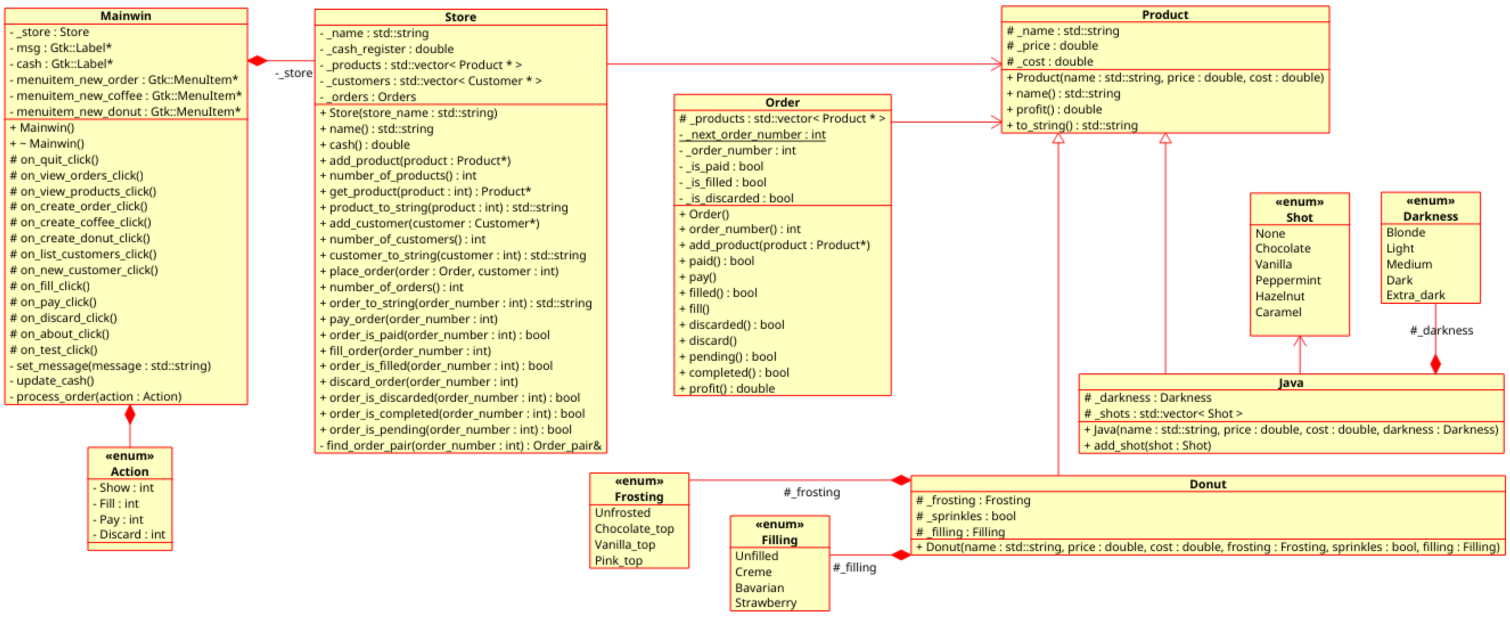
# Sprint 4 Guidance
# Java And Donut Express (JADE)

CSE 1325 – Fall 2018 – Homework #12
Due in Blackboard November 27 2018 at 8:00 am

The 4th sprint includes 13 “points”, filling in quite a bit of Store’s functionality. No constraint is required this week other than preparing for delivery of version 1.0! Note that the estimating team removed file I/O from the required features for the final sprint – those are now bonus features – and also reworked the next 6 features to better fit the Customer’s needs. The new Scrum Product Backlog is attached.

The class diagram for the suggested solution follows, but your implementation may vary.



