Assignment #4: Design Patterns

Due Date 1: Friday, November 8, 2024, 5:00 pm EST (30% marks)
Due Date 2: Friday, November 15, 2024, 5:00 pm EST (70% marks)

Learning objectives:

- `std::vector`
- Decorator design pattern
- Observer design pattern

- **Questions 1a and 2a are due on Due Date 1; Question 1b, 2b, and 3b are due on Due Date 2**

- See A1 for notes on: test suites, undefined behaviour, test partners, using C++-style I/O and memory management, hand-marking,
- In all cases, your test suites should be testing the functionality that you are responsible for. **Do not submit test cases whose sole purpose is to verify the behaviour of the test harness.** There is no value in that, and there are no marks allocated for that.
- There will be a hand-marking component in this assignment. In addition to the usual style requirements, you will be required to implement a design pattern. Your solution must follow the necessary pattern form. **Note that we expect you to separate your classes into appropriate header and implementation files for each class.**
- You must use the C++ I/O streaming and memory management facilities on this assignment. Moreover, the only standard headers you may import are: `<iostream>`, `<fstream>`, `<sstream>`, `<string>`, `<utility>`, `<compare>`, `<stdexcept>` `<memory>`, and `<vector>`. Marmoset will be programmed to **reject** submissions that violate these restrictions. **You may use any of the functions within those libraries.**
- 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`.
- **You are not permitted to ask any public questions on Piazza about what the programs that make up the assignment are supposed to do.** A major part of this assignment involves designing test cases, and questions that ask what the programs should do in one case or another will give away potential test cases to the rest of the class. Instead, we will provide compiled executables, suitable for running on `linux.student.cs`, that you can use to check intended behaviour. Questions found in violation of this rule will be marked private or deleted; repeat offences could be subject to discipline.
- **Note:** Question 3 asks you to work with XWindows graphics. Well before starting that question, make sure you are able to use graphical applications from your Linux session. If you are using Linux locally, you should be fine (if making an `ssh` connection to a campus machine, be sure to pass the `-Y` option). If you are using an SSH connection over Windows, you should download and run an X server such as XMing, and be sure that your SSH connection is configured to forward X connections. For Mac users, the advice is similar, but look for the XQuartz server. Alert course staff immediately if you are unable to set up your X connection (e.g. if you can't run `xeyes`).

  Also (if working on your own machine) make sure you have the necessary libraries to compile graphics. Try executing the following:

  ```
  g++20 window.cc graphicsdemo.cc -o graphicsdemo -lX11
  ```

  (That last flag is "dash little-ell big-ex one one".)

1. **(40% of DD1; 20% of DD2)** For this question: use C++20 imports. Compile the system headers with `g++20h` and compile your files with `g++20`.

   In this question, you will implement a class that represents a mathematical set for integers using `std::vector`. The class definition and its associated method and function signatures have been given to you in the file `IntSet.cc` in your repository.

   You have also been given a test harness in `a4q1.cc` as the `main` function. **Do not change it.** Make sure you read and understand this code, as you will need to know what it does in order to structure your test suite. It isn't very robust and does no error-checking. You are allowed to assume that the commands it receives are valid, so *do not write test cases for it*!

   Implement the described `IntSet` methods and functions. Your implementation must satisfy the following requirements:

   - A set cannot contain duplicate items. Attempting to add an item already in the set does not change the set nor does it produce an error (see `add` and `operator&`).
   - Set intersection (`operator&`) returns a new set containing only the elements that are in both sets.
   - Set union (`operator|`) returns a new set containing all of the elements from both sets, without any duplicates.
   - Two sets A and B are equal (`operator==`) if and only if no element of one is not an element of the other. More precisely: $(\nexists x s.t. x \in A \wedge x \notin B) \wedge (\nexists x s.t. x \in B \wedge x \notin A)$.
   - `i1.isSubset( const IntSet & i2 )` returns true if and only if every element in `i2` is an element of the set `i1`: otherwise it returns false.
   - `i.contains( int e )` returns true if `e` is an element of the set `i`: otherwise it returns false.
   - After `i.remove( int e )` is called, `e` must not be in the set `i`.
   - The input operator (`operator>>`) adds the read-in integers to the set until a non-integer value is read. The first non-integer character is thrown away.
   - The output operator (`operator<<`) first prints a left parenthesis '`(`', then each integer in added order, delimited by a space, and ends with a right parenthesis '`)`'. For example the set containing 3, 5, and 2 is printed as `( 3 5 2 )`. An empty set is printed as `( )`.

   (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++. Submit your solution files `IntSet.cc`, `IntSet-impl.cc`, and `a4q1.cc` in a ZIP file named `a4q1b.zip`.

2. **(60% of DD1; 40% of DD2)** For this question: use `#includes`. Compile your program with `g++20`.

   In this question you will be building a fun ASCII art drawing and animation program! You'll start with a blank canvas (files `studio.{h,cc}`), and will draw rectangles consisting of characters of your choosing on that canvas (remember that any point is a 1x1 rectangle, so you can draw anything!), and choose how your drawing will evolve with time. The rectangles may be larger than the canvas, and may disappear from the canvas by moving outside the canvas boundaries.

   The key to the functioning of this program is the *Decorator* design pattern. A blank canvas (files `blank.{h,cc}`) will respond with a blank space to any query about the contents of a cell at a particular row or column. However, it can be decorated by rectangles, which will respond in their own way to the same query, or will pass it along to the ASCII art element behind them. (See the provided file `boxdimensions.pdf` for an explanation of how the rectangle dimensions are specified and drawn.)

