Assignment #4: Polymorphism and Design Patterns

Due Date 1: Wednesday, July 8, 2020, 5:00 pm
Due Date 2: Wednesday, July 15, 2020, 5:00 pm
Online Quiz: Friday, July 17, 2020, 5:00 pm

Topics that must have been completed before starting Due Date 1:

1. Error Handling with Exceptions
2. STL `<vector>` and `<map>`
3. Object-oriented programming: Inheritance
4. UML
5. Design patterns: introduction
6. Design patterns: Iterator, Decorator, Observer

Learning objectives:

- C++: exceptions
- C++: polymorphism
- Design patterns: Iterator, Decorator, Observer
- Introduction to X11 graphics, which will likely be useful for the final project.

- **Test suites for questions 1, 2, and 3 are due on Due Date 1; the remaining questions are due on Due Date 2. You must submit the online quiz on Learn by the Quiz date.**
- You must use the C++ I/O streaming and memory management facilities on this assignment. Marmoset will be programmed to **reject** submissions that use C-style I/O or memory management.
- You may include the headers `<iostream>`, `<fstream>`, `<sstream>`, `<iomanip>`, `<string>`, `<utility>`, `<stdexcept>`, `<vector>` and `<map>`.
- For this assignment, you are **not allowed** to use the array (i.e., `[]`) forms of `new` and `delete`. Further, the `CXXFLAGS` variable in your `Makefile` **must** include the flag `-Werror=vla`.
- Each question on this assignment asks you to write a C++ program, and the programs you write on this assignment each span multiple files. For this reason, we **strongly** recommend that you develop your solution for each question in a separate directory. Just remember that, for each question, you should be *in* that directory when you create your ZIP file, so that it does not contain any extra directory structure.
- A proper object-oriented design is achieved through the use of virtual methods that allow you to perform the required operations, without ever knowing exactly what kind of object you have. Therefore any attempt to "query" the run-time types of objects, and then make decisions based on the discovered types, will not earn marks. Additionally, since this assignment focuses on object-oriented design, not following the requirements will cause you to lose up to 100% of Marmoset correctness marks.
- You are required to submit a `Makefile` along with every code submission. Marmoset will use this `Makefile` to build your submission.
- Question 4, the bonus question, asks you to work with XWindows graphics. (This may also be something you need for the final project, depending upon which choices are available.) **Well before starting that question, make sure you are able to use graphical applications from your Unix session.** If you are using

Linux you should be fine. **If making an SSH connection to a campus machine, be sure to pass the `-Y` option and turn on X-forwarding.**

- **Note for Windows users:** If you are using Windows and putty, you should download and run an X server such as XMing, and be sure that PuTTY is configured to forward X connections. Alert course staff immediately if you are unable to set up your X connection (e.g. if you can't run `xeyes`).
- If working on your own machine, make sure you have the necessary libraries to compile graphics. Try executing the following from within the `graphics-demo` directory:

  ```
  g++14 window.cc graphicsdemo.cc -o graphicsdemo -lX11
  ```

