Assignment 3

COMP SCI 2ME3 and SFWR ENG 2AA4

March 10, 2019

Assigned: February 25, 2019

Part 1 (Spec): March 1, 2019

Receive Full Spec: March 6, 2019

Part 2 (Implement and Test): March 22, 2019

Last Revised: March 10, 2019

All submissions are made through git, using your own repo located at:

`https://gitlab.cas.mcmaster.ca/se2aa4_cs2me3_assignments_2018/[macid].git`

where `[macid]` should be replaced with your actual macid. The time for all deadlines is 11:59 pm.

1 Introduction

The purpose of this software design exercise is to design and implement a portion of the specification for the game of Forty Thieves (`https://greenfelt.net/fortythieves`). For Part 1, you are given a partial specification and asked to fill in the specification of the missing semantics. Once the specification is complete, you will implement it for Part 2. To have a common interface across the class, to facilitate unit testing by the TAs, you will implement the instructor provided specification, rather than your own.

The game of Forty Thieves uses two standard 52-card decks, for a total of 104 cards. The set-up of the board and the rules are as follows:

- For the initial board, a tableau of 10 stacks of 4 cards are created, with the remaining cards still in the deck. All of the tableau card stacks are face up and visible.

- Above the tableau are 8, initially empty, foundation stacks. There are two foundation stacks for each suit.
- Cards can only be moved from the top of each tableau stack. You may place any card in an empty tableau space.
- The tableau cards can be moved from the top of one tableau to the top of another, by building down by suit, from King to Ace.
- The foundation slots are built up by suit in the opposite order, from Ace to King.
- Cards may be dealt one at a time from the remaining deck to the waste stack.
- You may use the top card from the waste stack. You may only go through the deck once.
- The object of the game is to move all the cards to the foundation stacks.

You will complete the specifications for the modules described in the specification file. Your specifications should not involve writing algorithms or pseudo-code. The specifications should use discrete mathematics to specify the desired properties. That is, you should be writing a *descriptive* specification as opposed to an *operational* specification. Specifications within a module are free to use access programs defined within the current module or from another module that is used by the current module. You should use the provided local functions. You do not have to add more local functions, but you can, if you find them helpful.

All of your code should be written in C++. All code files should be documented using doxygen. Your specification/report (for Part 1) should be written using LaTeX. Your code should follow the given specification exactly. In particular, you should not add public methods or procedures that are not specified and you should not change the number or order of parameters for methods or procedures. If you need private methods or procedures, you can add them by explicitly declaring them as private.

Part 1

Step 1

Complete the specification by addressing the comments given in the specification file `spec.tex`. You should note that this specification uses a "functional" approach. Rather than modify the state variables, the specification calls for the creation of a new object with the new state.

Step 2

Write a critique of the interface for the modules in this project. Is there anything missing? Is there anything you would consider changing? Why? Your critique should appear as the last section of `spec.tex`.

Step 3

Push your spec, which will include the critique, in (`spec.tex` and `spec.pdf`) to your GitLab project repo. The report, including the specifications, should be written in LaTeX. This step should be completed by the deadline for Part 1 of the assignment.

Part 2

Step 4

After the report has been submitted, you will be provided with a complete specification for all of the modules. Implement the modules in C++. Blank versions of the required source files and header files will be pushed to your personal repo.

Step 5

Experiment with the implementation. Test the supplied `Makefile` rule for `experiment`. The purpose of this rule is to provide a means for "playing" with the code as you develop it.

Step 6

Test the supplied `Makefile` rule for `doc`. This rule should compile your documentation into an html and LaTeX version. Along with the supplied `Makefile`, a doxygen configuration file is also given in your initial repo. You should not change these files.

Step 7

In the `test` folder you should have a corresponding test file for each module. The testing framework being used is called catch, as discussed in the tutorials. The header file you need, along with the `testmain.cpp` file, will already be pushed to your repo. Each procedure should have at least one test case. For this assignment you are not required to submit a lab report, but you should still carefully think about your rationale for test case selection. Please make an effort to test normal cases, boundary cases, and exception cases.

The supplied makefile (named `Makefile`) will have a rule named `test`. This rule should run all of your test cases.

Step 8

Push all of your code files to your GitLab project repo. This step should be completed by the deadline for Part 2 of the assignment.

Notes

1. Please put your name and macid at the top of each of your source files.
2. Your program must work in the ITB labs on mills when compiled with its versions of g++ (Version 7), LaTeX, doxygen and make.
3. So that you will have the correct version of g++, please add the following:

   ```
   . /opt/rh/devtoolset-7/enable
   ```

   to the bottom of your `.bashrc` file on mills. (An earlier version of g++ has to concurrently exist on mills; this allows you to switch to the version we are using.)
4. Many choices are available for containers to store sequences. So that unit testing will work between submissions, we need to make a standard decision. Therefore, please use a vector for your C++ implementation for the constructor for Stack:

   ```
   template <class T>
   Stack<T>::Stack(std::vector<T> s)
   {
     //details
   }
   ```
5. Pointers should *not* be used in your interfaces. All methods should use pass by value and all methods should return a value.
6. Implement templates following the approach used in the tutorial. That is, the CardT instance should be explicitly created in the `Stack.cpp` file. The header file `CardStackT.h` will only need to be the appropriate typedefs.

7. Enumerated types, like SuitT, should be implemented using `enum` in C++.
8. The specification shows RankT restricted to values between 1 and 13. For your implementation you are not expected to enforce this. You shoud use `unsigned short int`.
9. Natural numbers (ℕ) can be represented in C++ by `unsigned int`.
10. Tuple types, like CardT, should be implemented using `struct` in C++.
11. For exception, you should use the standard exceptions for C++ (`http://www.cplusplus.com/reference/stdexcept/`). The names of exceptions were chosen for the specification to make this straightforward. For the C++ implementation each exception name should be prefixed by `std::`. For instance, popping from an empty stack should raise a `std::out_of_range` exception.
12. **Your grade will be based to a significant extent on the ability of your code to compile and its correctness. If your code does not compile, then your grade will be significantly reduced.**
13. **Any changes to the assignment specification will be announced in class. It is your responsibility to be aware of these changes. Please monitor all pushes to the course git repo.**