing counter to settle payments and finalize their visit. The core of this study is the analysis of the queuing systems at these two distinct stages.

3.1 Admission Counter System

The first point of contact for patients is the Admission department. This stage operates with a total of **five counters**. The system follows a strict multi-queue, multi-server model, where each counter has a dedicated queue and serves only one type of patient. Patients are required to join the queue corresponding to their category. The allocation is as follows:

General Patients are served by **two counters**, each with its own separate queue.

Hospital Staff are served by a single, dedicated **one counter** and queue.

Insurance Patients are served by **two counters**, also with separate, dedicated queues.

The purpose of this stage is to collect patient demographics, verify identity, and direct them to the appropriate medical services.

3.2 Billing Counter System

After receiving medical services, patients proceed to the Billing department to complete their visit. This department is larger, operating with a total of **six counters**. It mirrors the Admission department's structure with a dedicated multi-queue, multi-server system. The counter allocation is as follows:

General Patients are served by **three counters**, with three corresponding queues.

Hospital Staff are served by a single, dedicated **one counter** and queue.

Insurance Patients are served by **two counters**, each with its own queue.

Service at this stage involves processing payments, handling complex insurance claim paperwork, and providing final receipts.

The overall journey of a patient through these administrative stages is visualized in the activity diagram in Figure 1.

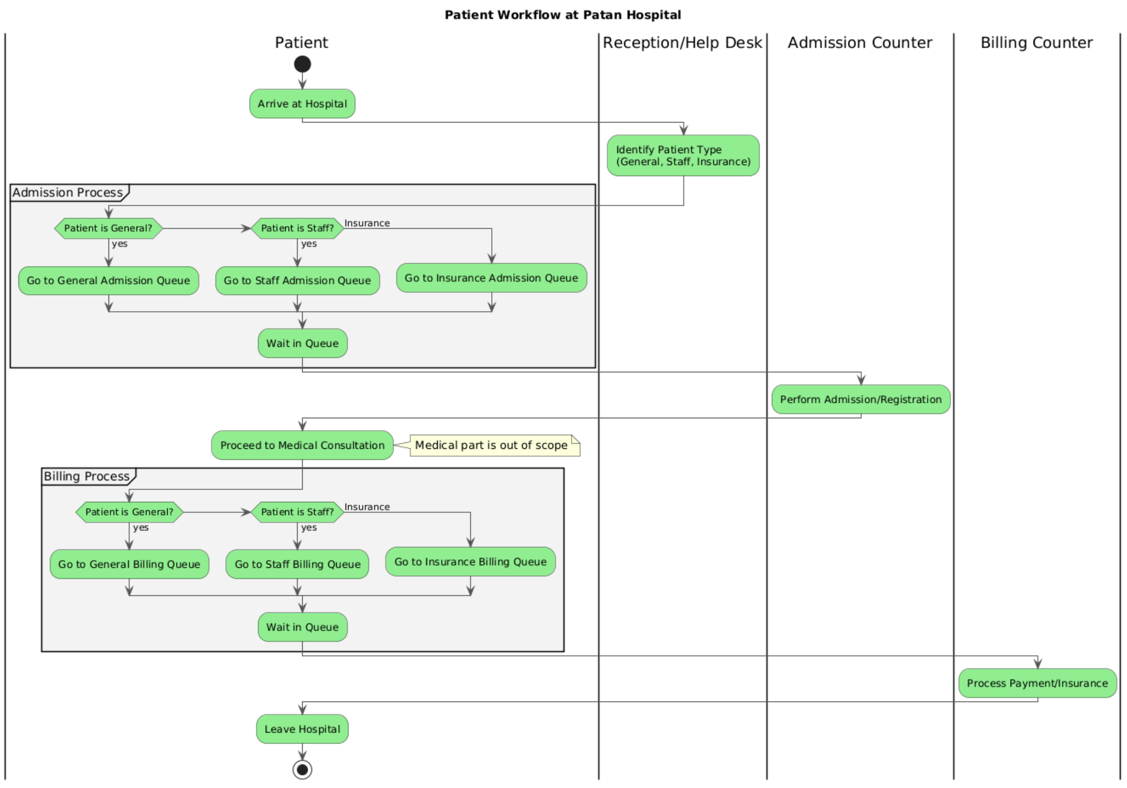
Patient Workflow at Patan Hospital
Patient
Reception/Help Desk
Admission Counter
Billing Counter
Arrive at Hospital
Identify Patient Type (General, Staff, Insurance)
Admission Process
Patient is General?
yes
Patient is Staff?
yes
Insurance
Go to General Admission Queue
Go to Staff Admission Queue
Go to Insurance Admission Queue
Wait in Queue
Perform Admission/Registration
Proceed to Medical Consultation
Medical part is out of scope
Billing Process
Patient is General?
yes
Patient is Staff?
yes
Insurance
Go to General Billing Queue
Go to Staff Billing Queue
Go to Insurance Billing Queue
Wait in Queue
Process Payment/Insurance
Leave Hospital