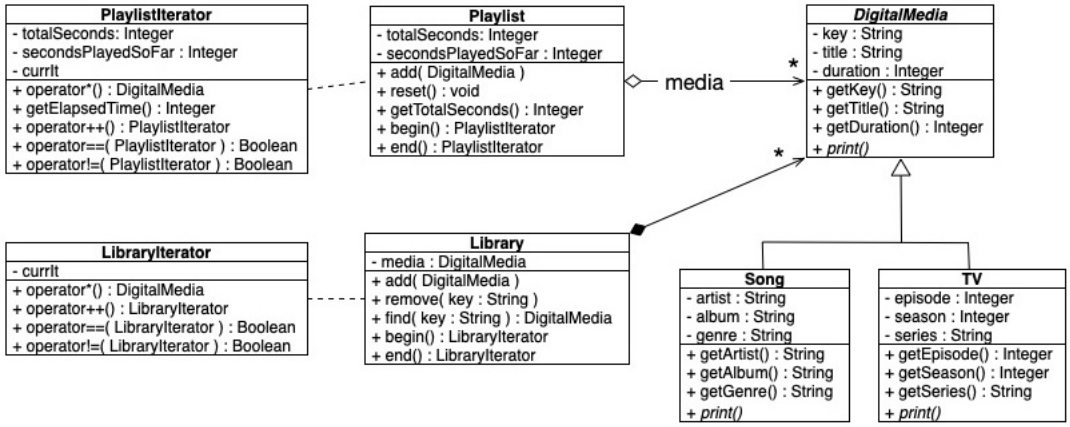
  (Note: this is a dash followed by lower case L followed by X and then one one.) Run the program

  ```
  ./graphicsdemo
  ```
- **Note for Mac OS users:** On machines running Mac OS you will need to install XQuartz. (See `https://www.xquartz.org/`.) Once installed, you must restart your machine. You will also need to explicitly tell g++ where the `X11` library is located. If the above compilation command does not work, browse through your Mac's file system looking for a directory `X11` that contains directories `lib` and `include`. You must then specify the `lib` directory using the `-L` option and the `include` directory using the `-I` (uppercase i) option. For most installations, the following should work (from within the `graphics-demo` directory):

  ```
  g++14 window.cc graphicsdemo.cc -o graphicsdemo -lX11 -L/usr/X11/lib -I/usr/X11/include
  ```

Question 1

This question simulates a simplified digital music player that has an underlying library of digital media (either songs or television shows) and a limited number of media play lists that can be set up from the library contents. In particular, we'll use the Iterator design pattern to traverse the library and the play list objects. The resulting UML class model looks like the following, though you can change your private data field declaration names, and add private methods as desired.



(You have been provided with some starter code, a test harness, a sample input file, and a sample executable to help you. Make sure that you read through the test harness carefully to understand what it does.)

The `Song` and `TV` classes are concrete subclasses of the abstract base class `DigitalMedia`. They just contain data, with the appropriate accessors and no mutators. (One simplifying assumption that you may make is that the keys are alphanumeric sequences.) They implement the pure virtual method `print`, which is used to help the iterators output the object information. Use the provided `util.h` and `util.cc` to simplify your implementations of the associated input operators, though you will have to add some code to `util.cc` to raise the necessary exceptions and the necessary handler code to `main.cc` in the indicated locations.

- The `TV` constructor sets the key, title, duration (in seconds), episode, season, and series information. See the provided starter file for the list of exceptions raised, and under what order and condition they are raised.

- `operator<<` for `TV` outputs the object's information as `(key, "series" S<season>E<episode> "title", duration)` where each string that might have whitespace within it (i.e. title, series) is surrounded with double-quotation marks. The season number and episode number have a default width of 2, and must start with '0' if their value is less than 10.
- `operator>>` for `TV` reads in information in the following format `key\ntitle\nduration\nseason\nepisode\nseries\n`. It raises `std::runtime_error` if it fails to obtain any of the elements, if the strings are empty, or if the duration, season, or episode cannot be successfully converted to an `int`.
- The `Song` constructor sets the key, title, duration (in seconds), artist, album, and genre information. See the provided starter file for the list of exceptions raised, and under what order and condition they are raised.
- `operator<<` for `Song` outputs the object's information as `(key, "title", "album", "artist", duration, "genre")` where each string that might have whitespace within it (i.e. title, album, artist, genre) is surrounded with double-quotation marks.
- `operator>>` for `Song` reads in information in the following format `key\ntitle\nduration\nartist\nalbum\ngenre\n`. It raises `std::runtime_error` if it fails to obtain any of the elements, if the strings (except for genre or album) are empty, or if the duration cannot be successfully converted to an `int`

The `Library` holds the pointers to the objects of the concrete `DigitalMedia` subclasses, `Song` and `TV`. When traversed, the items are output in lexicographical order by their key values i.e. standard `std::string` sort order. The keys must be unique. (While this can be implemented using `std::vector`, `std::map` will makes things considerably simpler, so you want to read ahead a bit.)

