Shawn Hillyer

CS 162

Date 10/08/2015

Assignment 3 – Design, Test Plan, & Reflections

Reviewing the specifications, I've identified the following nouns and actions which are related to specific classes

Class: Item

Nouns / Data: item name, unit (ie can, box, pounds, ounces), number to buy, unit price

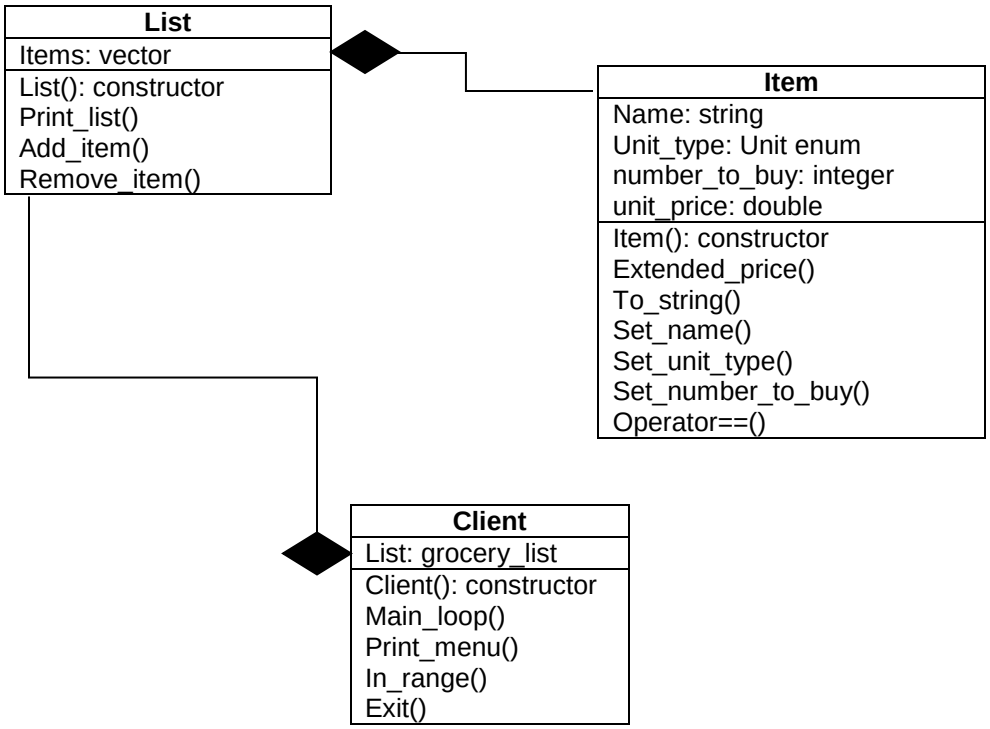
Verbs / Methods: constructor, extended price, print to screen, overloaded operator==()

Class: List

Nouns / Data: items

Verbs / Methods: create a list (constructor), add items, remove items, display the list, prompt for item data, get total price of list items

Based on that, the two classes can be diagramed as such. Note that I have added a Client class that will extend the object oriented design such that I can initialize the list and call a loop and print menus without modifying the List or Item designs at all. I am pretty confident I will extend my client a little bit.



Additional requirements

Add_item():

- Must prompt the user for the name, unit_type, number_to_buy, and unit_price.
- Must also use comparison operator == to validate if Item is in list already and not add duplicate items
    - This will be useful in the remove_item() method as well

Print_list():

- Must display each item in the list:
    - Name of items
    - # of items
    - Unit_type
    - Unit_price
    - Extended_price
    - Total price of all items

Use of the classes

Client will be called by main and will be the starting point for my program. It will instantiate a single List item and control the loop during Main_loop(). The Main_loop will call print menu, validate that the options input are in range using in_range, and finally an exit call will drop us out of the program.The Client will call on the List methods like print_list() and add_item() using the menu options.

List will be the data structure that holds our list of Items. The constructor will create a vector of type Item that will hold our list of items. We use the vector because this is a safe type which handles memory allocation, The add_item() method will prompt the user for the relevant data and instantiate an item. Then the item will be pushed to the vector. If the operation fails, it will signal this to the caller, Client, so that Client will know there was a filaure.

Additionally, remove_item() will search the vector for a specified item as identified by its name, only. The user will supply this item's name and the operation will fail if the item is not found in the list. As did add_item(), it will return false if the operation failed.

Print_list() will iterate through every item in the list and call that item's to_string() method, which should provide the name, unit type, unit price, and number_to_buy required in the form of a string. The item's extended_price() method will be called on each iteration to increment a local list_total variable that stores the total price of all the items. The vector .size() method will provide us the total price of all the items.

Item will initialize all of its data with the constructor, which will in turn call the setters for each data member. These will validate that the number_to_buy is greater than 0, the unit_price is non-negative, and set the name as a std::string type. The operator==() will return true of two Items that are compared have the same name.

To calculate the extended_price(), I will simply multiply the unit_price by the number_to_buy. This is to prevent stale data from occurring. This method will be called by the List::print_list() method.

