Diffchecker

- 116 Removals + 54 Additions

7 java

```
package server.main;


import java.util.NoSuchElementException;
import java.util.Queue;
import java.util.concurrent.ConcurrentLinkedQueue;

import server.services.protocol.InputMessageQueue;

/**
 * A threadsafe global queue of all the messages received by all clients
 * connected to the server. The server is constantly processing this queue.
 *
 * The queue uses Round-Robin scheduling; that is, messages of higher priority
 * are always dequeued before messages of lower priority, and it follows a FIFO
 * scheme for messages with the same priority.
 *
 * @author Adrian Petrescu
 */
public class GlobalInputMessageQueue {

        private static GlobalInputMessageQueue self;
        /**
         * Singleton accessor to the GlobalInputMessageQueue. If the queue has not
         * yet been accessed, it will be created.
         *
         * @return A reference to the GlobalInputMessageQueue.
         */
        public static GlobalInputMessageQueue getGlobalInputMessageQueue() {
                if (self == null) {
                        self = new GlobalInputMessageQueue();
                }
                return self;
        }

        protected Queue<InputMessageQueue> lowPriorityInputMessageQueue;
        protected Queue<InputMessageQueue> medPriorityInputMessageQueue;
        protected Queue<InputMessageQueue> highPriorityInputMessageQueue;

        /**
```

```
package server.main;

import java.util.EnumMap;
import java.util.NoSuchElementException;
import java.util.Queue;
import java.util.concurrent.ConcurrentLinkedQueue;

import server.services.protocol.InputMessageQueue;

/**
 * A threadsafe global queue of all the messages received by all clients
 * connected to the server. The server is constantly processing this queue.
 *
 * The queue uses Round-Robin scheduling; that is, messages of higher priority
 * are always dequeued before messages of lower priority, and it follows a FIFO
 * scheme for messages with the same priority.
 *
 * @author John Smith :)
 */
public class GlobalInputMessageQueue {

        private static GlobalInputMessageQueue instance;
        public static GlobalInputMessageQueue getInstance() {
                if (instance == null) {
                        instance = new GlobalInputMessageQueue();
                }
                return instance;
        }

        protected EnumMap<QueuePriority, ConcurrentLinkedQueue<InputMessageQueue>> queues;

```

```
         * Creates a new instance of GlobalInputMessageQueue.
         */
        protected GlobalInputMessageQueue() {
                lowPriorityInputMessageQueue = new ConcurrentLinkedQueue<InputMessageQueue>();
                medPriorityInputMessageQueue = new ConcurrentLinkedQueue<InputMessageQueue>();
                highPriorityInputMessageQueue = new ConcurrentLinkedQueue<InputMessageQueue>();
        }

        /**
         * Add a message to the global queue.
         *
         * @param message The message to be queued up.
         */
        @SuppressWarnings("deprecation")
        public synchronized void enqueue(InputMessageQueue message) {
                boolean unlockListener = this.isEmpty();
                switch (message.getPriority()) {
                        case 0:
                                lowPriorityInputMessageQueue.add(message);
                                break;
                        case 1:
                                medPriorityInputMessageQueue.add(message);
                                break;
                        case 2:
                                medPriorityInputMessageQueue.add(message);
                                break;
                }
                if (unlockListener) {
                        /* TODO: According to my experiments, main is always the first thread in
                         * the main thread group, but I'm not sure if this is always
                         * necessarily the case. Do more research here.
                         *
                         * Assuming that it is saves time, but may cause deadlock if we're
                         * wrong.
                         *
                         * If you have definite knowledge about the likelihood of main not
                         * being the first thread, please file a ticket!
                         */
                        Thread[] threads = new Thread[1];
                        Thread.currentThread().getThreadGroup().enumerate(threads);

                        if (threads[0].getName().equals("main")) {
                                threads[0].resume();
                        }
                }
```

```
        protected GlobalInputMessageQueue() {
        queues = new EnumMap<>(){{
            put(QueuePriority.LOW, new ConcurrentLinkedQueue<InputMessageQueue>());
            put(QueuePriority.MED, new ConcurrentLinkedQueue<InputMessageQueue>());
            put(QueuePriority.HIGH, new ConcurrentLinkedQueue<InputMessageQueue>());
        }};
        }

        @SuppressWarnings("deprecation")
        public synchronized void enqueue(InputMessageQueue message) {
                switch (message.getPriority()) {
                        case 0:
                                addToQueue(Priority.LOW, message);
                                break;
                        case 1:
                                addToQueue(Priorirty.MED, message);
                                break;
                        case 2:
                                addToQueue(Priorirty.HIGH, message);
                                break;
                }
                if (this.isEmpty()) {
                        Thread[] threads = new Thread[1];
                        Thread.currentThread().getThreadGroup().enumerate(threads);

                        if (threads[0].getName().equals("main")) {
                                threads[0].resume();
                        }
                }
```

```java
    }

    /**
     * @return The oldest message of the highest priority available, or NULL if the
     * queue is empty.
     */
    public synchronized InputMessageQueue dequeue() {
        try {
            return highPriorityInputMessageQueue.remove();
        } catch (NoSuchElementException noHigh) {
            try {
                return medPriorityInputMessageQueue.remove();
            } catch (NoSuchElementException noMed) {
                try {
                    return lowPriorityInputMessageQueue.remove();
                } catch(NoSuchElementException noLow) {}
            }
        }
        return null;
    }

    /**
     * Return the total number of input message queues still queued up
     * in the global buffer, of all priorities.
     *
     * @return The total number of queued InputMessageStacks.
     */
    public int getSize() {
        return lowPriorityInputMessageQueue.size()
            + medPriorityInputMessageQueue.size()
            + highPriorityInputMessageQueue.size();
    }

    /**
     * Return the total number of input message queues still in the queue
     * with a given priority.
     *

