Requirement:

Design and implement a message queue that supports multiple producers and consumers. The queue tries to deliver messages in FIFO order and exactly once to each consumer. Whenever a message is delivered to a consumer, it becomes invisible for a specific time period (as defined by the VisibilityTimeout attribute of the Queue); if the message is not consumed/deleted by the consumer within that timeframe, then it becomes visible again for re-delivery.

This project provides an InMemory implementation of the Queue Service. Other implementations will be added gradually.

Design overview:

The design comprises of the several key components, which are described as follows:

MessageQueue (interface) / InMemoryQueue (implementation): MessageQueue provides an interface for common operations of a message queue, such as, message manipulation (push, pull delete message), getting/setting the queue specific attributes and extracting the current state of the queue (number of available and inflight messages). InMemoryQueue is a concrete implementation of the MessageQueue interface which provides an in-memory implementation of the queue so that it can be accessed by multiple producers and multiple consumers of the same JVM.

QueueAttributeValidator (interface) / QueueAttributeValidatorImpl (implementation): QueueAttributeValidator provides an interface for validating queue specific attributes, such as, validating an attribute name and its corresponding value. It also provides default value of a valid attribute and set of default attribute name/value pairs. QueueAttributeValidatorImpl is the concrete implementation of this interface. This class contains a map of <attribute_name, attribute_validator> pairs.

The validator interface is passed to the MessageQueue during instantiation. The message queue simply depends on the abstraction layer (the validator interface), not the actual implementation. Therefore, when the QueueAttributeValidatorImpl is replaced with another implementation, e.g., QueueAttributeValidatorImplV2 of the validator interface, message queue does not get affected (Dependency Inversion principle). This design choice also decouples the unit testing the message queue from the concrete validator class (a mock of the validator interface has been used while unit testing the message queue).

AttributeValidator (interface) / VisibilityTimeoutValidator (implementation): AttributeValidator provides an interface for validating the given value of a single queue attribute, such as, message size, policy, visibility timeout etc. It also provides the default value of the specific attribute. A concrete implementation of this interface is VisibilityTimeoutValidator. It checks whether the given value of the visibility timeout stays within the specific range. It also provides default value of the visibility timeout.

ScheduledTask: This class is an implementation of the TimerTask class. It periodically calls the refreshQueue method of the given message queue, provided during its instantiation. The refreshQueue method checks which messages have expired their visibility timeout period and makes those available for reprocessing. The ScheduledTask class is configured in the constructor of the message queue.

QueueService (interface) / InMemoryQueueService (implementation): The QueueService provides an interface for manipulating messages in multiple queues. It handles queue specific operations (create/delete/purge queue, set/get queue attributes), message manipulation (push/pull/delete message from a specific queue) and extracting the state of a specific queue (number of

available and inflight messages). InMemoryQueueService is a concrete implementation of this interface.

QueueFactory (interface) / InMemoryQueueFactory (implementation): While creating a new queue in the InMemoryQueueService, if we use the new keyword (such as, new InMemoryQueue()), then this class needs to be updated for every change in the implementation of the InMemoryQueue; for instance, if InMemoryQueue is replaced with another version, such as, InMemoryQueueV2, we have to change the code in the InMemoryQueueService class; thus violating the open/close principle. For this reason, I decided to inject a QueueFactory interface in the constructor of the QueueService implementation; when we need to create a new queue, we just need to call the create method of the QueueFactory abstraction and it will provide us the desired message queue. InMemoryQueueFactory is a concrete implementation of the QueueFactory interface. This design choice also decouples the unit testing of the QueueService class from the concrete implementation of the message queue (mocked the message queue during unit testing). For more information, please visit: **http://stackoverflow.com/questions/21879804/inject-mocks-for-objects-created-by-factory-classes**

MessageCreator: MessageCreator class provides a static method for creating a message object from the message body.

Testing:

This code has 53 unit tests in total, which takes about half a second to run in a 1.8 GHz Intel core i7 macbook air OSX 10.9.5. During unit testing, I tried to check whether the returned value of the method matches the expected value, side effects of the method are as expected and commands sent to other objects are received properly (performed using mocks).

For unit testing visibility timeout, just made sure that the refreshQueue() method is working properly. The timer will run periodically (if properly configured), haven't felt the need to test its periodic execution.

Also wrote a small integration test to check whether the InMemoryQueue is thread safe or not. Other tests can be written to check the thread safety of the InMemoryQueueService.