Testing Plan / Methodology

I will use incremental development and test methods as I implement them. Since I have such a solid design laid out, I first write all of my headers/class declarations, write a main() client that instantiants my Client, and create a very simple main loop that is based in the framework I used for my GameOfLife. However, this time I have encapsulated the main logic loop and other relevant functions into the Client class. Then I'll write out each method and check for compile-time errors only until every method is written. Once I have a compiling set of classes, I'll go through and test the Client so that I can use it to drive my testing automatically, then Item, and finally the List class.

Item

Item() Constructor – should accept 4 arguments and initialize the data members appropriately. Calls the setters to do ensure all validation of input in one place. Testing will be reliant on the setters properly validating and assigning the passed arguments. Also, the to_string() method will need to be implemented to verify the data has been initialized (I'm providing no direct data accessors in this class).

Extended_price() – Accepts no arguments. Returns the unit_price X number_to_buy.

Set_name() – Accepts a string as an argument and sets name to that value.

Set_unit_type() – accepts a unit_type enum value from the list defined in unit_type enum header.

Set_number_to_buy() – Accepts an integer as an argument and initializes the data member number_to_buy after validating is greater than 0.

Set_unit_price() – accept a non-zero positive double and initialize unit_price.

Operator==() – compares two Item objects and returns true of their name data member is the same.

Item test cases to include:

- set_name() passed no characters, one word, multiple words
- set_unit_type() passed UnitTypes – variety of types
- set_number_to_buy() passed negative, 0, 1, and variety of values (to test extended price
- set_unit_price() passed negative, 0, 1, and variety of values (to test extended price)
- extended_price() will return the number_to_buy * unit_price

List

List() constructor – The list constructor won't do anything other than instantiate the class. Its only data member, a vector, will be empty by default, which is what we want.

Print_list() – This should print the string representation of every Item in the list (per the specifications above) – name of item, # of items, unit type, unit price, extended price, and total price of all items in the list.

Add_item() – Should add an item only if an item with the same name does not exist in the list. Returns true if it works, false if it doesn't. *Calling code* should print a message echoing the added item's string value or an error message if not.

Remove_item() – Should remove an item only if an item with the same name exists in the list. Returns true if it works, false if it doesn't. *Calling code* should print a message echoing the removed item's string value or an error message if not. For user feedback.

List test cases to include:

- Print_list() will need multiple tests before and after adding and removing items to verify items are on/off list, and that it reports the count and total price correctly (as these are local calculations)
- Add_item() should be tested using the testing parameters from Item, as the values here will be passed to Item to initialize the objects. Error-handling here should cause Item not to be improperly initialized. Also need to test using same string multiple times to validate item not added more than once.
- Remove_item() should be tested by adding an item then attempting to remove the item and other (non-existant) items, using a print_list() between tests.

Client

Client() constructor – should call main_loop() when client is instantiated to begin the program.

Main_loop() – This loop will instantiate a List object and prompt user for a menu choice. Then branch into the appropriate logic by calling the methods on List to add and remove items and print the list.

Print_menu() – Should print the menu to the screen (only)

In_range() – validates user input is in the valid range for the main menu

Exit() – handles any necessary functions to close the program (for extensibility)

Test Cases

Client

Method	Purpose	Inputs	Expected Results / Outputs	Actual Results
Client() constructor	Verify client begins main_loop()	Client instantiated by main()	Main_loop() will be called and execute Should see menu option and allow for input	Main menu prints and validates user input and calls subroutines correctly
main_loop()	Verify menu prints and input is accepted and validated	-1	Reprompt for a selection, re-call print_menu()	Passes as described
	**Dependent on prnit_menu() **Dependent on in_range()	0	Reprompt for a selection, re-call print_menu()	Passes as described
		1	calls List::add_item()	Passes as described
		2	calls List::remove_item()	Passes as described
		3	calls List::print_list()	Passes as described
		4	calls exit()	Passes as described
print_menu()	Verify menu prints to	Called by main_loop()	Prints menu to screen with 4 selections (1. Add Item 2. Remove Item 3. Print List	Passes as described

	screen		4. Exit) Passes as described	
in_range()	verifies user input properly validated	menu_choice (local var) from cin, MIN_CHOICE and MAX_CHOICE local constants	The test cases for main_loop should verify this function works.	Passes as described – only the options 1-4 are accepted
exit()	Verify any exit related code executes	Called by main_loop if "4"' input at main menu	Program terminates and executes any code in exit (probably just a goodbye message)	Passes as expected

The test script below also implicitly tests the constructors and the overloaded operator==() method of Item.

The idea is to hard-code a test script that tests using these values then run a “live” test using the same test script and we should see the same result. The hard-coded test exists to ensure that the setter methods ALSO validate their input because I’m coding validation into the List as well. I want to make sure the Item works as a standalone module by calling constructors manually.

