Code Rationale

Question: Why did we make the **message class a generic class** (<T>)?

Answer:

- Prevented compile-time errors as it catches invalid types at compile-time
- Allowed us to create with a single declaration a class of related types
- Example, in the portal class we can display different messages on the GUI with different types, such as String and MetaAgent.

Question: Why did we use the **LinkedBlockingQueue**?

Answer:

- It implements BlockingQueue. Therefore, it has a built-in wait state, so it waits for the queue to be none empty before retrieving an element and waits until there is a space to become available before offering an element to the queue.
- It is thread-safe all queuing methods achieve their effects atomically using internal locks and other forms of concurrency control.
- LinkedBlockingQueue operates as a first in first out basis (FIFO). Messages are added at the tail of the queue and taken off the head of the queue.
- LinkedBlockingQueues can handle more of a through rate than array queues.
- No set limit so can store many messages, good for scalability.

  An alternative was the ConcurrentBlockingQueue but kept throwing a “NoSuchElement” exception because it kept reading from the queue when nothing was there.

Question: Why did we use **offer** and **not add** when adding messages to the queue?

Answer:

- Adds an element to the tail of the queue if possible without exceeding the queues limit
- Add throws an error if it can’t add an element to the queue whilst offer won’t and will return false

Question: Why did we use a timeout in the offer method?

Answer:

- The timeout waits up to ten seconds to try and add an element to the queue, waiting for a space to become available. If not, an error message will be produced and the message added to a logger.
- It allowed for scalability. We tried changing it to one nano second and it didn't break then. If we sent the software out into industry it may need a timeout as millions of messages and agents are possible and it could break.

Question: Why did we use take and note remove?

Answer:

- Remove will throw an error whilst take waits for an element to become available.

Question: Why did we use a ConcurrentHashMap?

Answer:

- It implements ConcurrentMap which provides thread-safety and atomically guaranteed.
- The ConcurrentHashMap has many methods that have synchronized blocks so we don't have to which guarantees thread safety and validation.
- Allows for scalability with multiple threads
- ConcurrentHashMaps locks a portion of the data which is being updated while another portion is being accessed by another thread. So there is no interleveing issues
- No ConcurrentModificationException while one thread is updating and one is iterating through.

Question: Why didn't we use synchronisation?

Answer:

- Synchronization is used when multiple threads try to access the same resources and produce errorous or unforeseen data results.
- Synchronization allows only one thread to access that resource at a time.
- The ConcurrentHashMap and the LinkedBlockingQueue seem to support enough synchronization throughout the program with no known errors.
- We could change the ConcurrentHashMap and LinkedBlockingQueue into standard versions with no automatic concurrency and implement it ourselves
- We would simply synchronize on a static object. We could use it when we are adding and removing from the message queue and reading and updating the portals routing tables.

Question: Why did we use composition over inheritance?

Answer:

- Inheritance = “Is-a” relationship
- Composition = “Has-a” relationship
- An Example, a car has an engine, a car isn’t an engine. So we would create an instance of an engine in the car class instead of making the car class extend/implement engine.
- A Meta-Agent has a BlockingQueue, A Meta-Agent isn’t a BlockingQueue.
- Composition over inheritance is an OOP principle that states classes should achieve polymorphic behaviour and code reuse by composion (Instances in a class rather than extension).
- It’s more flexable, has a relationships
- No complex structure with many classes extending/implementing others. For example, Portal extends NodeMonitor which extends MetaAgent which Extends LinkedBlockingQueue. This is complex and can be hard to change in future development.