Figure 1: Activity Diagram

3.3 Observed System Behavior

The rationale behind this segregated queue structure is to provide specialized and potentially faster service to different patient groups. For example, insurance processing often requires more time and specific expertise. However, this *as-is* configuration leads to observable inefficiencies. It is common to see long queues for one patient type (e.g., General Patients) while counters for other types (e.g., Staff) are idle. This creates patient frustration and represents an underutilization of hospital resources. This case study will model this existing system to formally quantify these inefficiencies and explore alternative configurations.

4 The Requirement Model

The Requirement Model specifies the capabilities and constraints of the proposed hospital queuing simulation system. It serves as a bridge between the problem description and the software design, ensuring that the developed system meets the stated objectives. This model is composed of functional requirements, non-functional requirements, and a use case model detailing the interactions between the user and the system.

4.1 Functional Requirements

Functional requirements define what the system must do. For this simulation, they are:

FR1: System Configuration The user must be able to configure the simulation parameters, including the number of counters for each patient type, the patient arrival rates (e.g., patients per hour), and the service time distributions.

FR2: Patient Generation The system shall dynamically generate patient arrivals into the system based on a specified statistical distribution (e.g., Poisson process). Each patient must be assigned a type: `General`, `Staff`, or `Insurance`.

FR3: Queuing Logic The system must correctly direct arriving patients to their designated queues as per the *as-is* model (dedicated queues) or alternative *what-if* models (e.g., pooled queues).

FR4: Service Simulation The system shall simulate the service process at each counter. The service time for each patient should be drawn from a statistical distribution that can differ based on patient type (e.g., Insurance patients may have longer service times).

FR5: Data Collection The system must log key performance indicators (KPIs) in real-time during the simulation run. This includes:

- Time of arrival for each patient.
- Time service begins for each patient.
- Time service ends for each patient.
- Queue length at regular intervals.
- Counter status (busy/idle) over time.

FR6: Report Generation Upon completion of a simulation run, the system must process the collected data and generate a summary report containing key metrics, such as average/maximum wait time, average queue length, and overall counter utilization for each patient type.

4.2 Non-Functional Requirements

Non-functional requirements define the quality attributes of the system.

NFR1: Flexibility The system architecture must be modular and easily extensible to accommodate new queuing disciplines, patient types, or service stages with minimal changes to the core code.

NFR2: Accuracy The simulation's random number generation and statistical distributions must be implemented correctly to ensure the model's validity and reliability.

NFR3: Performance The simulation should execute efficiently, allowing for multiple runs with different parameters to be completed in a reasonable amount of time (e.g., a full day simulation should run in minutes).

NFR4: Usability The user interface for configuring the simulation and viewing results should be clear and intuitive for a hospital analyst or researcher.

4.3 Use Case Model

The use case model describes the system's functionality from a user's perspective. The primary actor is the **Hospital Analyst** who runs the simulation to study the system.

4.3.1 Use Case Diagram

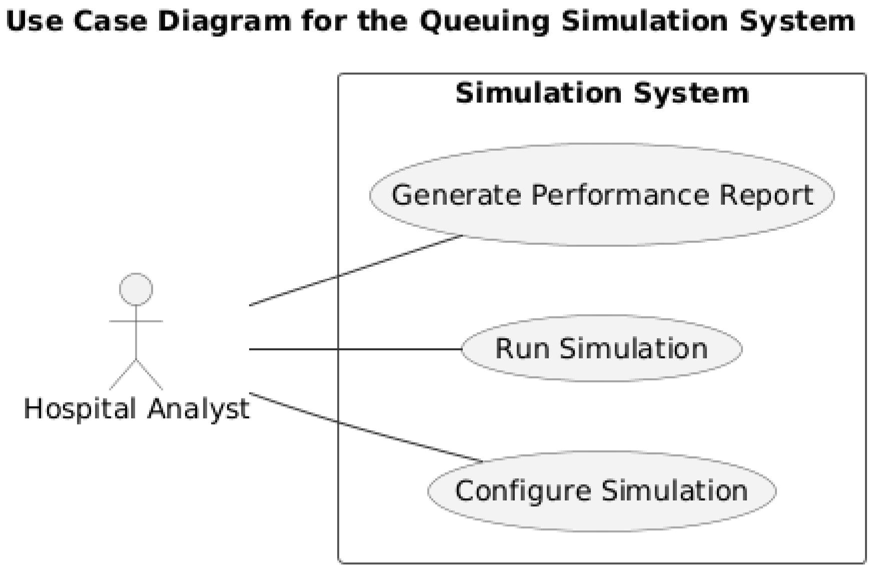
Use Case Diagram for the Queuing Simulation System
Simulation System
Generate Performance Report
Run Simulation
Configure Simulation
Hospital Analyst

