Pizza Shop Point of Sale Design Project

CSE 503

Michael Gasser, Marius Popa, Xenia Hertzenberg

Introduction

Our design revolves around both providing the necessary functionality, and being extensible in a way that does not limit the fictitious pizza store's growth. It relies on blackbox external systems for charging customer cards, and for estimating a delivery time (without taking into account whether the pizza is ready or what the order queue looks like).

Our design considers as its primary actors a store employee and a store manager. Most flows only involve the employee, but the manager may be required for some actions that require authorization (for example voiding an order), for replenishing the inventory, and for setting prices and sales.

The design is relatively straightforward. There are three main layers, the GUI (both for manager and employee), a business logic layer, and a persistence layer. It is intended that all of this might be run on the same POS register, or easily deployed with the durable storage shared across multiple registers. The design is a layered system and uses the MVC pattern at the GUI layer, the builder and the adapter patterns in the business logic layer

Extensibility and InventoryItem

There are two important forms of extensibility in the system. Inventory control, and specials. These two points of extensibility intend to allow the store to rapidly create new sales, and new highlighted pizzas, combos, or otherwise. Used in its fullness the system could be a complete inventory management system, alerting the manager when inventory is low and needs to be replenished.

The inventory is composed in software of objects of class InventoryItem. These objects have a name, unit price, and count. For example, this may look like {Pepperoni, $.50, 150}. This represents something named Pepperoni, which costs $.5 per unit, of which there are 150 units in inventory. Unit here is left to the discretion of the manager, and with pepperoni, it would likely be a small pizza size serving, but probably not a single pepperoni slice. Note that we are abusing the word "unit" here. The unit count is actually a double, to allow things like a medium pepperoni pizza being composed of 1.5 units of pepperoni.

Of course, selling units of pepperoni is a little uninteresting, so InventoryItems are composed into objects called Items. These are just lists of InventoryItems, with a name, size, and possible price override. An example of this might look like {Hawaiian Pizza, $10.99, Medium, <Thick Crust, Red Sauce, Pineapples, Ham>}. Each of the items in the list are InventoryItem, but the per unit price and count are omitted here. In this way the manager can easily create new pizzas to sell (which will suggest a reasonable price based on the input ingredients), or really any arbitrary food item. This was an important design decision – to sacrifice some specificity towards pizza and drinks with an eye towards allowing extensibility. Phrased another way, trading away some cohesion, in return for lower coupling.

Sales (by which we mean discounted offers) are composed in much the same way as Items. They contain a list of InventoryItems that are matched against. Any Item being sold that matches all the items in the sale qualifies for the sale price. The sale price can take the form of a discount percent, a subtractive amount, a set price, or a reference to another Item, whose price will be taken. Henceforth we tend to use the word “special” to describe a sale, as to avoid confusion with a sale being the normal thing an employee does.

Storage

The system stores four sets of information, customer data, order data, special data, and inventory data. It stores these objects and simple key value serialized versions of the representative classes with arbitrary keys. This is done as an extensibility point, and to avoid worrying about table design. The only drawback of this design is that secondary indices effectively have to live in the various classes. For example, an order has a pointer to the customer object in storage. We represent this as type Customer* for clarity even though it is just a string. This design makes it somewhat difficult to generate reports based on queries like pizzas sold on Sundays, but we expect the magnitude of this data to not prohibit running that query by simply scanning the entire set of orders.

Flow

On startup, the system loads from the inventory store to populate the employee GUI. As customer interaction progresses it builds orders with the Orderbuilder class, and proceeds to build and process orders. The orders go through several states, starting with opened and then proceeding through validated (ie all the inventory items are present, customer is in delivery range, etc), processed (meaning out of the kitchens hands), paid, and delivered. An order can also be voided at the discretion of the manager. This is intended to mean no money changed hands. If inventory changed hands the manager can “waste” inventory through his interface, but it doesn’t affect order state.