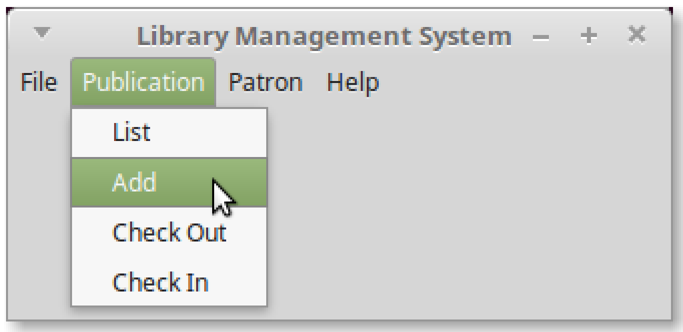
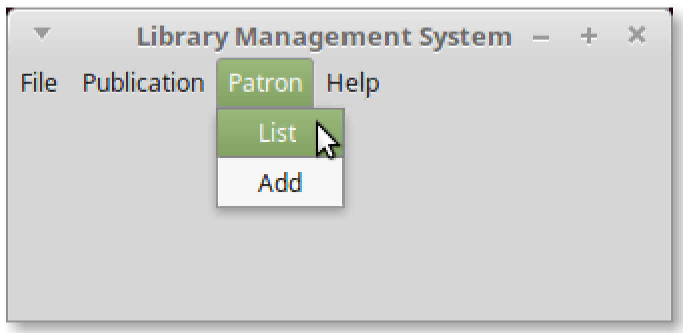
Getting Organized – Library Model

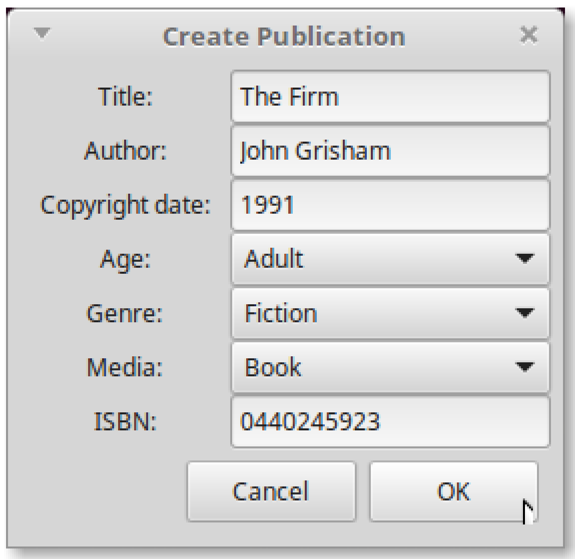
Sprint #3

CSE 1325 – Fall 2017 – Homework #5
Due Thursday, October 12 at 8:00 am

Command Line Interfaces (CLI) can be very productive for technical people, but are not acceptable to normal users. They prefer rich, graphical interfaces, and our job as professional software developers is to provide interfaces that delight our users. **Here's your chance!** We'll replace the exclusively dialog-driven Library Management System from last sprint with a full-featured main window-centric application!







Title:	The Firm
Author:	John Grisham
Copyright date:	1991
Age:	Adult
Genre:	Fiction
Media:	Book
ISBN:	0440245923

Full Credit

Port your own or one of the suggested solutions from Homework #6 (the Extreme Bonus is recommended) to include a main window. This will replace your own main loop with a Main Window class derived from Gtk::Window, complete with drop-down menus.

For this sprint, you may find the Nim game that we reviewed in Lecture 14 (with the source code that was attached) to be exceptionally helpful.

Last week's **Scrum spreadsheet** *already included* the features in the Product Backlog that we'll implement this Sprint. You should already have completed the Sprint 01 Backlog and Sprint 02 Backlog tabs, so this week, **plan and manage your tasks using the Sprint 03 Backlog tab**.

As in Sprints 01 and 02, you will be graded on the extent to which you cover the requirements and the quality of your code. As always, use git to version control your software. **You will receive partial credit if you only implement a portion of the requirements,** so don't hesitate to submit partial results early. If you don't understand the *intent* of a requirement, feel free to ask – although reasonable assumptions without asking are also fine if you've ever used a library. **If you have questions on *anything*, ask!!!**

Suggested Approach

I recommend the following approach for this sprint.

1. "Baseline" your dialog version in git. If you're building from your own code, you need do nothing, but if you adopt a suggested solution, **put it under git as is FIRST**.

