Jack Diaz
111499298
Design Document

- Imperative Graph
  - Four fields, one that holds the nodes, one that holds the edges, and two locks. The lists holding the nodes and edges are both immutable and the locks are both final. The locks are labeled with the order they should be synchronized on in order to avoid deadlock.
  - Two constructors, one that sets the nodes and edges to null and the other that sets them based on the parameters. The first is the public constructor and is used by the user to create a new ImperativeGraph and the second constructor is private and is used internally to create copies of this ImperativeGraph.
  - addNode takes in a node and looks to add it to the graph. This will modify the state, but only the node list; the edge list will not be affected. As such we only need to acquire the node lock. This means that things affecting only the edges can happen at the same time. After acquiring the lock, it adds the node if it's not already a member.
  - addEdge adds an edge to the graph. It should also add the nodes on either side of the edge if they're not already there. This means it could be modifying both edges and nodes, so we need to acquire both locks. We cover the whole method with both locks because if you tried to copy the graph after adding the edge but before adding the nodes you would have copied a bad state that never should have existed.
  - removeNode also involves both pieces of state so we acquire both locks. Then it removes all edges associated with the parameter node. After that it removes the node if it's there and returns true. If it's not there then it returns false. We cover the whole method with both locks because if you tried to copy the graph after removing the edges but before removing the node you would have copied a bad state that never should have existed.
  - removeEdge method removes an edge from the graph and does not affect the nodes so we only acquire the edgeLock. This means that things affecting only the nodes can happen at the same time. If the edge is in the graph it removes it and returns true; otherwise false.
  - clear method requires both locks because it affects both pieces of state by setting them to null. It would be bad if while removing an edge, after checking if the given edge is a member, and finding it is, that we make edges null and try to remove from it. That situation would cause an exception. There are other situations in which clearing at an inopportune time could be fatal to thread safety.
  - Copy makes a copy of the graph by acquiring both locks and calling the constructor. This will ensure the state isn't modified after writing one but before writing the other.
  - iterNodes iterates over the nodes and applies a function to each, this only uses nodes so it only needs the one lock
  - iterEdges iterates over the edges and applies a function to each, this only uses edges so it only needs the one lock
  - iterSuccessors iterates over the successor nodes and applies a function to each, this only uses nodes so it only needs the one lock
  - iterPredecessors iterates over the predecessor nodes and applies a function to each, this only uses nodes so it only needs the one lock
  - toPersistentGraph returns a persistent graph. It needs both locks because it iterates over both nodes and edges.

    - equals uses both locks because it iterates over both edges and nodes. It makes a copy of the other graph so it doesn't care if that one changes, it just compares against the older version of the other graph.
    - the locks in this class ensure that no two threads can affect the state at the same time, but they also allow operations that can go at the same time to be able to do so.
- PersistentGraph
    - This class's methods do the same things as the ImperativeGraph's methods except they return a new graph each time, which means the state of the PersistentGraph never changes. This means it's thread safe inherently because the state cannot be changed.
- Why one over the other?
    - The advantage to using a PersistentGraph over an ImperativeGraph is that it is guaranteed to be thread safe because it is immutable. They are also advantageous because they are easy to test. They can also be great for backing other objects that need to be thread safe, like a CopyOnWriteArrayList, because they create new graphs whenever there is a change.
    - The advantage to using an ImperativeGraph over a PersistentGraph is that it is easier to think and reason about because it changes in place. It's easier for some to look at a piece of code using a mutable object and know exactly what it does. Another advantage is that it uses less memory because you're not constantly creating new objects.
    - I think in most cases a user would prefer to use a PersistentGraph because I don't have much difficulty reasoning about immutable objects and they tend to solve a lot of problems for the user.