Concurrent Inventory System

Before describing the system itself, some assumptions about the context of this inventory must be made. First of all, the store that utilizes this system has a set inventory that does not often vary size or add new items. Second, the SKU code and item name are known previously as parameters for each request made. And finally, as soon as the item is purchased, the delivery of the item by the quadcopter occurs shortly after.

To increase efficiency, concurrency is used between different requests in the inventory management system. The following concurrent requests can occur in the inventory system: *insert item*, *insert more items*, *remove item*, *purchase item*, and *return (terminate) item*. To support these requests in an efficient manner, a concurrent data structure must be chosen. Since the inventory was previously assumed to not vary in size or change items often, the use of a ConcurrentHashMap is ideal due to its access time. Although a concurrent B-tree may seem better suited for operations and data that seem similar to a database, we don't need to worry about growing our inventory very often or requesting a range of values. Also, Java does not support any type of concurrent B-tree. However, the ConcurrentHashMap provides a useful means for concurrency. The ConcurrentHashMap is split up into segments where a lock can be obtained by a different thread on each segment. This allows for concurrent updates on different segments of the HashMap rather than one thread obtaining a lock on the whole HashMap. Additionally, reads do not require locks, and dirty data can be returned. To deliver the orders placed to the quadcopter, the orders are put into a concurrent queue.

Initially inserting into the inventory should happen well before any other activity is allowed to occur. Any inserts occurring after the inventory has been initialized, however, can happen at any time. In the context of a store, if an item is being inserted and does not exist yet, then there are no other concurrent operations occurring on this item that could leave our customer unhappy. Also, by using the insert operation to insert more items of an already existing item, we don't risk the customer seeing more than there already is. Since the insert function requires a comparison and a ConcurrentHashMap put call, inserting is not atomic. This means that a thread requesting an availability or a purchase could see fewer items than what is currently being inserted. Even if this turns away a customer from buying a large number of the product, it does not put the customer in a situation where he or she has purchased more of an item than is available. The only request one should worry about that may conflict with insert is remove item. If a thread removes an item in the middle of an insert more operation, this will cause a NullPointerException. However, this would never happen. If it did, then the person in charge of maintaining inventory intended it. Removing an item should not occur while an item is being purchased or returned. Fortunately, this is handled by the purchase and return item function. Removing the item simply removes the item pertaining to a particular SKU from the inventory.

Purchasing an item requires a check for whether the item is in stock and a reduction of the quantity of the item in the inventory. An order is created (containing the order number, SKU, put and quantity) and is inserted into the concurrent queue. The most worrisome concurrency issue is when there exists less stock than what the current purchase item request sees. In other words, the customer has purchased more than what is in stock. To avoid this, I implemented a lock on the item being purchased so that no other purchase or request could access this information until the

purchase completed. This lock also blocks the available request from reading information from this item when it is being purchased. Following a purchase, the user might decide to terminate their order and return their item. When this occurs, the queue is traversed to find the item ordered based on order number and the HashMap entry of the item's quantity is updated. Even though queue traversal is slow, the time for the quadcopter to return the item at the store will take longer, so there is no rush to update the inventory. However, if a remove item finishes before the return item request, then the customer will be frustrated by the nonexistent item they just ordered less than an hour ago. Fortunately, this will rarely happen.

At last, we must consider the availability request. Unfortunately, the get method of ConcurrentHashMap does not require a lock on a segment. Even if this may cause customers to be unreliably informed of what the stock actually is due to dirty reads, this unreliable information will not get us in trouble. With the lock used in the purchase item request, it is not possible for more items to be available than what actually exists. And that sums up the inventory system.

The following design of the inventory system was implemented in the accompanying file *Inventory.java***.**