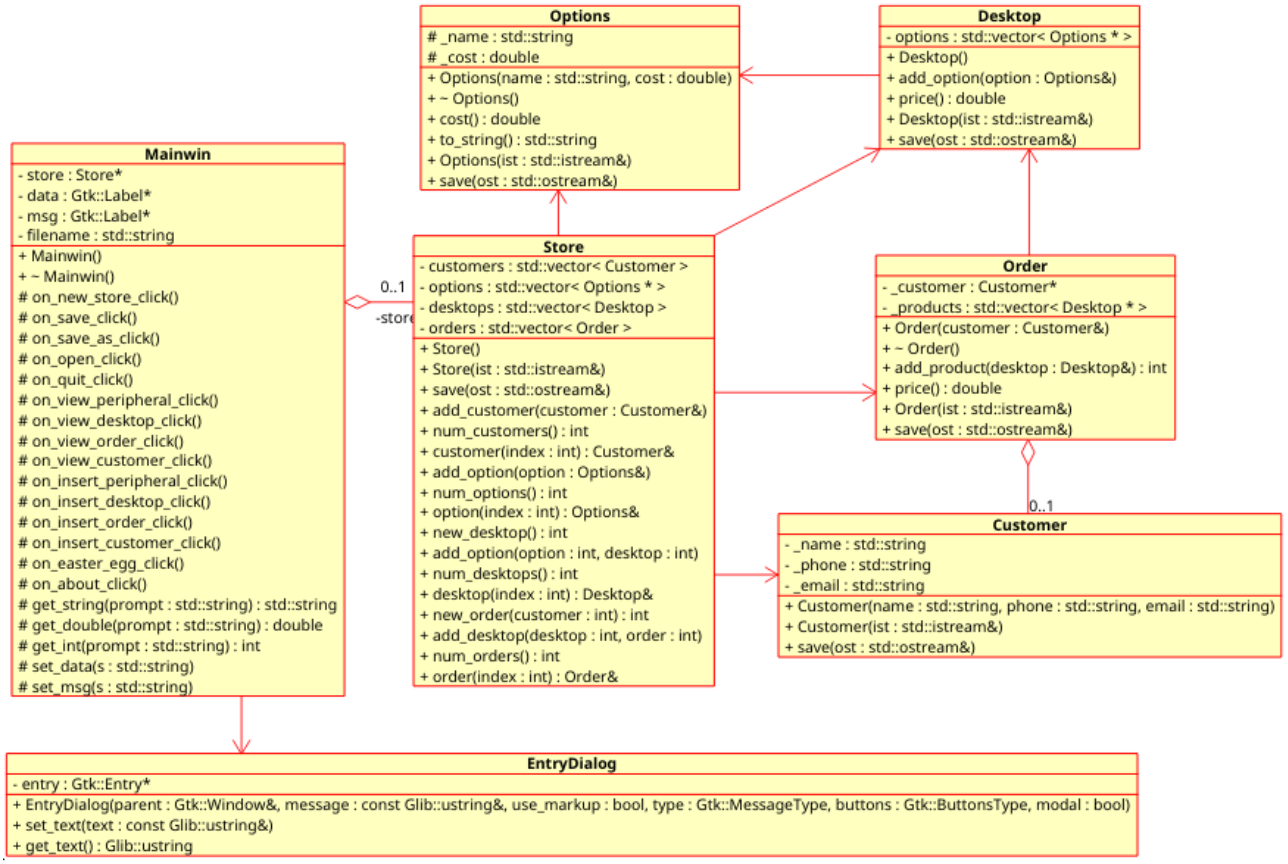
Sprint 3 Guidance

CSE 1325 – Spring 2020 – ELSA

IMPORTANT: Do NOT use full_credit, bonus, or extreme_bonus subdirectories for the final project. Keep all files to be graded in the cse1325/elsa directory on GitHub for the duration of the final project – source code in cse1325/elsa/src, and the spreadsheet in cse1325/elsa/doc.

For Sprint 3, you will add the ability to load and save ELSA data to files.

This class diagram is for sprint 3 only (also available at **cse1325-prof/elsa/sprint3/Sprint_3.xmi**):



This sprint adds roughly the same lines of code as was needed for the previous 2 sprints.

Concept

The concept described here follows that of Lecture 15, to wit: Each class that contains program data is responsible for saving its data to a stream (via its **`save(std::ostream& ist)`** method) and to construct itself from a stream returning the exact same data (via its **`Class::Class(std::istream& ist)`** constructor).