   Then **build and test the resulting dialog-centric application using your Makefile**. Use just "make" with a suggested solution to **build the "lms" executable**, and then test it interactively (./lms). You can use "make debug" if you run into issues and need to run the debugger (it will execute a "make clean" first).

   You <u>must</u> start with a solid, working application before making any gtkmm changes.

2. Copy the Main_window class from the Nim application we reviewed in Lecture 11. This is derived (or inherits) from the Gtk::Window base class. So in your main(), you'll replace your "Gtk::Window win(controller);" or similar line with "Main_window win(controller);".

   Don't forget to #include main_window.h, and to update your Makefile to properly compile and link the Main_window class into your executable.

3. Now, decide on your menu layout. Users commonly expect a File, Edit, View, and Help top-level menus at a minimum, but for this 1-week sprint, **your menus need only cover <u>ALL</u> of the functionality in your CLI version**. Get the design right before you code; it saves a LOT of time (it avoids what we call "scrap and rework" in the biz).

Add each menu entry as a separate task on your Sprint 03 Backlog list. Also add implementing the associated callback. You may have additional tasks, such as updating the Help text, too. Think carefully and write them all down.

4. **Carefully adjust the code in main_window.h to reflect *your menu design***. You'll probably delete more than you add. You can test compile the header file with the -c file if you like to ensure it has no compiler errors before moving forward. If you get really lost, just 'git checkout main_window.h' and start over.

5. **Now edit main_window.cpp, adapting it to your menu layout.** Work in small sets of code between test compiles and git commits. The old C multi-line comment indicators /* and */ , or the preprocessor statements "#if false" and "#endif", are very helpful for "commenting out" large sections of code that haven't been ported yet.

   Try to get something simple – maybe just File → Quit – working quickly. Don't proceed until what you've written works well, and you understand *why* it works well. **Remember the debugger!**

   Build up the entire menu in small steps, until it's all there, even though clicking the menu items will generate no response.

6. **One at a time, implement each callback from your menu.** If you have an execute_cmd method on Controller, your callbacks can simply call this with the integer from the menu. For example, the callback from your "Add Publication" menu callback may be able to just call something like controller.execute_cmd(1).

   Again **– baby steps!** Address one task on your Sprint 03 Backlog at a time, get it compiled and working well, add it to git, and move on. As always, **make periodic backups.** "All technology fails eventually", so be prepared.

   Once every menu item is working and stashed into git, you should cast a critical eye at your application. Clean up your help text to reflect the GUI. You might want a nice custom dialog for Add Patron if you have one for Add Publication. You might want to get rid of those "GtkDialog mapped without a transient parent. This is discouraged." messages. Have fun!

   And don't forget to turn in your work properly formatted for maximum credit!

Deliverables

As in previous assignments, you will deliver to Blackboard a **CSE1325_06.zip** file containing:

- Your **updated Scrum spreadsheet** named **Scrum_P6.ods**, Scrum_P6.xlsx, Scrum_P6.lnk (containing a link to Google Sheets), or other spreadsheet format *that your grader agrees to accept* for this sprint at the root level – one file for all levels.

- A **full_credit subdirectory** containing:
  - Your **local git repository** including **.h and .cpp files** and a **Makefile** that is competent to rebuild only files that have been modified. As before, we don't normally try to write regression tests for the GUIs themselves, as this is quite difficult, so you should NOT need

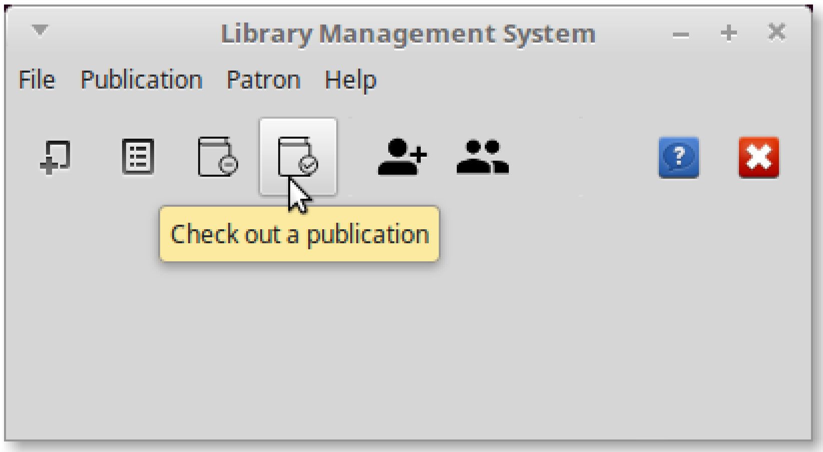
to add any regression tests (and none will be graded).