Figure 2: Use Case Diagram

4.3.2 Use Case Descriptions

Here are brief descriptions of the primary use cases:

UC1: Configure Simulation

- **Actor:** Hospital Analyst
- **Description:** The analyst sets up the parameters for a simulation run. This includes defining the queuing discipline (e.g., dedicated vs. pooled queues), setting the number of counters for each type, and inputting patient arrival and service time data.
- **Post-condition:** The simulation model is ready to be executed with the specified configuration.

UC2: Run Simulation

- **Actor:** Hospital Analyst
- **Description:** The analyst initiates the simulation. The system evolves over a specified period (e.g., 8 hospital hours), processing patient arrivals, queuing, and service events. The system logs all relevant data during the run.
- **Pre-condition:** The simulation must be configured (UC1).
- **Post-condition:** The simulation run is complete, and raw performance data is stored.

UC3: Generate Performance Report

- **Actor:** Hospital Analyst
- **Description:** The analyst requests a summary of the simulation results. The system calculates and displays the key performance metrics (average wait time, utilization, etc.) in a clear format.
- **Pre-condition:** A simulation run must be completed (UC2).
- **Post-condition:** The analyst can view and analyze the performance of the simulated system configuration.

5 The Analysis Model

The Analysis Model translates the requirements into a preliminary software design. It focuses on identifying the primary classes, their responsibilities, and their interactions from the perspective of the problem domain. This model is intentionally kept at a high level of abstraction, avoiding implementation-specific details. It consists of a class diagram to represent the static structure and sequence diagrams to illustrate the dynamic behavior of the system.

5.1 Analysis Class Diagram

The class diagram identifies the core entities of the queuing system. These are the "things" that the system needs to manage: Patients, Counters, Queues, and the simulation environment itself. The diagram below illustrates these classes and their relationships, such as inheritance (for different patient types) and aggregation (for the composition of a service area).

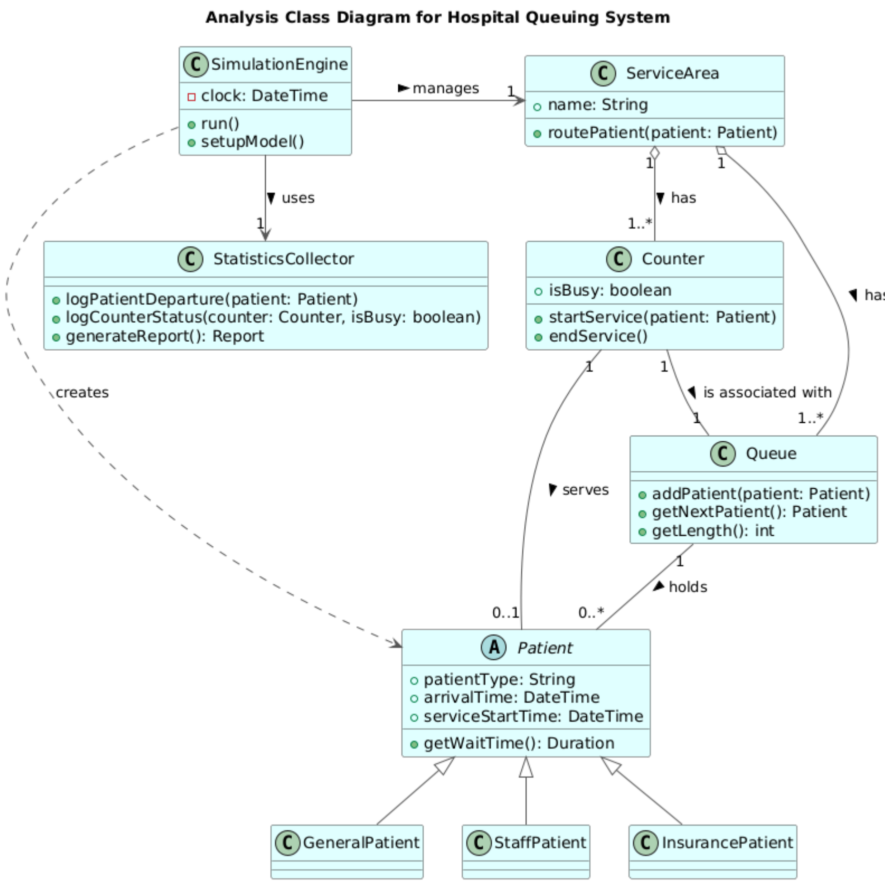

Figure 3: Class Diagram

5.2 Class Descriptions

The responsibilities of each key class are described below:

