Modularity: The code is divided into separate modules (main.py and message_queue.py), each responsible for a specific aspect of the system. This modular approach promotes code reusability and maintainability.

Abstraction: The MessageQueue class abstracts the implementation details of the message queue using a priority queue (PriorityQueue). This abstraction allows users to interact with the queue through high-level methods (enqueue_message, dequeue_message, peek_message, is_empty) without needing to understand its internal workings.

Encapsulation: The MessageQueue class encapsulates the underlying data structure (PriorityQueue) and its operations. By exposing only the necessary methods and hiding the internal implementation, the class ensures data integrity and prevents direct manipulation of the queue.

Concurrency Management: The use of threads (threading.Thread) and a thread pool (ThreadPoolExecutor) enables concurrent processing of messages, enhancing system responsiveness and efficiency. Synchronization mechanisms, such as locks and condition variables, ensure thread safety and prevent race conditions.

Testing: Although not explicitly shown, the code includes test cases to verify the correctness of critical functionalities, such as message enqueueing, dequeueing, and processing. Testing promotes code reliability and helps catch bugs early in the development process.

Documentation: The presence of documentation provides an overview of the system, including its purpose, components, and usage instructions. Documentation aids in understanding the codebase, facilitates collaboration among developers, and serves as a reference for future maintenance.

Overall, the implementation demonstrates adherence to good design and coding practices, including modularity, abstraction, encapsulation, concurrency management, testing, and documentation. These practices contribute to the creation of a well-structured, maintainable, and reliable system.