- `Library::add` adds the specified object pointer to the library, indexed by its key. It will raise the exception object `std::logic_error`, containing the message `"key KKKK already exists in library"` if the key `KKKK` is a duplicate.
- `Library::find` returns the pointer to the object with the specified key, or `nullptr`.
- `Library::remove` removes an item by first searching for the item with the specified key value, and then removing it if it exists. If it doesn't exist, nothing happens and no error messages are produced.
- `operator<<` outputs the string `Library:\n`, followed by each item in the library in key order. Each item is preceded by a tab character, `'\t'`, and terminated by a newline.
- `operator>>` adds to the existing library by reading a sequence of digital media objects. Each set of input starts with either an `'s'` to denote a `Song`, or a `'t'` to denote a `TV` object. The information is then read in according to the specified input format for the `Song` or `TV` classes. (The simplest approach is to use the defined input operators to read them into temporary objects whose contents are over-written, and then use copy operations.) If the specified type is neither an `'s'` nor a `'t'`, the exception `std::runtime_error` is raised with the message `"invalid media type"`.

A `Playlist` holds a collection of `DigitalMedia` pointers to the concrete subclasses. Items are kept in the order in which they are added to the play list. Play lists may share items. `Playlist::remove` removes an item by first searching for the item with the specified address, and then removing it if it exists. If it doesn't exist, nothing happens and no error messages are produced. A play list knows the sum of the durations for each of its items, and how much time has elapsed when it is being "played". Playing the `Playlist` is simulated by iterating over it. When the `PlaylistIterator` is created, it starts at the first item in the list, and the amount of elapsed time is 0 seconds. When the iterator is moved via `operator++`, the duration of the current item is added to the elapsed time before the iterator is moved to the subsequent item. When the end of the sequence is reached, the iterator stops and the elapsed time equals the sum of all of the item durations. The amount of elapsed time can be reset through `Playlist::reset`, though a new iterator will have to be created to start at the beginning. Any change to the play list structure invalidates the iterator. (See the provided test harness.)

The test harness is not intended to be particularly robust, so do not write test cases for it. It has an instance of the `Library` and an array of 5 `Playlist` objects. It handles the following commands read from standard input:

Command	Explanation
`q`	Quits the program. Same as reaching EOF.
`p m`	Prints the contents of the media library.
`p p` *idx*	Prints the contents of the playlist at index *idx*.
`r` *key*	Removes the item with the specified key from the media item. Silently does nothing if such an item doesn't exist. Also removes the item from all play lists.
`>`	If has a current playlist and haven't reached the end, "play" the current media item by printing its information, and how much time in the playlist has elapsed out of the total time. Advances the iterator.
`z`	Resets the current playlist iterator to start over from the beginning. Also resets the elapsed time to 0.
`f m` *key*	Looks for the item with the specified key in the media library.
`f p` *idx*	Sets the current playlist to the playlist at index *idx* and sets the play iterator to start at the beginning.
`i` *filename*	Inserts the contents of file *filename* into the media library. *filename* consists of a sequence of `a m` commands as described below.
`a m t\n`	Adds a TV show to a media library. The information that follows must follow the format as specified for the `operator>>` for a TV show.
`a m s\n`	Adds a song to a media library. The information that follows must follow the format as specified for the `operator>>` for a song.
`a p` *idx key*	Adds the media item with key *key* to the playlist at index *idx*. Silently fails if *idx* not in range `[0, MAX_NUM_PLAYLISTS-1]` or media library doesn't contain an item with key *key*.