- Basic types like int, double, and std::string are simply streamed out by the save method.
- Custom types are streamed out by calling their respective **`save(std::ostream& ost)`** methods.
- Vector types first stream out the vector's **`size()`** (on a separate line), then save each element to the ostream in vector order.

ELSA has an interesting twist with its aggregation between classes. *You cannot save a memory address (i.e., a pointer) to a file*. When the data is loaded at a later time, it will almost certainly be loaded to a different address, so the addresses loaded into the pointers will be wrong.

You have two options to address this.

- **Recommended: You are permitted to simply duplicate data during file I/O.** That is, when saving the file, instead of recording the *reference* in some way, you may simply write out another copy of the referenced object to the file. When you later load the data, you may then simply recreate a duplicate object on the heap and reference it. This isn't fully consistent with the class diagram, and would cause memory leaks in the long run, but your code should work just fine for now, and *you won't be penalized for taking this approach in any way*. This is the recommended approach for most students, which I describe below.
- **You may attempt to save and restore the aggregate references.** The (optional) Appendix describes one way to accomplish this, although other approaches are fine as well.

Scrum Spreadsheet

Allocate the features you intend to complete this sprint on your Product Backlog tab. You should at least allocate through those features with a "3" in the Required column, as these are the features to be graded at the end of this sprint.

Then read through the suggestions below, and create your anticipated tasks on the Sprint 03 Backlog tab for each feature, setting the Feature ID of the feature that each task helps implement.

IMPORTANT: You may NOT simply baseline the complete Sprint 2 Suggested Solution for this sprint! If you have issues with the previous sprint's code, you may examine the Suggested Solution to help you bring your code up to where we are – just add tasks for this in your Sprint 03 Backlog.

Once ready, proceed with Sprint 3, marking each task as complete on the day you finish it.

Class Overviews

Store, Customer, Options, Desktop, and Order

Start by add a matching pair of members to each class – a save method that accepts a std::ostream& parameter, and a constructor that accepts a std::istream& parameter. Leave the implementations empty for now.

Mainwin

Install the I/O Framework

First, add the new attribute Mainwin::filename of type std::string, and initialize it to your default filename, e.g., "untitled.elsa". This attribute remembers the most recently saved or opened filename, so that File > Save can rewrite it.

Now add 3 new callback methods: on_open_click(), on_save_click(), and on_save_as_click(). Leave these implementations empty as well for now.

Finally, add menu items File > Open, File > Save, and File > Save As, and connect their signals to the respective callback methods.

Build, test, and verify that your user interface now has the above menu items. **Add, commit, push.**

Add the Open and Save As Dialogs

Now add a **Gtk::FileChooserDialog** to both **on_open_click()** and **on_save_as_click**, configured so the user can select or specify an ELSA filename. I recommend that you *baseline*[1] the equivalent code from the nim Mainwin class at **cse1325-prof/15/nim/mainwin**.*

In **on_save_as_click()**, run the dialog and then assign the user-provided filename to Mainwin::filename. Then delegate to on_save_click(), which simply saves to Mainwin::filename.

In **on_save_click()**, open an output file stream using filename, then call Store's save method with this stream as the parameter. **Be sure to handle any exceptions.**

In **on_open_click()**, run the dialog and then assign the user-provided filename to Mainwin::filename. Open an input file stream using filename, delete the old store, and instance a new store with this stream as the constructor parameter. Refresh the data area of the main window with the loaded data. **Be sure to handle any exceptions.**

Build, test, verify that your user interface now has the above menu items. Open should always give you an empty store (you're not loading anything yet!), and save should always give you an empty file on the disk. **Add, commit, push.**

Store

The Store class has 4 attributes, each a vector. Start with one of them – I'd suggest customers – and code until you can reliably save and load just Customers from the user interface. Then implement options, and then desktops, and finally orders. Let's start with customers.

To save customers, in Store::save, first stream the number of customers to the output stream *on a separate line*. Then, iterate over all customers, calling the save method with the output stream for each. Finally, check the state of the stream – if not good(), throw an exception while we still can!

To load customers, in Store::Store, first read the number of customers from the input stream. If you use ist >> csize, don't forget to ist.ignore(32767, '\n') or equivalent to ensure you leave the stream pointer at the start of the next line. Then, for each customer to be read, construct a Customer object using the input stream as parameter, and push it onto the customers vector. Again, check the state of the stream – if not good(), throw an exception while we still can!

Customer

Now it's time to implement Customer::save(ostream&) and Customer::Customer(istream&).

For Customer::save(ostream&), just write the name, phone number, and address to separate lines, leaving the output buffer pointer to the start of the next line.

