KitchIntel Control System

A Summary of the Discussion at the Planning Meeting for Sprint 4

Overview of the Sprint

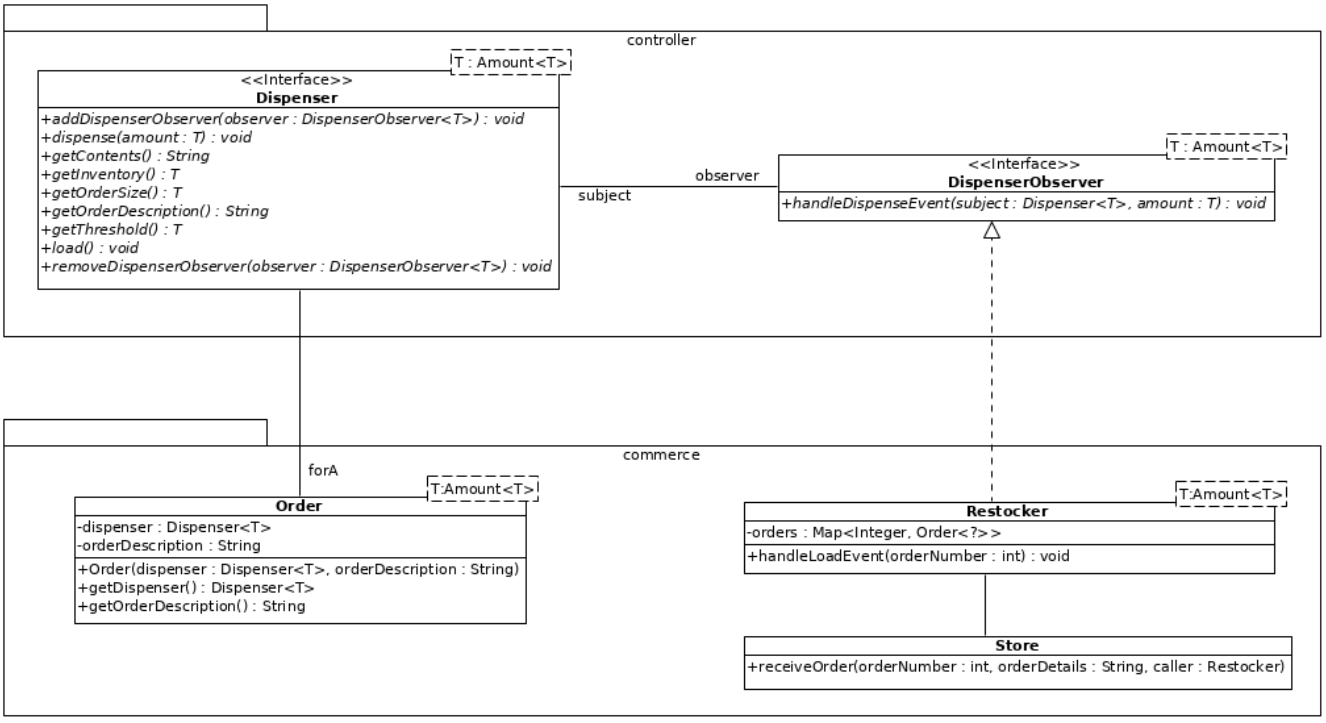
KitchIntel is now starting to work with various stores so that the system can place orders when the inventory of a dispenser falls below the threshold, the store can fulfill the order (some time later), and the system can load the delivered items into containers. They want the team to start working on the software components that will be needed to make this work. Other members of the team are working on the controllers for the devices that will actually receive the physical deliveries.

Overview of My Commitments

I have agreed to work on encapsulating an order (i.e., an `Order` class), writing a `DispenserObserver` that can place and receive orders, and writing a simple simulator of a `Store` that can receive and fulfill orders.

The Design

The team agreed to the following design.



Details

`Restocker` Class

A `Restocker` object can observe multiple `Dispenser` objects (as long as they are typed appropriately). Since each `Dispenser` object might running in it's own thread of execution (indeed, on it's own JVM on its own CPU) in the future, the `Restocker` class must be thread-safe.

I may need to modify my concrete classes that implement the `Dispenser` interface in a way that allows each to run in its own thread of execution so that I can test this, or I may need to create multiple threads that invoke the methods belonging to these objects. This requires some thought.

Note that the `Restocker` contains a collection of `Order` objects that are "in process" because its `handleLoadEvent()` method is passed an order number not an `Order` object.

When its `handleLoadEvent()` method is called it must call the `load()` method of the appropriate `Dispenser`.

`Order` Class

An Order object is associated with a particular `Container`. An `Order` object will be created by a `Restocker` object when the inventory falls below the threshold.

`Store` Class

The `Store` class is a simple simulator of what will later be a much more complicated subsystem. At the moment, when it's `receiveOrder()` method is invoked it must:

1. Do something with the `orderNumber` and `orderDescription` (as required);
2. Wait a random amount of time; and
3. "Call back" to the calling `Restocker` object's `handleLoadEvent()` method (passing it the `orderNumber`).

Of course, a single `Store` object must be able to receive orders from multiple `Restocker` objects, so it must be thread-safe. Again, I will need to think about how I invoke the methods in the `Restocker` objects in different threads of execution so that I can test the `Store` class. I will also need to think about how to manage/organize the different `Restocker` objects.

It was pointed out that the signature of the `receiveOrder()` method is likely to change in the future (i.e., when the system is distributed), but I probably don't have to worry about that now.