An example run of the program appears below. The information marked in blue (e.g. `i lib01.txt`, `p m`) is user input. The remaining text and numbers are the program output:

```
$ ./a4q1
i lib01.txt
p m
Library:
        (S01, "Great beat and you can dance to it", "I'd give it a 10!", "Too cool", 183, "Eclectic")
        (S02, "Is jive still cool?", "Trying to be funky", "Cooler than thou", 167, "dance")
        (S99, "dance dance dance", "album 3", "base10", 87, "surf punk")
        (TV01, "Amazing New Show" S01E01 "Somewhere far away, and long ago", 2580)
        (TV02, "Amazing New Show" S02E01 "Further away, and longer ago", 2585)
        (TV99, "Amazing newish show" S10E03 "Somewhere not so far away and less long ago", 2587)
pp0
Playlist[0]:
Total: 0 seconds
ap0 S01
a p 0 S02
a p 0 S98
a p 0 S99 pp0
Playlist[0]:
        000: (S01, "Great beat and you can dance to it", "I'd give it a 10!", "Too cool", 183, "Eclectic")
        001: (S02, "Is jive still cool?", "Trying to be funky", "Cooler than thou", 167, "dance")
        002: (S99, "dance dance dance", "album 3", "base10", 87, "surf punk")
Total: 437 seconds
ap1 TV01 ap1 TV02 ap1 TV99 pp1
Playlist[1]:
        000: (TV01, "Amazing New Show" S01E01 "Somewhere far away, and long ago", 2580)
        001: (TV02, "Amazing New Show" S02E01 "Further away, and longer ago", 2585)
        002: (TV99, "Amazing newish show" S10E03 "Somewhere not so far away and less long ago", 2587)
Total: 7752 seconds
fp0
>
(S01, "Great beat and you can dance to it", "I'd give it a 10!", "Too cool", 183, "Eclectic") 183/437
>
(S02, "Is jive still cool?", "Trying to be funky", "Cooler than thou", 167, "dance") 350/437
>
(S99, "dance dance dance", "album 3", "base10", 87, "surf punk") 437/437
>
```

```
>
z
>
(S01, "Great beat and you can dance to it", "I'd give it a 10!", "Too cool", 183, "Eclectic") 183/437
fp2
>
q
$
```

Your program must be clearly written, must use exceptions as specified, must follow the Iterator design pattern, and must not leak memory.

a) **Due on Due Date 1:** Design a test suite for this program. Submit a file called `a4q1a.zip` that contains the test suite you designed, called `suiteq1.txt`, and all of the `.in` and `.out` files.

b) **Due on Due Date 2**: Write the program in C++. Save your solution in a file named `a4q1b.zip`. It must contain a `Makefile` that creates an executable name `a4q1` when the command `make` is given.

Question 2

In this problem you will have a chance to implement the Decorator pattern. The goal is to write an extensible text processing package. You will be provided with two fully-implemented classes:

- `TextProcessor` (`textprocessor.{h,cc}`): abstract base class that defines the interface to the text processor.
- `Echo` (`echo.{h,cc}`): concrete implementation of `TextProcessor`, which provides default behaviour: it echoes the words in its input stream, one token at a time.

You will also be provided with a partially-implemented mainline program for testing your text processor (`main.cc`).
You are not permitted to modify the two given classes in any way.
You must provide the following functionalities that can be added to the default behaviour of `Echo` via decorators:

- `dropfirst n` Drop the first `n` characters of each word. If `n` is greater than or equal to the length of some word, that word is eliminated.
- `doublewords` Double up all words in the string.
- `allcaps` All letters in the string are presented in uppercase. Other characters remain unchanged.
- `count c` The first occurrence of the character `c` in the string is replaced with 1. The second is replaced with 2, ... the tenth is replaced with 10, and so on.

These functionalities can be composed in any combination of ways to create a variety of custom text processors.
The mainline interpreter loop works as follows:

- You issue a command of the form `source-file list-of-decorators`. If `source-file` is `stdin`, then input should be taken from `cin`.
- The program constructs a custom text processor from `list-of-decorators` and applies the text processor to the words in `source-file`, printing the resulting words, one per line.
- You may then issue another command. An end-of-file signal ends the interpreter loop.