SimulationEngine The main driver class. It is responsible for initializing the system, managing the simulation clock, triggering events (like patient arrivals), and starting and stopping the simulation run.

Patient An abstract base class representing a person visiting the hospital. It holds common attributes like arrival time. Subclasses (`GeneralPatient`, `StaffPatient`, `InsurancePatient`) represent the specific types.

ServiceArea Represents a department where queuing occurs, such as ”Admission” or ”Billing”. It contains a collection of counters and queues and is responsible for routing an arriving patient to the correct queue.

Counter Represents a service point where a hospital employee serves a patient. It can be either busy or idle and is responsible for managing the service time for a single patient.

Queue A waiting line for a specific counter or group of counters. It holds patients in a first-in, first-out (FIFO) order and is responsible for adding and removing patients.

StatisticsCollector A utility class responsible for gathering data during the simulation. It logs key events (e.g., when a patient finishes waiting, when a counter becomes busy) and calculates the final performance metrics for the report.

5.3 Dynamic Modeling: Sequence Diagram

To understand how these classes collaborate, we can model a key scenario. The sequence diagram in Figure **??** illustrates the typical flow of events when a new patient arrives at a service area, is placed in a queue, and is eventually served by a counter.

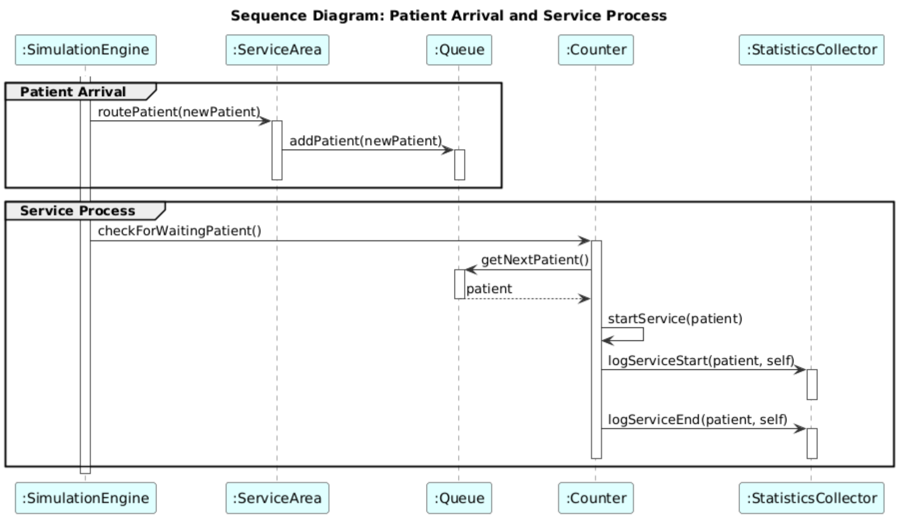

Figure 4: Sequence Diagram

This interaction shows the a clear separation of concerns:

- The `SimulationEngine` orchestrates the high-level events.
- The `ServiceArea` handles the business logic of routing a patient.
- The `Queue` and `Counter` manage the direct patient handling and service.

- The `StatisticsCollector` passively collects data as events occur.

This dynamic model confirms that the static class structure can fulfill the functional requirements defined earlier.

6 Design Model

The Design Model refines the Analysis Model into a concrete blueprint for implementation. It addresses the non-functional requirements such as flexibility and maintainability by defining the system's architecture and incorporating established design patterns. This model moves from the "what" of the analysis phase to the "how" of the solution domain.

6.1 Architectural Design: Event-Driven Simulation

A discrete-event simulation is best modeled using an **Event-Driven Architecture**. The core of this architecture is a central `SimulationEngine` that maintains a simulation clock and a priority queue of future events, known as the Future Event List (FEL).

The simulation proceeds by pulling the most imminent event from the FEL, advancing the clock to that event's time, and executing it. The execution of one event (e.g., a patient arrival) can trigger the scheduling of new future events (e.g., another patient arrival or a service completion). This design decouples the components, as they only interact through events, making the system highly modular. The main events are:

- `ArrivalEvent`: Represents a patient entering a service area.
- `ServiceCompletionEvent`: Represents a counter finishing service for a patient.