Mainwin
- _store : Store
- msg : Gtk::Label*
- cash : Gtk::Label*
- menuitem_new_order : Gtk::MenuItem*
- menuitem_new_coffee : Gtk::MenuItem*
- menuitem_new_donut : Gtk::MenuItem*
+ Mainwin()
+ ~ Mainwin()
# on_quit_click()
# on_view_orders_click()
# on_view_products_click()
# on_create_order_click()
# on_create_coffee_click()
# on_create_donut_click()
# on_list_customers_click()
# on_new_customer_click()
# on_fill_click()
# on_pay_click()
# on_discard_click()
# on_about_click()
# on_test_click()
- set_message(message : std::string)
- update_cash()
- process_order(action : Action)
-_store
Store
- _name : std::string
- _cash_register : double
- _products : std::vector< Product * >
- _customers : std::vector< Customer * >
- _orders : Orders
+ Store(store_name : std::string)
+ name() : std::string
+ cash() : double
+ add_product(product : Product*)
+ number_of_products() : int
+ get_product(product : int) : Product*
+ product_to_string(product : int) : std::string
+ add_customer(customer : Customer*)
+ number_of_customers() : int
+ customer_to_string(customer : int) : std::string
+ place_order(order : Order, customer : int)
+ number_of_orders() : int
+ order_to_string(order_number : int) : std::string
+ pay_order(order_number : int)
+ order_is_paid(order_number : int) : bool
+ fill_order(order_number : int)
+ order_is_filled(order_number : int) : bool
+ discard_order(order_number : int)
+ order_is_discarded(order_number : int) : bool
+ order_is_completed(order_number : int) : bool
+ order_is_pending(order_number : int) : bool
- find_order_pair(order_number : int) : Order_pair&
«enum» Action
- Show : int
- Fill : int
- Pay : int
- Discard : int
Order
# _products : std::vector< Product * >
- _next_order_number : int
- _order_number : int
- _is_paid : bool
- _is_filled : bool
- _is_discarded : bool
+ Order()
+ order_number() : int
+ add_product(product : Product*)
+ paid() : bool
+ pay()
+ filled() : bool
+ fill()
+ discarded() : bool
+ discard()
+ pending() : bool
+ completed() : bool
+ profit() : double
Product
# _name : std::string
# _price : double
# _cost : double
+ Product(name : std::string, price : double, cost : double)
+ name() : std::string
+ profit() : double
+ to_string() : std::string
«enum» Shot
None
Chocolate
Vanilla
Peppermint
Hazelnut
Caramel
«enum» Darkness
Blonde
Light
Medium
Dark
Extra_dark
#_darkness
Java
# _darkness : Darkness
# _shots : std::vector< Shot >
+ Java(name : std::string, price : double, cost : double, darkness : Darkness)
+ add_shot(shot : Shot)
«enum» Frosting
Unfrosted
Chocolate_top
Vanilla_top
Pink_top
#_frosting
«enum» Filling
Unfilled
Creme
Bavarian
Strawberry
#_filling
Donut
# _frosting : Frosting
# _sprinkles : bool
# _filling : Filling
+ Donut(name : std::string, price : double, cost : double, frosting : Frosting, sprinkles : bool, filling : Filling)

A brief description of suggested changes to each class follows.

## Donut, Java, Customer

No changes are anticipated.

## Product

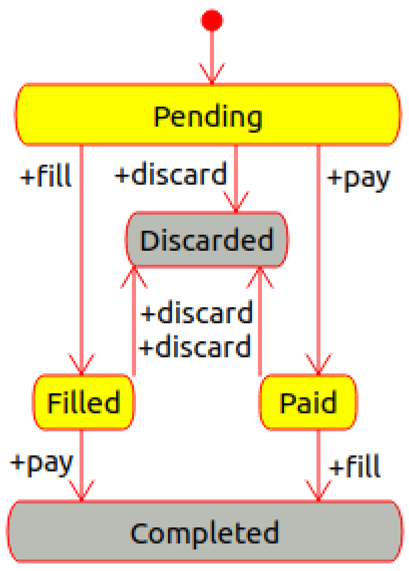
Because feature CE adds a cash register that is updated as orders are paid, we need to know the profit (price – cost) of each order. Thus a small method is required here.