An example interaction follows (assume `sample.txt` contains `Hello World`): The information marked in blue (e.g. `sample.txt doublewords dropfirst 2 count 1`, `sample.txt allcaps`) is user input. The remaining text and numbers are the program output:

```
sample.txt doublewords dropfirst 2 count 1
1 12o
2 34o
3 r5d
4 r6d
sample.txt allcaps
1 HELLO
2 WORLD
```

The numbers at the beginning are word numbers (word 1, word 2, etc.), and are supplied by the test harness. You do not need to generate them. Your program must be clearly written, must follow the Decorator design pattern, and must not leak memory.

a) **Due on Due Date 1:** Design a test suite for this program. Submit a file called `a4q2a.zip` that contains the test suite you designed, called `suiteq2.txt`, and all of the `.in` and `.out` files.

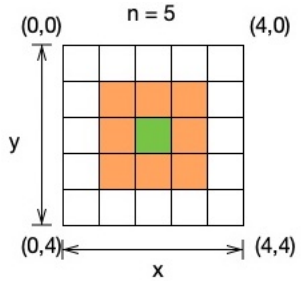
b) **Due on Due Date 2**: Write the program in C++. Save your solution in a file named `a4q2b.zip`. It must contain a `Makefile` that creates an executable name `a4q2` when the command `make` is given.

Question 3

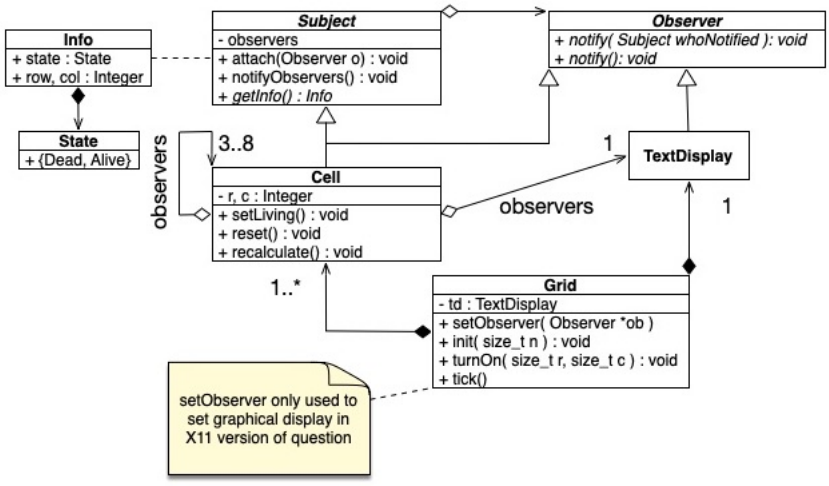
In this problem, you will use C++ classes to implement Conway's Game of Life (`https://en.wikipedia.org/wiki/Conways_Game_of_Life`). An instance of Conway's Game of Life consists of an $n \times n$-grid of cells, each of which can be either alive or dead. When the game begins, we specify an initial configuration of living and dead cells. The game then moves through successive generations, in which cells can either die, come to life, or stay the same, according to the following rules:

- a living cell with fewer than two live neighbours or more than three live neighbours dies,
- a living cell with either two or three live neighbours continues to live, and
- a dead cell with exactly three live neighbours comes to life; otherwise, it remains dead.

The *neighbours* of an inner cell are the eight cells that immediately surround it (cells on the grid edges and corners naturally have fewer neighbours). An example is shown in the following figure, where an inner cell is shown in green, and its surrounding neighbours are shown in orange. Note that each has a position (x, y), where x represents the horizontal direction, and y represents the vertical direction. The top-left corner of the grid has position $(0, 0)$.



To implement the game, you will use the following classes organized to follow the Observer design pattern:



(You have been provided with some starter code, a test harness, a sample input file, and a sample executable to help you. Make sure that you read through the test harness carefully to understand what it does.)