- ○ **Screenshot(s)** as needed to help the graders know what to expect when they "make".
- ○ Your **library.xmi** representing your class diagram. It almost certainly changed since Sprint 02 – you added a Main_window class, right? You generally do **NOT** include gtkmm classes on your own diagram, though – only where needed for clarity. None are expected, but if you want (for example) to show Main_window inheriting from Gtk::Window, that's fine.

Bonus

Add a tool bar to your LMS main window.

You MUST cover the 5 use cases we provided to you back in Homework #5 with buttons on your tool bar. You must include tool tips as well.



If you've added the ability to manage a list of patrons in your LMS, you'll get a bit of extra credit for also including those buttons.

You'll have to come up with icons. For some buttons, you may be able to use Gtk::Stock icons (see https://developer.gnome.org/gtk3/stable/gtk3-Stock-Items.html). But for some, you'll need to create your own or use something that you find on the Internet. In the latter case, failure to follow the images' license requirements will cost you a LOT of points. Attribution licenses such as Creative Commons or Gnu Free Documentation License (GFDL) are typically fulfilled by included the required text in your Help → About dialog box. (NOTE: Use Advanced Search in Google to filter to only reusable content.)

Deliverables

As in previous assignments, you will add to your **CSE1325_07.zip** the following:

- A **bonus subdirectory** containing:
  - ○ Your **local git repository** including **.h and .cpp files, image files,** and a **Makefile** that is competent to rebuild only files that have been modified. All should be git contolled.
  - ○ **A screenshot** demonstrating your main window, complete with gleaming tool bar.

Extreme Bonus

Add a scrollable table of Publications to your main window, just below the tool bar, such that it expands to fill all available space as the window is resized.

The table should show the 7 fields of data that we know about each publication (or more, if you based your code on Homework #5's extreme bonus), as well as an indication of whether each Publication is checked out (an icon would be nice, but text is acceptable).

Deliverables

As in previous assignments, you will add to your **CSE1325_07.zip** the following:

- An **extreme_bonus subdirectory** containing:
  - Your **local git repository** including **.h and .cpp files, image files,** and a **Makefile** that is competent to rebuild only files that have been modified.

  - **Screenshot(s)** demonstrating your table in action, as needed, in PNG format.

Frequently Asked Questions

From Homework #7

None yet.

From Homework #6

Q: I’m getting error “...”. What causes that?

Good question! Here are some common errors, what they mean, and how to correct them.

main.cpp:1:19: fatal error: gtkmm.h: No such file or directory
compilation terminated.

You forgot to add `/usr/bin/pkg-config gtkmm-3.0 --cflags –libs` to the end of your g++ compiler line for the indicated file (in this case, main.cpp) in your Makefile. That’s the clause that tells g++ where to find all of the gtkmm-related header files.

main.o: In function `main':
main.cpp:(.text+0x34): undefined reference to `Gtk::Main::Main(int&, char&, bool)'**
(with a LOT of other text!)

You forgot to add `/usr/bin/pkg-config gtkmm-3.0 --cflags –libs` to the end of your g++ linker line (the g++ line with all the .o files instead of .cpp files) in your Makefile. You made a reference to a Gtk+ resource that relies on a gtkmm-related compiled (.o) file, but didn’t tell g++ where to find it.

undefined reference to `Dialogs::input’ (with a LOT of other text!)

You forgot to add dialogs.o to your linker line (the g++ line with all the .o files instead of .cpp files).

Gtk-Message: GtkDialog mapped without a transient parent. This is discouraged.

This (legitimate) complaint by gtkmm shows up on cerr every time you pop up a dialog, complaining that you don’t have a main window. As I mentioned, that’s not really suitable for production, but we can ignore it for now. *Next* sprint we’ll fix it!

If you’d like to not see it anymore, run your program like this (Linux / Mac only):

```
./lmi 2> /dev/null
```

This literally sends your output stream #2 (STDERR / cerr) to /dev/null, an output device that simply ignores all output in Unix-like systems. Yes, it’s very useful at times. Note that it will also discard any error messages that YOU send to cerr - caveat scriptor!

Q. I want to add icons to my dialogs and tool bars. Where do I get an icon to use?

Oh, anywhere you like – *as long as you respect copyright law*. Remember, we talked about this!