## Order

Similarly, Order’s profit method returns the sum of the profits of each of the products included in the order.

For feature MST, to support tracking the state of an order, the Order class requires some enhancement. You have options.

The simplest approach is to implement a state machine similar to the one shown to the right, as covered in Lecture 24.



Alternately, as shown in the suggested class diagram, three Boolean variables could be maintained that in aggregate represent Order state, all initially set to false. These variables together define the state of the order, represented by the methods shown on the class diagram: pending(), paid(), filled(), completed(), and discarded().

- Pay() sets _is_paid true as long as both _is_paid and _is_discarded are false. Otherwise, it throws an exception indicating an invalid operation (your GUI should prevent this exception by, for example, not offering to pay an order that is already paid or that has been discarded).
- Fill() sets _is_filled true as long as both _is_filled and _is_discarded are false. Otherwise, it throws an exception.
- Discard() sets _is_discarded true as long as both _is_discarded and completed() are false. Otherwise, it throws an exception.

Similarly, the 5 states of Order can be calculated based on the 3 Booleans. A logic table with the 3 Booleans on the left and 8 rows should help you derive those.

Or you can implement the state machine directly, e.g., using an enum for the states, which was originally the plan when Lecture 24 was on the Tuesday on which sprint 4 started.

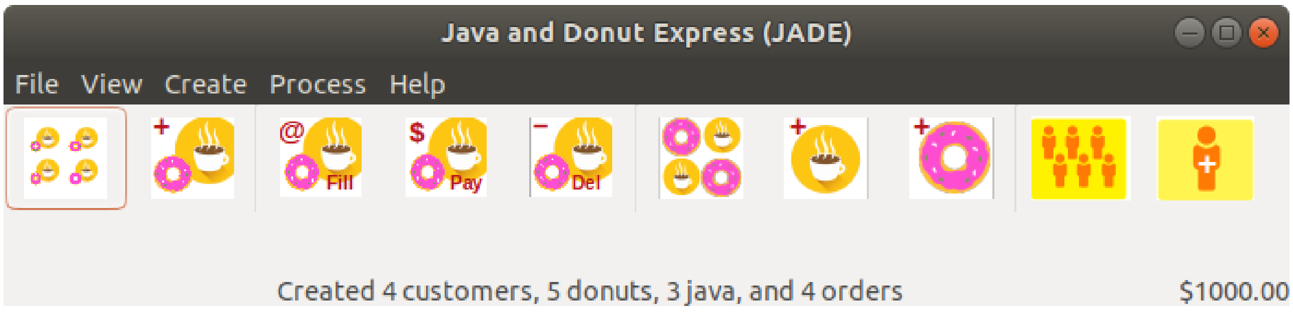
## Store

Store includes proxies for each of Order’s state-oriented methods, passing the order number whose state should be checked or changed. If order is associated with the customer placing the order via a map, this will involve searching through the map (think a for each loop in Store::find_order_pair) for the pair with the matching order number.

