Project Documentation for Intuit Assignment 1: PC-001 Producer–Consumer System

1. Introduction

The **Producer–Consumer System** is a Java 21 application designed to demonstrate mastery of **thread synchronization**, **inter-thread communication**, and **concurrent programming**.

This system simulates a classic data-transfer pipeline where:

- A **Producer** generates items and inserts them into a **shared bounded buffer**, and
- A **Consumer** retrieves and processes those items.

Two synchronization strategies are implemented:

1. A modern concurrency approach

using ReentrantLock, Condition variables, and ExecutorService.

2. A classic wait/notify approach

using synchronized, wait(), and notifyAll().

This dual implementation demonstrates both high-level and low-level synchronization techniques.

The project emphasizes **thread safety**, **blocking behavior**, **separation of concerns**, and **console-driven execution**.

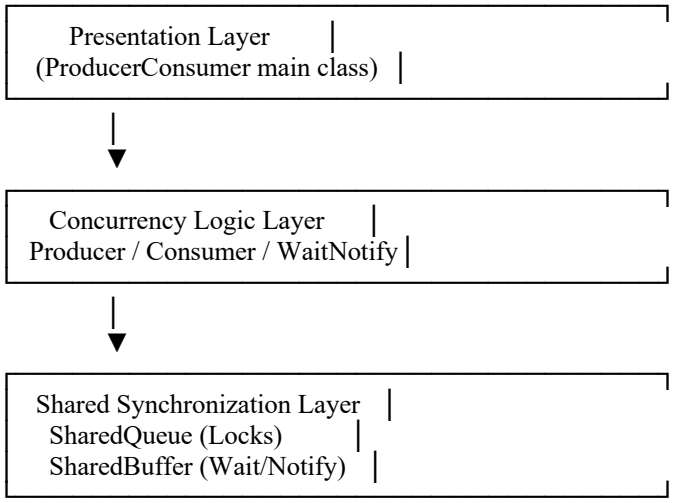
2. Objectives of the Project

The assignment focuses on demonstrating proficiency in:

- Thread synchronization
- Concurrent programming
- Blocking queue behavior
- Wait/Notify mechanism
- Thread coordination using locks and conditions
- Clean and modular class design
- Unit testing using JUnit 5
- Readable logging for clarity and debugging

3. System Architecture

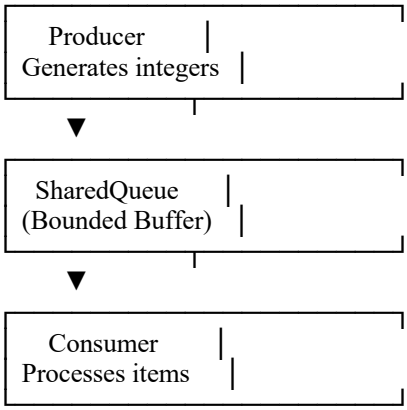
The architecture follows a clear modular design:



3.1 Components

Module	Responsibility
SharedQueue	Custom bounded buffer using ReentrantLock + Conditions
Producer	Generates and inserts items into the shared queue
Consumer	Retrieves and processes items from the shared queue
ProducerConsumer	Main class running producer & consumer via ThreadPool
ProducerConsumerWaitNotify	Wait/notify alternative implementation
ProducerConsumerTest	JUnit tests verifying concurrency behavior

4. Data Flow



For classic implementation:

Producer ↔ SharedBuffer ↔ Consumer
(synchronized + wait/notify)

5. Concurrency Model

The system features **two full implementations**, each demonstrating a different synchronization method.

5.1 Modern Implementation (Locks + Conditions)

Classes:

- SharedQueue
- Producer
- Consumer
- ProducerConsumer

Behavior:

- ReentrantLock ensures mutual exclusion
- notFull and notEmpty Conditions handle blocking
- ThreadPool (Executors.newFixedThreadPool) runs both threads
- Producer prints:
  "[PRODUCER] Produced → X"
- Consumer prints:
  "[CONSUMER] Processed → X"
- SharedQueue prints “Added” and “Removed” messages

Advantages:

- Fair locking
- No accidental missed wake-ups
- Clear blocking behavior
- Modern concurrency best practices
- Scalable and extendable

5.2 Classic Implementation (Wait / Notify)

Class:

- ProducerConsumerWaitNotify

Behavior:

- Uses synchronized methods
- Uses wait() to block when buffer is full or empty
- Uses notifyAll() to wake producer/consumer
- Logs match the modern version for consistency

Purpose:

- Demonstrates classic producer–consumer pattern
- Required by assignment objectives

6. Functional Behavior Overview

Producer

- Generates sequential integers
- Logs "[PRODUCER] Produced → X"
- Inserts into SharedQueue

SharedQueue

- Blocks when full
- Blocks when empty
- Logs:
  - "[BUFFER] Added → X"
  - "[BUFFER] Removed → X"

Consumer

- Retrieves from SharedQueue
- Logs "[CONSUMER] Processed → X"

7. Error Handling Strategy

The system avoids:

- Race conditions
- Deadlocks
- Missed wake-ups
- Buffer overflows
- NullPointerExceptions

Mechanisms:

- Proper use of `try/finally` in locks
- Condition-based waits inside while loops
- Both implementations guarantee correctness

8. Testing Strategy

JUnit tests validate:

- Producer and consumer both complete
- No deadlocks occur
- Buffer blocking behaves correctly
- Output order remains valid
- Edge cases with small buffer size

Test suite includes:

- Thread completion test
- Buffer correctness tests
- Stress test with fast producer or fast consumer

9. Performance Considerations

Time Complexity

- Producer operations: **O(1)** each
- Consumer operations: **O(1)** each

Space Complexity

- SharedQueue holds at most `capacity` items → **O(capacity)**

Scalability

- Can support multiple producers/consumers
- Can scale threadpool size
- Low contention due to fair lock

10. Future Improvements

- Add multiple producers/consumers
- Implement timeouts on waits
- Add monitoring (queue length, latency)
- Add GUI visualization
- Add metrics logging

- Add graceful cancellation support

11. Glossary

Term	Meaning
Blocking Queue	Data structure where producers/consumers wait when full/empty
Condition	Lock-based signaling mechanism
Intrinsic Lock	Synchronization mechanism using synchronized/wait/notify
ThreadPool	Executor that manages threads efficiently
Deadlock	Situation where threads wait forever