   Moreover, the key to the animation aspect of this program is that the response to a query for a cell can be time-varying. So a query to a cell actually involves three parameters: the row, the column, and the tick count. The *tick count* is an abstraction of a clock that represents the number of rendering events that have occurred since the last time the animation was reset. There's an internal counter representing how many "ticks" have elapsed so far, and each call to "render" advances the tick counter (clock) by 1. "reset" puts it back to 0.

   You are to support the following kinds of ASCII art elements:

- *filled box:* A simple solid rectangle, specified by top and bottom row (inclusive) and left and right column (inclusive). Does not move.
- *blinking box:* A box that is displayed when the tick count is even, and is transparent otherwise.
- *moving box:* A box that moves one position (either up, down, left, or right) with each tick.
- *mask box:* A box that is invisible unless there is a visible art element beneath it. In that case, the part of the mask that lies above the art below becomes visible and covers that art. Does not move.

The provided starter code comes with a test harness `main.cc` that supports the following commands:

- `render` — causes the current artwork to be displayed, with a frame around it, so that the boundaries are clear (see the provided executable for details). Counts as a tick.
- `animate n` — renders the artwork `n` times, where `n` must be ≥ 0.
- `reset` — resets the tick count to 0.
- `filledbox t b l r c` — adds a filled box with given top, bottom, left, right boundaries, filled with the character `c` (assumed to be printable ASCII in the range 33–127). Invalid if top exceeds bottom or left exceeds right.
- `blinkingbox t b l r c` — adds a blinking box, with parameters understood as above.
- `movingbox t b l r c dir` — adds a moving box, with first five parameters understood as above and a direction `dir`, which can be one of `u d l r`. **Clarification:** These parameters are understood to be the position of the box when the tick count is 0. If the tick count is not 0 when the moving box is placed, it will show up shifted by a number of positions equal to the current tick count.
- `maskbox t b l r c` — adds a mask, with parameters understood as above.

Note: Some implementation notes for this problem can be found in the provided `README.txt`. Be sure to read and follow it!

Important: As the point of this problem is to use the Decorator design pattern, if your solution is found in hand-marking to not employ the Decorator design pattern, your correctness marks from Marmoset will be revoked.

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:** Submit your test suite (`suiteq2.txt` and all necessary files) in the file `a4q2a.zip`.

b) **Due on Due Date 2:** Submit your solution as `a4q2b.zip`. You must include a `Makefile`, such that issuing the command `make` will build your program. The executable should be called `a4q2`. **Do not submit your Q2 solution for Q3.**

3. **(0% of DD1; 40% of DD2)** For this question: use `#includes`. Compile your program with `g++20`.

This question extends the previous question, in which you wrote an ASCII art animator program. In that problem, you had a `Studio` object that managed the `tick` count, and was responsible for rendering your artwork to an output stream (`std::cout`). In this problem, we'll take that second responsibility away from the `Studio` class by employing the Observer design pattern. Your `Studio` class will become a concrete "subject" class in this new model.

If you wish to actually see your artwork, you will need to create an observer object and attach it to the subject. But why stop at one? You could, in fact, create several observers! (And why should they all be necessarily text-based? More on that in a bit.) A bunch of observers all observing and rendering the same text is boring. But who says they all have to be watching the same part of your artwork? And who says your artwork has to be limited to just a 10x10 grid?

No, in this problem, your art has rows and columns spanning **the entire range of 32-bit integers!** But each of your observers, of course, will only see a much smaller portion of it.

In this problem, you will restructure your solution so that, in addition to following the Decorator design pattern, it also conforms to the Observer design pattern. As well, incorporate the other changes described above.

The test harness for this problem includes all of the commands from the previous problem. Those whose meanings have changed are described below, along with the new commands specific to this problem:

- `render` — causes all observers to render their images. Images should be displayed in the order the observers were registered with the subject.
- `addtext t b l r` — adds a text-based observer watching a portion of the image with the given top, bottom, left, right boundaries. Invalid if top exceeds bottom or left exceeds right. This observer, when notified, will display its view to `std::cout`.
- `addgraphics t b l r` — adds a graphical observer, with parameters understood as above.

For the graphical observers, you will use the `Xwindow` class that has been provided to you. **Make sure you have a working XWindows server running on your computer, such that you can run graphical applications over an SSH connection. Test this EARLY.** Your windows will be of size $10r$ by $10c$, where r is the number of rows being observed, and c is the number of columns being observed. Represent the rendered objects by colour-coded 10x10 pixel squares:

- A red square represents any lower-case letter.
- A green square represents any upper-case letter.
- A blue square represents any digit.
- A black square represents any other printable character.
- A white square represents “no” character.

If you have difficulty distinguishing colours, such that this portion of the assignment would be difficult for you to complete, then please contact an ISA to request an alternative means of completing the graphical display.

Note: Some implementation notes for this problem can be found in the provided `README.txt`. Be sure to read and follow it!

Important: As the point of this problem is to use the Decorator and Observer design patterns, if your solution is found in hand-marking to not employ these patterns, your correctness marks from Marmoset will be revoked.

Something to think about: In the Observer pattern, the subject does not own its observers, and is not responsible for deleting them. Therefore, you must think about who is going to be responsible for the observers that you create and how they should eventually be deleted.

Something else to think about: You will probably find that your graphical display, when running anywhere other than on campus, will be quite slow. Think about how you might optimize the performance of your graphics observer, so that it renders more quickly. This is not an assignment requirement, but something to think about for your own interest.

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:** Nothing. A test suite is not required for this problem.

b) **Due on Due Date 2:** Submit your solution in the file `a4q3b.zip`. You must include a `Makefile`, such that issuing the command `make` will build your program. The executable should be called `a4q3`. **Do not submit your Q2 solution for Q3.**