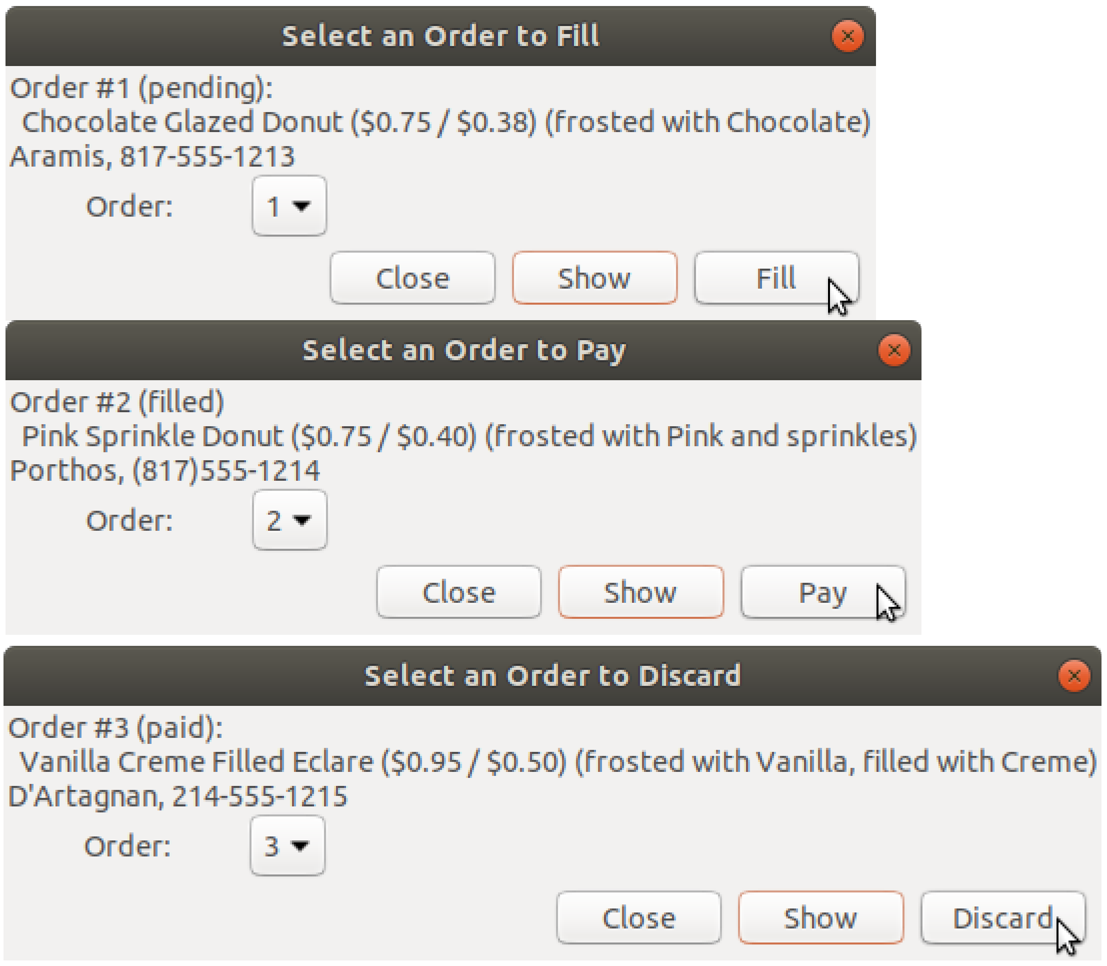
Note that the for each loop may return a *const* Order, Customer pair in which Order cannot be changed (to fill, pay, or cancel it). The solution is to use the copy constructor to copy Order and Customer to separate stack variables, erase the pair from the map, update Order, and recreate the pair in map. It sounds a lot more complicated than it is, truly!

## Mainwin

Additional changes are needed to Mainwin to add menu items and buttons for initiating the pay, fill, and discard behaviors. One option is to respond using a dialog to select an order, similar to what was used in sprint 3 to view orders (you could even generalize the sprint 3 view method into a common method named process_order, with an Action enum class to indicate which behavior was requested, as shown on the above suggested class diagram).



Created 4 customers, 5 donuts, 3 java, and 4 orders $1000.00



Order #1 (pending):
Chocolate Glazed Donut ($0.75 / $0.38) (frosted with Chocolate)
Aramis, 817-555-1213

Order #2 (filled)
Pink Sprinkle Donut ($0.75 / $0.40) (frosted with Pink and sprinkles)
Porthos, (817)555-1214

Order #3 (paid):
Vanilla Creme Filled Eclare ($0.95 / $0.50) (frosted with Vanilla, filled with Creme)
D'Artagnan, 214-555-1215

## Main

No changes are expected (ever!) to main.cpp.