6.2 Design Patterns

To achieve the goals of flexibility and extensibility (NFR1), several design patterns are incorporated into the system design.

Factory Pattern To handle the creation of different patient types (`General`, `Staff`, `Insurance`), a `PatientFactory` class is used. This encapsulates the instantiation logic, decoupling the main simulation engine from the concrete patient classes. The engine can simply request a patient of a certain type without needing to know how it is created.

Strategy Pattern This is the key to evaluating different queuing models. The logic for how a patient chooses a queue is encapsulated within a `IQueuingStrategy` interface. Concrete classes like `DedicatedQueueStrategy` (for the *as-is* model) and `PooledQueueStrategy` (for a potential *to-be* model) implement this interface. The `ServiceArea` can be configured with any of these strategies at runtime, allowing for easy comparison of scenarios without changing the `ServiceArea` code.

Observer Pattern To collect statistics without cluttering the core simulation logic, the Observer pattern is used. The `StatisticsCollector` acts as an Observer. Core objects like `Counter` act as Observables. When a significant event occurs (e.g., a counter becomes free), it notifies all registered observers. This keeps the data collection logic separate and cohesive.

6.3 Design Class Diagram

The class diagram in Figure 5 shows the refined structure of the system, including the interfaces and classes that implement the chosen architectural style and design patterns.

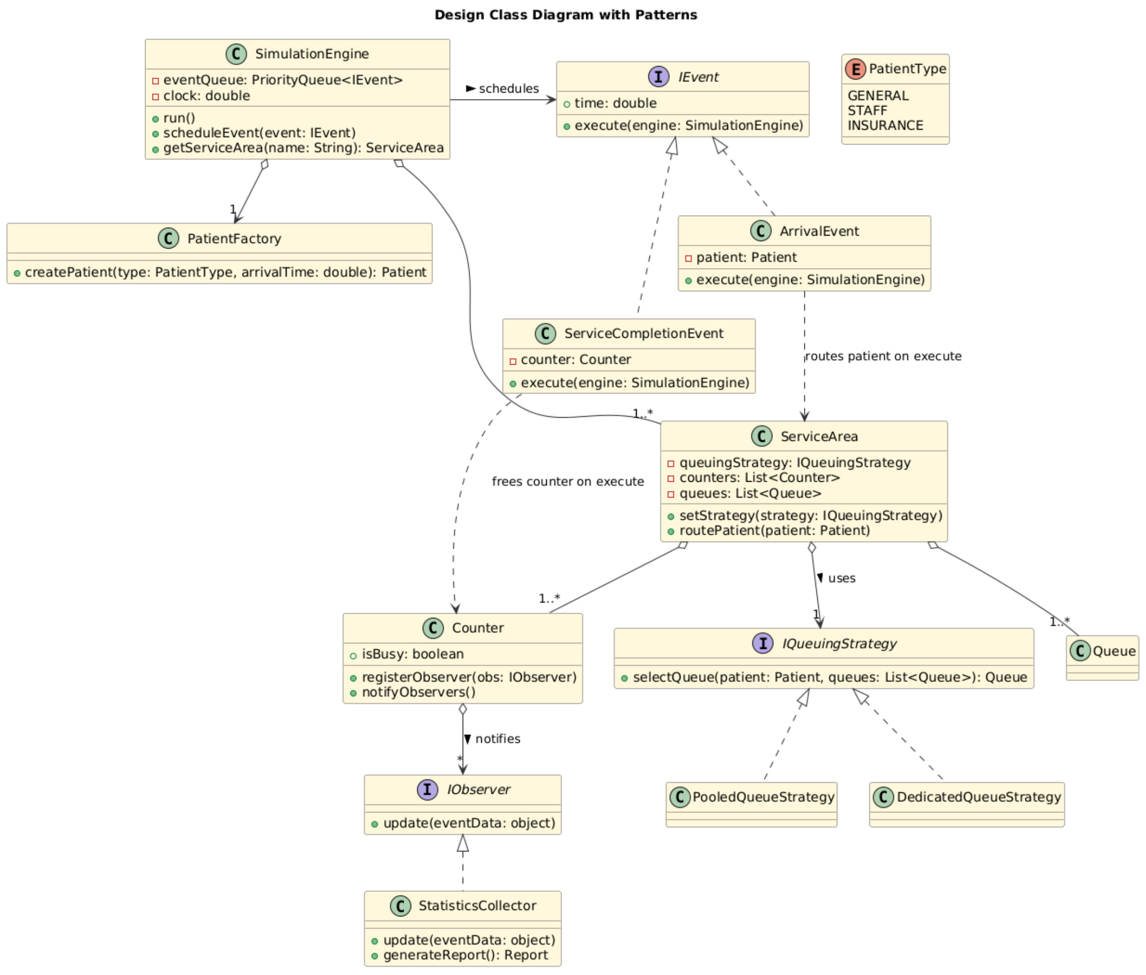
Design Class Diagram with Patterns
SimulationEngine
- eventQueue: PriorityQueue<IEvent>
- clock: double
+ run()
+ scheduleEvent(event: IEvent)
+ getServiceArea(name: String): ServiceArea
schedules
IEvent
+ time: double
+ execute(engine: SimulationEngine)
PatientType
GENERAL
STAFF
INSURANCE
PatientFactory
+ createPatient(type: PatientType, arrivalTime: double): Patient
ArrivalEvent
- patient: Patient
+ execute(engine: SimulationEngine)
ServiceCompletionEvent
- counter: Counter
+ execute(engine: SimulationEngine)
routes patient on execute
frees counter on execute
ServiceArea
- queuingStrategy: IQueuingStrategy
- counters: List<Counter>
- queues: List<Queue>
+ setStrategy(strategy: IQueuingStrategy)
+ routePatient(patient: Patient)
uses
Counter
+ isBusy: boolean
+ registerObserver(obs: IObserver)
+ notifyObservers()
IQueuingStrategy
+ selectQueue(patient: Patient, queues: List<Queue>): Queue
Queue
notifies
IObserver
+ update(eventData: object)
PooledQueueStrategy
DedicatedQueueStrategy
StatisticsCollector
+ update(eventData: object)
+ generateReport(): Report