For Customer::Customer(istream&), just getline the name, phone number, and address. Easy peasy.

Build, test, verify that you can save customer to a file and then read them back. Try to break your code. When you can't, **add, commit, push.**

1This just means you take the unmodified files from the original source and add / commit them to your current project's git repository, *then* modify them to fit your new requirements.

Store and Options

Now do the same for options, adding code to save and load the Store::options vector to Store::save(std::ostream&) and Store::Store(std::istream&).

Then implement Options::save(std::ostream&) and Options::Options(istream&). **Build, test, verify** that the options save and reload as well. **Add, commit, push.**

Store and Desktop

Now do the same for desktops, adding code to save and load the Store::desktops vector to Store::save(std::ostream&) and Store::Store(std::istream&). Now, Desktop contains a vector of pointers to a vector of Options objects in Store, but you may ignore this and write out a duplicate copy of each Options and then reload the duplicate in the constructor.

Then implement Desktop::save(std::ostream&) and Desktop::Desktop(istream&). **Build, test, verify** that the desktops save and reload as well. **Add, commit, push.**

Store and Order

Finally, do the same for orders, adding code to save and load the Store::orders vector to Store::save(std::ostream&) and Store::Store(std::istream&). Of course, Order contains a reference to a Customer in Store and a vector of pointers to Options objects in Store, but you may ignore these and write out a duplicate copy of each Options and then reload the duplicate in the constructor. (You will likely need to change Customer& to Customer*, for reasons that will become clear.)

Then implement Order::save(std::ostream&) and Order::Order(istream&). **Build, test, verify** that the orders save and reload as well. **Add, commit, push.**

Test Error Handling

Write out a file and analyze your file format. Then modify the file to break the file format and try to load it into ELSA. Does your program handle bad data ungracefully, or do you show a simple error message and keep going? Or do you segfault? Avoid segfaults if at all possible.

Write-protect a file and try to save to it – does your program handle that case?

Keep trying to creatively break your program, and you could become a TA! :-D

What Is To Come

Exam #2 will distract us from our project for a week before the final sprint.

In **Sprint 4** we'll replace one of the sequence of dialogs in this sprint's implementation with a single custom dialog. Instead of a dialog asking for the customer's name, then another asking for their phone number, then yet a third asking for the email, you'll present a single dialog with 3 labeled Gtk::Entry widgets, and Add button, and a Cancel button. Better!

In addition, we'll collect an additional attribute, the number of gigabytes, for each RAM peripheral. This means we'll need to invoke the operator<< function *polymorphically*, which is a bit of a trick since polymorphism only works with classes. :-D Not to worry, we'll walk you through it all.

Bonus Features: To earn additional credit, you may implement additional features beyond those required. For example, adding other derived classes for disks or CPUs, replacing the other sequences of dialogs with custom dialogs, making the data area in the main window scrollable,

You will only receive bonus points for implementing bonus features actually listed in your Scrum spreadsheet and approved by the professor. A few pre-approved bonus features are provided in the Sprint 2 Scrum spreadsheet, with the *maximum* bonus points available for an *excellent* implementation in the Bonus column (less excellent implementations will receive appropriate partial credit). **You may suggest unique features** that you would like to implement, and if appropriate and in line with our educational objectives, I will approve them. **This allows you to earn credit for taking the project in your own unique direction.**

Our project is complete at this point, though we'll hopefully have time for a few students to present their projects to the class (for extra credit, of course). Our last assignment will be a **stand alone multi-threaded app**, and then we're on to **Exam #3** and wrapping it all up with a bow on top!

OPTIONAL Appendix: Supporting Aggregation

For Advanced Students Only: Saving and restoring pointers to data in other classes is messy, which is why it's not required for this assignment. If you really want to do it anyway, here's how I implemented a second branch of the code that seems to accomplish it.

Consider Desktop. It has a vector of pointers to Option instances in the options vector in Store. What we need to write to the disk is the index into that vector to which each pointer in our vector points. Similarly, when loading the file, we'll load the index from the file, then we'll need to get the address of that indexed Desktop in Store::desktops.

But Desktop has no visibility to Store, and it's a Bad Idea to make them mutually dependent. So, the only way I can see to accomplish the two things in the previous paragraph is for Store to pass a reference to the Store::desktops vector as a second parameter to Desktop::save and to Desktop::Desktop. This allows those two members to translate the pointers into indices, and back again during reconstruction.

Order::save and Order::Order will need both a reference to the Store::customers vector and the Store::desktops vector, but can otherwise follow the same approach.

It's not pretty, but it's C++. :-)