## Bonus Features

For students interested in attempting bonus features for additional points, here's the background information on the first few. Feel free to email questions.

### SAVD – Save All Data to Default File on Exit or Command

### LOAD – Load all data from a default file on startup

The two features are similar to the CSE1325 Paint project and Homework 6 Extreme Bonus. Identify where all of your data is stored (in the suggested solution, that would be the Store class). The add save(ostream& ost) methods and *classname*(istream& ist) constructor sto all of the classes on which it relies, such that all of the data is streamed to ost on save such that the same objects can be reconstructed and added back to Store on load.

You'll need to add a File → Save for full credit here, as well as invoking save when exiting and load when opening for natural persistence.

### CM – Create roles with (simple) logins that change the GUI

This is in effect the Role menu in the suggested User Interface document, such that the user can switch roles between Owner, Manager, Server, and Customer. You may request a common password for all roles (optionally except Customer), or you can just be trusting and switch directly.

On a role switch, set the sensitivity of each menu item and toolbar to true if that role is permitted to perform that action, or false if not. The suggested User Interface  document lists the role(s) that should be made sensitive for each menu item listed, though you're free to adjust if you have better ideas – but the educational goal here is to use set_sensitivity on every menu item that may be affected on every role change, which means those menu items and tool buttons must be class attributes.

In addition, for the Customer role, don't show the cash register in the status bar (or wherever). Naturally.

### PIX - Add and display pictures for each item

This is NOT adding a photo onto the tool bar buttons – this is including a DIFFERENT picture with EACH product item that can be displayed (in-line or via a button) when the Customer is placing an order.

This will require adding a filename to the Product class and collecting that filename in the dialog when a new Java or Donut is created, and (at least) a button on the Order dialog to display a photo of the selected item.

Adding an image to the Order dialog, and automatically displaying the picture on the Order dialog when a selection is made from the drop-down, is worth more points – this requires a callback on Gtk::ComboBox::signal_changed (), which is a great practice challenge.

## POS - Show the products in an order for the servers

If you make it this far, here’s a relatively easy one – just display on request the minimum information on an order that has not been filled (states Pending or Paid) to enable a server to fill the order. This is basically just a message dialog with the consolidated list of products (e.g., 3 Caramel Machiattos, not 3 separate listings of Caramel Machiatto) for the entire order.

## P&L – Add a Profit & Loss Statement

Finally, you get to use the main body of the main window!  For the owner role by default, display a P&L in the in the main window.

This table should show how much of each product was sold, the gross revenue (price x # sold) of each, the cost of goods (cost x #sold), and the profit or loss (gross revenue – cost of goods).

Display as a *scrollable* table with one row per product, EITHER as a true table or as a Pango-formatted label or text box.

## AI – Restock Donuts and Coffee

Modify your Store class to keep track of how many of each product are available at the front counter, and decrement for each product sold. If a product runs out (quantity is zero), disable ordering any more of that product.

Add Edit → Restock to add additional items to the front counter.

## RI – Display an Inventory Report

As with P&L, but display columns for each product showing number at the front counter and total number sold.

## RO – Create an Order Report

For the Server and Customer role by default, but also available for the Manager or Owner roles, show a scrollable table for Pango-formatted label list all Orders in the Pending, Filled, or Paid state (omitting those in Discarded or Completed state).

Include the order number, state, total price, and what was ordered similar to POS above.

## RB – Display a Customer Report

For the Manager role by default, but also available for the Owner by request, show a scrollable table of all customers’ name and phone numbers along with any Pending, Paid, or Filled orders for them.

## EF, ROA, SALL, LALL, CENEW, XI, CS, ASTO, XT, and XXT

See me if you get to these.  :-)  Note that not every menu item in the User Interface design has a supporting feature. Maybe *next* release!