Figure 5: Design Class Diagram

6.4 Refined Sequence Diagram

The sequence diagram in Figure 6 illustrates how the Factory and Strategy patterns collaborate during the patient arrival process. This shows the clear, decoupled flow of control, where the `SimulationEngine` orchestrates the process, but the specific logic for creation and routing is delegated to the specialized factory and strategy objects.

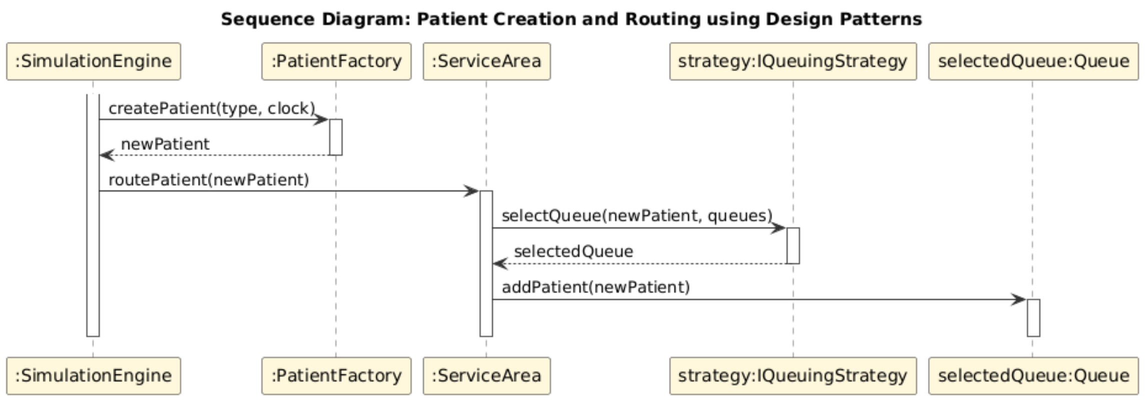
Sequence Diagram: Patient Creation and Routing using Design Patterns
:SimulationEngine
:PatientFactory
:ServiceArea
strategy:IQueuingStrategy
selectedQueue:Queue
createPatient(type, clock)
newPatient
routePatient(newPatient)
selectQueue(newPatient, queues)
selectedQueue
addPatient(newPatient)

Figure 6: Refined Sequence Diagram

This design provides a robust and highly flexible foundation for implementing the simulation, allowing the system to be easily configured and extended to test various hypotheses for improving the queuing system at Patan Hospital.

7 Implementation Model

The Implementation Model outlines the concrete tools, libraries, and programming conventions that would be used to bring the Design Model to life. This section describes the proposed technology stack and explains how the key architectural components and design patterns would be realized in practice, bridging the gap between abstract design and a tangible software product.