- `Subject` is the abstract base class that defines the subject interface and operations (see provided `subject.h`). All subjects inherit the `attach` and `notifyObservers` methods. An observer uses the `getInfo` method to obtain new information about the subject. (In the game context, it will only ever be called on `Cell` objects.)

- `Observer` is the abstract base class that defines the observer interface and operations (see provided `observer.h` and `observer.cc`). There are two versions of the `notify` method. The version without any parameters is called solely by the `Grid` object at the start of each game's "tick" to tell each cell to notify its neighbours of its status. The version that takes the subject as a parameter is called by either a `Cell` object as part of its status notification, or by the `Grid` on a cell's behalf when either `Grid::turnOn` has been called, or the cell states have changed and the display needs to be updated.

- `Cell` implements a single cell in the grid (see provided `cell.h`). It is both a *Subject* and an *Observer* since each cell acts as an observer to its neighbours and must notify its neighbours of its state changes. Each cell needs to know its position in the grid and its current state. The information as to who its neighbours are is stored in the inherited observers container, along with the display.

- `Grid` implements a two-dimensional grid of cells (see provided `grid.h`). It creates a new `TextDisplay` every time `Grid::init` is called, and a new collection of `Cell` objects. (You must avoid directly allocating the cells on the heap. Read the `<vector>` documentation carefully to determine the approach.) It is also responsible for attaching observers to the cells after they have been created and before the game continues. When `Grid::turnOn` is called, the specified cell state is set to `Alive` and an update of the display must be triggered. `Grid::tick` is used to calculate the state for the next generation of the game.

- `TextDisplay` is an observer on each `Cell` object (see provided `textdisplay.h`). It stores a set of characters that represents the current state of each cell object, where `'X'` represents an alive cell, and `'_'` (underscore) represents a dead cell. In order to print the grid, `operator<<` is overloaded for the `Grid` class and invokes the overloaded `operator<<` for the `TextDisplay` class.

- `State` is an *enumerated class* in C++ (see provided `state.h` and `info.h`). This is a more type-safe version of an *enumeration*. (See *scoped enumerations* at `https://en.cppreference.com/w/cpp/language/enum` for documentation on their use. `https://www.geeksforgeeks.org/enum-classes-in-c-and-their-advantage-ov` provides a good explanation of why you'd want to use them.)

Note: you are not allowed to change the public interface of these classes (i.e., you may not add public fields or methods), but you may add private fields or methods if you want.

An iteration of the game (which corresponds to one invocation of `Grid::tick`) happens in two steps, as follows:

1. The grid calls each cell's overridden `Cell::notify()` method.
   - When the cell's `notify` method is called, the cell, if alive, tells all of its neighbours that it is alive by calling `Subject::notifyAll()`. (`Subject::notifyAll()` notifies each neighbour, as well as the `TextDisplay`, by calling the virtual overridden `Observer::notify(Subject & whoNotified)` method for the concrete subclass.)
   - When a cell's overridden `Cell::notify(Subject & whoNotified)` method is called by one of its neighbours, it updates its record of how many of its neighbours are alive.
2. After the grid has called each cell's `Cell::notify()` method, the grid calls each cell's `recalculate` method.
   - When a cell's `recalculate` method is called, it calculates its alive or dead status for the next round, based on the number of messages it received from living neighbours.
   - The grid must then tell the text display to update its information by notifying it on each cell's behalf using the overridden `TextDisplay::notify(Subject & whoNotified)` method.