Step	**Method call / user input**	**Input / arguments**	**Expected result**	**Actual result**
1	print_list()	none	Prints number of items and total price are both 0	Passes as described
2	add_item()	Item1,BOX,1,1	"Item1 added to list."	Passes as described
3	print_list()	none	Prints Item1.to_string() value, extended price = 1, count = 1, total price = 1	Passes as described
4	add_item()	Item1,GALLON,1,1	"Item1 already on list!"	Passes as described
5	add_item()	Item two, BOX,1,2	"Item two added to list."	Passes as described
6	add_item()	3rd item, POUND,2,2	"Item two added to list."	Passes as described
7	add_item()	Eggs, DOZEN, 2,5.99"	"Eggs added to list."	Passes as described

8	print_list()	none	"Item 1: 1 BOX, $1.00 ea, $1.00 subtotal" "Item two: 1 BOX, $2.00 ea.$2.00 subtotal" "3rd item: 2 POUND, $2.00 ea. $4.00 subtotal" "Eggs: 2 DOZEN, $5.99 ea., $11.98 subtotal" Total for 4 items: $18.98	Passes as described
9	remove_item()	pizza	"Pizza wasn't on the list!"	Passes as described
10	remove_item()	EGGS	"EGGS wasn't on the list!"	Passes as described
11	remove_item()	Eggs	"Eggs removed from the list."	Passes as described
12	print_list()	none	"Item 1: 1 BOX, $1.00 ea, $1.00 subtotal" "Item two: 1 BOX, $2.00 ea.$2.00 subtotal" "3rd item: 2 POUND, $2.00 ea. $4.00 subtotal" Total for 3 items: $7.00	Passes as described
13	add_item()	"",BOX,1,1	"Please enter a name at least 1 character long."	Passes as described
14	add_item()	Lucky Charms,BOX,-1,1	Manual entry - will reprompt for units	Passes as described
15	add_item()	Lucky Charms,BOX,3,0	Manual entry - will reprompt for price	*Failed* – program logic corrected
16	add_item()	Lucky Charms,BOX,3,2.75	"Lucky charms added to list."	Passes as described

Reflections

Testing went well. I wanted to think about how I represent the string for each Item in to_string() and how I would build it up, so I decided to make my output very simple and just print the name of the Item to make sure the core logic worked in Client and that it basically didn't crash.

Firing up the program, the main menu loop worked fine. Calling List::add_item() and List::remove_item() and the print method all worked as expected out of the gate. I was even able to add and remove items and the operator overload worked on the first go.

However, everything I implemented skipped over actually *using* UnitType. Because I couldn't easily represent it as a string, first of all, and also that the enum list was in all caps. But that's just the beginning.

There were a few other problems. First, if I were to add a type to the list of UnitType that I created, I knew I'd have to try and modify a few things I had not yet implemented. I'd have to make a method for the user to select that enum value during a prompt, which is one modification. I'd have to convert each one to a string somehow, which

is a second modification. Then there is the problem of how to handle singular units (1 box) vs plural (2 boxes). It was starting to look messy.

It seemed that I could convert UnitType to a class with a few data members. Namely, a UnitType might look like:

UnitType
Base_name: string Plural_suffix: string
UnitType(str base, str suffix) : To_singular() – returns base_name To_plural() – returns base_name + plural_suffix

** Items will “has-a” relationship with UnitType

I took a break at this point in my testing to write the above class outline and talk through those steps in my reflections. Today (10/9/15) on the ride home from work I set to implementing this class so that I could verify my program works entirely.

My idea was that the UnitTypes could be constructed at the List level by hard-coding in (for now) the relevant strings and pushing them onto a vector maintained inside the List class. I would ensure all of these objects exited by calling this little factory inside the List constructor.

By having this list of unit_types maintained as a data member, the List could then iterate over the UnitType vector and print the valid options for the user. They select the unit type for each item they add to the list by selecting the index of the UnitType they want.

The (much-simpler to implement) option would be to have the units be a simple string defined by the user. This wouldn’t fix the plural problem, but it would let the user quickly fill in their units each time.

After revising UnitType to be a class with the structure I planned, I then had to make the other changes. My add_item() method was growing too big. So I decided to refactor out each prompt to a private method that validates the input, and this really helped the UnitType logic get set aside for last. I spent a fair amount of time just formatting the string output of Items and how the report printed for the entire List printing (about 40 minutes) and then finally got to implementing the prompt_unit_type() method in List.

It worked out beautifully. At this point, everything seemed to be fully functioning, so I set about using my full test script. The results are documented in the tables above at that point in time.

I did notice that my add_item() return value wasn’t being used in the way I implemented the program. However, I left it in because I think this could be useful for error-handling or extending the methods elsewhere.

Also I realized my test script didn’t explicitly plan to check adding and removing the first and last item added to the list (to ensure I had no off-by-one type errors in my delete algorithm), so I did that as well.

All in all, having a robust design was a huge boon. Also, I’m glad that my initial plan called for an Enum for UnitType. When I decided to abstract out the details of how to print and handle unit types as strings, it didn’t take much effort to refactor the rest of my code because it was in the same header file, same type name, etc.