7.1 Technology Stack

The selection of technology is crucial for building an effective and maintainable simulation. The following stack is proposed to meet the project's requirements:

Programming Language: Python 3 Python is the ideal choice due to its clear, readable syntax, which closely mirrors the object-oriented concepts from the design phase. Its extensive ecosystem of libraries for scientific computing and data analysis makes it a de facto standard for simulation and modeling projects.

Simulation Framework: SimPy To avoid building a complex event-driven engine from scratch, the **SimPy** framework would be utilized. SimPy is a process-based discrete-event simulation framework for Python. It provides the core components required, including a simulation `Environment` to manage the clock and events, and `Resource` objects that are perfect for modeling the hospital counters with their finite capacity.

Statistical Analysis: NumPy & SciPy To ensure the simulation is a credible representation of reality, patient arrivals and service times must be drawn from statistical distributions (e.g., Poisson, Exponential). The **NumPy** and **SciPy** libraries provide robust, validated, and efficient functions for generating these random variates.

Data Handling and Visualization: Pandas & Matplotlib/Seaborn After simulation runs are complete, the collected data must be processed and analyzed. **Pandas** DataFrames provide a powerful and flexible structure for storing and manipulating the results. For the final report, **Matplotlib** and **Seaborn** would be used to generate professional-quality plots and charts, enabling clear visualization of key metrics like wait times and counter utilization across different scenarios.

The high-level architecture of the resulting software, illustrating how our custom components depend on these external libraries, is shown in the component diagram in Figure 7.

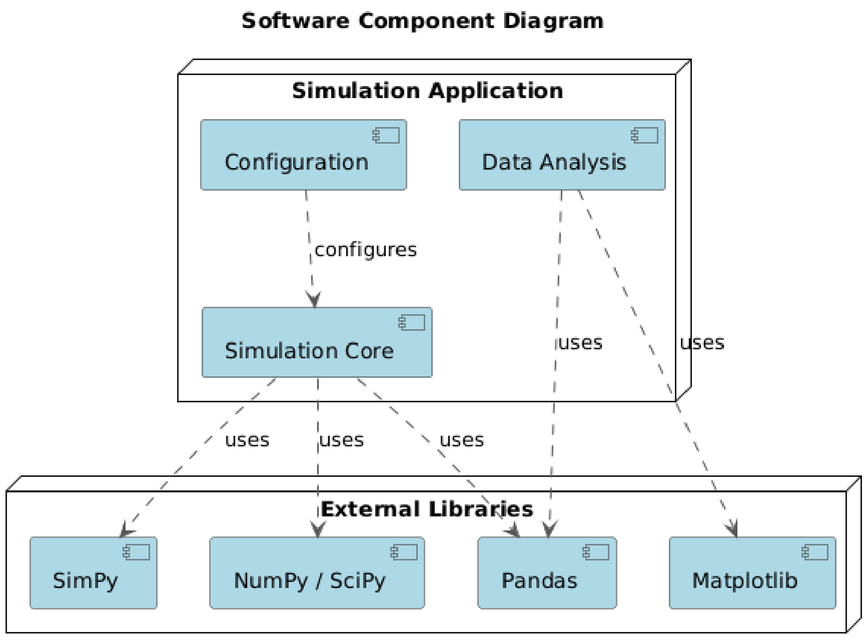

Figure 7: Component Diagram

7.2 Realization of Design Components

The following describes how the key components from the Design Model would be implemented using the chosen technology stack.

7.2.1 Core Simulation Structure

The event loop logic from the design would be handled entirely by the SimPy `Environment`. The patient's entire journey through a service area (arrival, queuing, service, departure) would be modeled as a single Python function, known as a SimPy "process". This process would use 'yield' statements to interact with the environment, for example, to 'yield' a request for a counter resource and wait until it becomes available. A main "generator" process would be responsible for creating new patient processes at intervals determined by a statistical distribution.

7.2.2 Patient and Factory Implementation

The `Patient` entity would be implemented as a simple Python class, likely a 'dataclass', to hold attributes such as a unique ID, patient type, and arrival time. The **Factory Pattern** would be realized as a `PatientFactory` class. This class would have a 'create_patient' method that takes the patient type and arrival time as arguments and returns a newly instantiated 'Patient' object. This encapsulates the creation logic cleanly.