- You may draw your own, if you have a vaguely artistic bent, or just don't care. The graders will not be judging your artistic ability.

- You may go to a public domain collection, as I did for my first book, or to a creative commons library, as I did for the suggested solution icon (from The Noun Project's Sathish Selladurai collection). The Wiki Commons is a good place to find art licensed under the Creative Commons or the Gnu Free Documentation License (GFDL).

- You may use licensed clipart that came with a program you own, or a clip art collection you previously purchased – I have several million images that for which we've purchased a perpetual license over the years (disk space is cheap).

If the grader detects a violation of copyright law, your grade could be reduced as low as a 0, and for more egregious offenses may result in your being reported for an honor code violation.

Integrity matters.

From Homework #5

Q. What does Publication::to_string() do?

As in earlier homework assignments, to_string returns a string containing a textual representation of the object. An example is shown in the requirements.

> "The Firm" by John Grisham, 1991 (adult fiction book) ISBN: 0440245923
> Checked out to Professor Rice (817-272-3785)

Q. What types are Genre, Media, and Age?

The types Genre, Media, and Age are left to your preferred implementation. Design something! Here are a few ideas to get you started.

Option #1: They are well-suited to be enums, with the enumerated values given in the requirements:

> The library consists of a lot of publications in several types of **media** – books, periodicals (also called magazines), newspapers, audio, and video. Each publication falls into a particular **genre** – fiction, non-fiction, self-help, or performance – and target **age** – children, teen, adult, or restricted (which means adult only).

The problem with enums is that it's obviously unacceptable to print "age 3" or "media 5" when printing out a publication – you'll need some way to convert a value back to a string.

- Implement a (possibly private) method of Publication to convert the resulting int back into a string for Publication::to_string. Calling these methods "genre_to_string" et. al. may offer a certain consistency, e.g., “string genre_to_string(Genre genre);”.
- Implement a “helper function” of the same name that exists in global space. This is u-g-l-y from an object-oriented programming perspective, though – we don’t leave functions just floating around in global space (it’s not even *possible* in Java!).
- Use a vector of strings, e.g., given “enum Color {red, green, blue}; vector<string> color_to_string = {“red”, “green”, “blue”};”, you could convert a Color variable to a string using, e.g., “Color color = Color::red; cout << color_to_string[color];”

Option #2: Or, you can make them each a class with an intrinsic to_string method of their own (called from the publication's to_string method). This is the most object-oriented solution.

Option #3: Or, to stretch the definition a bit, you could just make them a string, either directly or via typedef, which we haven't covered yet - see http://www.cplusplus.com/doc/tutorial/other_data_types/. The downside is that you'll need some significant data validation code to ensure that *every string in every object is always correct*. If you just let users type whatever they want, you'll have a disaster instead of a database.

Q. The UML shows a Main class with a main() method. How is this implemented in C++?

Remember that the **UML is not specific to C++** - a UML class diagram can be implemented in any language. Since some languages (looking at you, Java) require that main() be part of a class, a UML class diagram *may* show the design in this way.

As discussed in previous assignments, however, C++ is unable to specify main() as part of a class, so it **must** be implemented as a stand-alone function, e.g., int main( ) { }.

Part of understanding UML is to be able to translate from constructs valid in the UML to constructs that a C++ compiler can handle. This is one such case.

Q. What type do I return if the UML specification of a method doesn't list a type at all?

We always implement the return type of a method specified in the UML with no return type as a void in C++.

For a constructor, which of course is not a method, we would never specify a return type in C++ - the return type is an object, and is implicit.

Q. How do I implement main.h?

Since main.cpp just contains the main() function, which is invoked when you run the program, you don’t need and shouldn’t create a main.h file.

Q. How do I deal with check_in and check_out methods in both the Publication and Library classes? Don't the conflict with each other?

No. An object instanced from the Publication class represents an individual book, magazine, DVD, etc. Think of one of the textbooks for this class. When you call check_out on this object, you are checking out the associated media - thus you only need to know the patron_name and patron_phone of the person who will temporarily take possession of the associated artifact.

An object instanced from the Library class represents a *collection* of books, magazines, DVDs, etc. Think of the UTA library, which contains a large collection of media. When you call check_out on this object, you must also specify which specific publication to check out from the library.

Similarly, calling check_in on a publication object requires no parameters - it's checking in the publication associated with that object. But calling check_in on the library object requires that you specify as a parameter which publication in the library is being checked back in.

If this worries you, consider using different names for each class' methods. That's not necessary, but it's certainly permissible.