Jennifer Liu
6.170 Nomination
11/30/12

# Project 2: Shopping Cart
# Design Analysis

## Overview

**Key design challenges**

- *Separating shopkeeper and shopper interfaces*
- *Dealing with price changes and changes in other product information*
- *Avoiding concurrency issues in the event of low-stock items*

## Details

**Data representation**

Products and Items are distinct classes—Products represents elements in the catalog that are added and managed by the shopkeeper, while Items represent instances of Products that have been added to the cart or purchased by a shopper. There is one Item object for each Product that is added to the cart by the user (if multiple instances of the same Product are added, the Item's "qty" field gets incremented). So each Item corresponds to exactly one Product and can be placed into one Cart.

Even though the Product already has a "price" field, the Item also has a "price" field in case the shopkeeper decides to change the price. This way, orders that have already been placed will retain their original prices in the shopkeeper's records, while new orders will have the new prices. If an Item is located in a cart during a price change, the shopper will be notified when s/he views the cart.

Each Order is associated with exactly one Cart. Once a Cart is confirmed at checkout, an Order is created with the specified Address. At this point, the Cart's "status" field becomes "ordered" and the "price" field of all the Items belonging to the Cart is locked.

The Cart object has two other possible statuses: "current" and "saved." The "current" Cart refers to the one that appears in the upper corner, where Items get added to while the user is browsing

the site. “Saved” carts refer to Carts that the user has put aside and may re-claim as their current Cart at any time.

Regarding user authentication, Admin is a subclass of User with special privileges. There can be multiple Admin accounts that access the admin portion of the site, but these must be manually entered into the database so that no random person may arbitrarily create an Admin account. Admin is a subclass of User because it shares the same attributes of email and password, but Rails’ Single Table Inheritance feature allows us to recognize it as a separate type, facilitating shopkeeper authentication without having to create a separate table with only a few rows in it.

**Key design decisions**

*Separating shopkeeper and shopper interfaces*

The main site is for the shopkeeper, while the admin has access to another part of the site with a path that begins with “/admin.” This is to keep the two interfaces separate. The admin logs in with the same authentication system, but is re-directed to the admin interface upon a successful login.

*Dealing with price changes and changes in other product information*

Product and Item are separate classes, parallel to the way shopkeeper and shopper interfaces are separate. Each class has its own “price” field to maintain a distinction between current, dynamically updated information and historical logged information. So a shopkeeper may change the price of a product without affecting the records of instances of the product that have already been bought. If the shopkeeper decides to change the price of a product while an instance of the product is located in a user's cart (but has not been purchased yet), the user will be notified of the price change the next time s/he views the cart, and the cart will display the updated price. With this design, the shopkeeper has flexibility in modifying prices with fair warning to the shoppers.

*Avoiding concurrency issues in the event of low-stock items*

In order to avoid concurrency issues during purchasing, when a user adds an item to his cart, the Product’s “qty_in_stock” field is decremented. This is essentially a mechanism for the user to lay claim to the item. The claim lasts only as long as the cookie holding his cart information lasts, however, which is 48 hours, unless the user decides to save his cart, in which case it lasts for one week. After the period of the claim is over (or if the user removes the item from his cart), the Item instance is deleted and the “qty_in_stock” field is updated accordingly.

Given more time for this project, I would implement an email notification system so that users would be notified right before their claims expire. These reminders telling them to complete their

purchases would hopefully prevent surprises and/or disappointment from occurring when saved carts are deleted.

Another possibility for keeping track of product stock is to have a separate table to keep track of claims on items. We could call this table Locks, and have a separate column in the Products table called “qty_available” in addition to “qty_in_stock.” Every time a user places an item in her cart, a Lock would be created on the item, and the “qty_available” column would be decremented accordingly. This would create a distinction in the Products table between an item being placed in the cart versus an item being purchased. A user could only place an item in her cart if the “qty_available” was greater than zero; that way, when an item is inside the cart, the user would be guaranteed to be able to buy it for a certain amount of time (until the lock expires). This would be helpful for the shopkeeper, who could see exactly how much of the product is left in stock versus how much of the product has been placed in carts. It is also beneficial for the shopper, who does not have to rush to check out if a product is low in stock. However, it would also be possible for a sneaky shopper to hoard entire stocks of products by placing them in his cart, and repeating this every time his lock expires. This would be problematic for the shopkeeper, as well as the rest of the shoppers.

Yet another option for managing stock is to not create any type of hold when a user adds an item to her cart. So the product’s “qty_in_stock” column would only be decremented when the user confirms the purchase, and there would be no way to reserve items. This would create race conditions for checkout, and could lead to user frustration and disappointment if, for example, a user entered in all her credit card information and confirmed the purchase just to find out some other user just purchased the last item in stock. Although this design may lead to more sales for the shopkeeper, it was rejected on the basis of potential for user frustration.

*Other rejected designs*

One object model design considered was making Order a subclass of Cart, since they both share a one-to-many association with Items and a one-to-one association with a User. This was rejected on the principle that an Order is not fundamentally a type of Cart, and in fact should have its own fields for time created and address to be shipped to. It therefore makes more sense for an Order to be a separate class associated with exactly one Cart, and by means of this association have access to the Items within the Cart.

## Evaluation

**Design Critique**

*Summary assessment from user's perspective*

The site is pretty straightforward to use, and the page flow is relatively smooth. The informational messages at the top of each page are helpful in guiding the user along and confirming their actions. However, currently, no error messages are displayed if the user enters invalid login/ registration information. This is confusing and should be implemented. Email validation would also be helpful.

An additional useful feature would be the ability to select a quantity before adding an item to the cart, to save the user an extra step in case s/he wishes to purchase more than one.

On the shopkeeper front, it would be nice if the shopkeeper were redirected to the admin page upon login instead of the shopper home page. It would also be nice for the shopkeeper to have some way of keeping track of orders, whether by having a "status" column where they can mark the order as "processed" or "shipped," for example, or even just the option to delete an order.

*Summary assessment from developer's perspective*

The code, for the most part, is well-structured. The model-updating methods are nicely separated from the controller methods. Perhaps it might have been wise to have a controller specific for admin functions; however, all admin functions have a before_filter applied to them which authenticate the admin status of the user before allowing them to view the page.

*Most and least successful design decisions*

Most successful – creating the item model as a snapshot of the products class, with a separate price column that is subject to change. Creating the order model as a class with a one-to-one relationship with a cart, so that most of the cart's attributes can be accessed through the association.

Least successful - limited shopkeeper capabilities in dealing with orders

*Analysis of design faults in terms of design principles*

The reuse of code in the layouts for the admin and shopper views is not very DRY. Ideally there

would be a shopper.html.erb which, along with admin.html.erb, would inherit from application.html.erb, which would contain all the common code.

*Priorities for improvement*

Fix redirects upon login (redirect to last page if visited, redirect to admin page if admin). Implement cookie expiration so that items in carts cannot be reserved forever, since quantity in stock is decremented when items are added to the cart. Add form validation and error checking for login/registration form.

**UPDATE:** Upon admin login, page now redirects to admin orders page. Form validation added for login/registration form.