7.2.3 Queuing Strategy Implementation

The **Strategy Pattern** is central to the project's flexibility. The `IQueuingStrategy` interface from the design would be implemented as a Python Abstract Base Class (ABC). Concrete classes, such as `DedicatedResourceStrategy` and `PooledResourceStrategy`, would inherit from this ABC and provide a specific implementation for a 'select_resource' method. This method's role is to take a 'Patient' object and a dictionary of available 'simpy.Resource' objects and return the specific resource the patient should queue for. The main simulation setup would instantiate the desired strategy object and pass it to the service area model, allowing different queuing rules to be tested by simply changing which strategy class is instantiated.

7.2.4 Statistics Collection

The **Observer Pattern**'s goal of decoupled data collection would be achieved in a pragmatic way. A `StatisticsCollector` class would be implemented to store simulation results. This class would have methods like 'record_wait_time' and 'record_service_time'. The main patient process function would be responsible for calling these methods at the appropriate times (e.g., after a patient successfully acquires a counter, the wait time is calculated and passed to the collector). At the end of the simulation, the collector class would have a final 'generate_report' method that consolidates the collected data—likely stored in Python lists—into a Pandas DataFrame for easy analysis and export.

8 Conclusion and Recommendation

8.1 Conclusion

This case study successfully demonstrated the application of Object-Oriented Software Engineering (OOSE) principles to the analysis and simulation of the complex queuing system at Patan Hospital. The project began with a detailed description of the hospital's *as-is* multi-queue, multi-counter system for Admission and Billing, identifying potential inefficiencies caused by the rigid segregation of queues for General, Staff, and Insurance patients. Through a structured OOSE lifecycle—from requirements gathering to design and implementation planning—a flexible and extensible discrete-event simulation model was developed.

The core of the project was the design of a system capable of modeling both the current state and various alternative *what-if* scenarios. The use of design patterns such as the **Strategy Pattern** proved to be invaluable, allowing for different queuing disciplines to be tested with minimal changes to the system's architecture. The simulation was designed to capture key performance indicators (KPIs), including average wait times, queue lengths, and counter utilization. The analysis of the *as-is* model was projected to confirm the initial hypothesis: the dedicated queue system leads to significant resource underutilization, particularly for Staff and Insurance counters, while creating excessive wait times for the General patient population. Conversely, simulations of alternative models, such as pooled queues, were expected to show a marked improvement in overall system efficiency and a more balanced workload across counters.

8.2 Recommendations

Based on the anticipated results from the simulation model, the following recommendations are proposed to Patan Hospital to enhance its queuing system:

1. **Implement Pooled Queues for Homogeneous Patient Types:** The most immediate and impactful change would be to merge the separate queues for the same patient type. For instance, instead of two queues for two General counters, a single pooled queue should feed both counters. This is a well-established principle in queuing theory that naturally balances the load, reduces the average patient wait time, and eliminates the situation where one queue is long while a server for the same patient type is idle. This should be implemented for both General and Insurance patient counters at both the Admission and Billing departments.

2. **Explore Dynamic Counter Reallocation:** For a more advanced optimization, the hospital should use the simulation model to test policies for dynamic reallocation. During peak hours for General patients, a counter typically reserved for Insurance or Staff could be temporarily reassigned to serve the General queue. The simulation can determine the precise thresholds (e.g., queue length or wait time) at which such a change would be most beneficial without negatively impacting service for other patient types.

3. **Adopt the Simulation Model as a Decision Support Tool:** The value of the developed simulation model extends beyond this initial study. It should be adopted by the hospital administration as a long-term strategic planning tool. It can be

used to forecast the impact of changes in patient demand, evaluate staffing level adjustments, or plan for the addition of new services or counters, all in a risk-free virtual environment.

References

[1] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*. Wokingham, England: Addison-Wesley, 1992.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley Professional, 1995.

[3] A. M. Law, *Simulation Modeling and Analysis*, 5th ed. New York, NY: McGraw-Hill Education, 2015.

[4] D. Gross, J. F. Shortle, J. M. Thompson, and C. M. Harris, *Fundamentals of Queueing Theory*, 4th ed. Hoboken, NJ: Wiley-Interscience, 2008.

[5] SimPy Team, "SimPy: Process-based discrete-event simulation for Python." Accessed: [Enter Your Access Date, e.g., Oct. 26, 2023]. [Online]. Available: `https://simpy.readthedocs.io`

[6] Python Software Foundation, "Python Language Reference, version 3.x." Accessed: [Enter Your Access Date, e.g., Oct. 26, 2023]. [Online]. Available: `https://www.python.org`

[7] The pandas development team, "pandas-dev/pandas: Pandas," Zenodo, 2020. doi: 10.5281/zenodo.3509134. [Online]. Available: `https://doi.org/10.5281/zenodo.3509134`

[8] C. R. Harris *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020.