```

```java
    }


    private synchronized void addToQueue(QueuePriority priority, InputMessageQueue message){
        getQueue(priority).add(message);
    }

    private synchronized ConcurrentLinkedQueue<InputMessageQueue> getQueue(QueuePriority priority) {
        return queues.get(priority);
    }

    public synchronized InputMessageQueue dequeue() {

        try {
            return getQueue(QueuePriority.HIGH).remove();
        } catch (NoSuchElementException noHigh) {
            try {
                return getQueue(QueuePriority.MED).remove();
            } catch (NoSuchElementException noMed) {
                try {
                    return getQueue(QueuePriority.LOW).remove();
                } catch(NoSuchElementException noLow) {}
            }
        }
        return null;
    }

    public int getSize() {
        return getQueue(QueuePriority.LOW).size()
            + getQueue(QueuePriority.MED).size()
            + getQueue(QueuePriority.HIGH).size();
    }

    public int getSize(QueuePriority priority) {
        return getQueue(priority).size();
```

```
         * @param priority The priority level of counted messages.
         * @return The total number of queued InputMessageStacks of the given
         * priority.
         */
        public int getSize(int priority) {
                switch (priority) {
                case 0:
                        return lowPriorityInputMessageQueue.size();
                case 1:
                        return medPriorityInputMessageQueue.size();
                case 2:
                        return highPriorityInputMessageQueue.size();
                default:
                        return 0;
                }
        }

        /**
         * Returns <code>true</code> if the queue contains no elements of any
         * priority.
         *
         * @return <code>true</code> if the queue is completely empty.
         */
        public synchronized boolean isEmpty() {
                return
lowPriorityInputMessageQueue.isEmpty()
                                        &&
medPriorityInputMessageQueue.isEmpty()
                                        &&
highPriorityInputMessageQueue.isEmpty();
        }

        /**
         * Returns <code>true</code> if the queue contains no elements of the
         * given priority.
         *
         * @param priority The priority level to check.
         * @return <code>true</code> if the queue is empty of messages of the given
         * priority.
         */
        public boolean isEmpty(int priority) {
                switch (priority) {
                case 0:
                        return lowPriorityInputMessageQueue.isEmpty();
                case 1:
                        return medPriorityInputMessageQueue.isEmpty();
                case 2:
                        return highPriorityInputMessageQueue.isEmpty();
                default:
```

```
        }
        public synchronized boolean isEmpty() {
                return getQueue(QueuePriority.LOW).isEmpty()
                                        &&
getQueue(QueuePriority.MED).isEmpty()
                                        &&
getQueue(QueuePriority.HIGH).isEmpty();
        }

        public boolean isEmpty(QueuePriority priority) {
                return getQueue(priority).isEmpty();
```

```
                        return true;
                }
        }
        
}
```

```
        }
    
    enum QueuePriority {
        LOW,
        MED,
        HIGH
    }
}
```