When you run your program, it will listen on standard input for commands. Your program must accept the following commands:

Command	Explanation
`new n`	Creates a new $n \times n$ grid, where $n \geq 1$, by invoking `Grid::init`. If there was already an active grid, that grid is destroyed and replaced with the new one. (Note that we are using the defined type `size_t` since it is an unsigned integer. For more information, see `https://en.cppreference.com/w/cpp/types/size_t`.)
`init`	Enters *initialization mode*. Subsequently read pairs of integers `x y` and sets the cell at coordinates (x, y) as alive by calling `Cell::setLiving`. The coordinates `-1 -1` end initialization mode. It is possible to enter initialization mode more than once, even while the simulation is running.
`step`	Runs one tick of the simulation i.e. transforms the grid into the immediately succeeding generation by calling `Grid::tick`.
`steps n`	Runs n steps of the simulation.
`print`	Prints the grid.

The program quits when the input stream is exhausted.

An example interaction follows. The information marked in blue (e.g. `new 5`, `init`) is user input. The remaining text is the program output:

```
new 5
init
2 1
2 2
2 3
-1 -1
print
_____
__X__
__X__
__X__
_____
step
print
_____
_____
_XXX_
_____
_____
```

Note: Your program should be well documented and employ proper programming style. It should not leak memory. Markers will be checking for these things.

a) **Due on Due Date 1:** Design a test suite for this program. Submit a file called `a4q3a.zip` that contains the test suite you designed, called `suiteq3.txt`, and all of the `.in` and `.out` files.

b) **Due on Due Date 2**: Write the program in C++. Save your solution in a file named `a4q3b.zip`. It must contain a `Makefile` that creates an executable name `a4q3` when the command `make` is given.

Question 4

*** BONUS *** This question is worth 5% as a bonus i.e. if the assignment is out of 100 marks, and you obtain full marks, your maximum assignment grade is 105 marks.

Note: there are no Marmoset tests for this problem since this problem will be entirely hand-marked.

In this problem, you will adapt your solution from question 3 to produce a graphical display in addition to the existing text display. You are provided with a class `Xwindow` (see files `window.h` and `window.cc`) to handle the mechanics of getting graphics to display. Declaring an `Xwindow` object (e.g. `Xwindow xw;`) causes a window to appear. When the object goes out of scope, the window will disappear (thanks to the destructor). The class supports methods for drawing rectangles and printing text in five different colours. For this assignment, you should only need white and black rectangles where black represents a live cell and white represents a dead cell.

To make your solution graphical, you should carry out the following tasks:

- Implement a `GraphicsDisplay` class as a concrete subclass of the abstract base class `Observer`.
- `Grid` creates a `GraphicsDisplay` object and registers it as an observer of each cell object **in addition to the existing** `TextDisplay` **object**.
- The class `GraphicsDisplay` is responsible for mapping the row and column numbers of a given cell object to the corresponding coordinates of the squares in the window.
- Your `GraphicsDisplay` class should have a composition relationship with Xwindow. This implies that your constructor for `GraphicsDisplay` should also create a `Xwindow` object, which is a member of the `GraphicsDisplay` class. Each time a new grid is created, you should create a new `GraphicsDisplay` to set up an appropriate window for the grid size.
- Your cell objects should not have to change at all.

The window you create should be of size 500×500, which is the default for the `Xwindow` class. The larger the grid you create, the smaller the individual squares will be. (If the size of the grid doesn't divide evenly, **any reasonable solution is acceptable** since the program will be marked by hand.)

Note: to compile this program, you need to pass the option `-lX11` to the compiler *at link time* and it must be the last option for the linking command. For example:

```
g++ -std=c++14 *.o -o a4q4 -lX11
```

This option is not relevant during compilation, so it should not be put in your `CXXFLAGS` variable. You should only use it during the linking stage, i.e. the command that builds your final executable.

Note: Your program should be well documented and employ proper programming style. It should not leak memory (note, however, that the given `Xwindow` class leaks a small amount of memory; this is a known issue with X11). Markers will be checking for these things.

a) **Due on Due Date 1:** Not applicable.

b) **Due on Due Date 2**: Write the program in C++. Save your solution in a file named `a4q4b.zip`. It must contain a `Makefile` that creates an executable name `a4q4` when the command `make` is given.