Sprint 2 – Design & Implementation

Overview

In this sprint, you will:

- Design and implement a text-based card game using the Model-View-Controller architectural pattern.
- Keep track of your progress using version control.
- Maintain a coding style with a style checker.
- Check for memory leaks using a memory checker.
- Use static analysis to detect bugs and avoid dangerous coding practices.
- Generate documentation for your code using Doxygen.
- Use continuous integration to automate the running of software engineering tools.

Instructions

Setup

1. Fork your assigned card game repository so that your team has their own GitLab repository for completing the assignment.
    a. This step needs to be done by one of the team members.
        i. Add the other team members to the forked repository with `Maintainer` permissions.
2. Setup your repository to run continuous integration.
    a. A sample `.git-ci.yml` file is provided. It should be sufficient, but the team may need to update this depending on if the team changes the targets in the `Makefile`.
    b. Remember to make sure that the CI is set to `git clone` and not `git fetch`.
        i. This is found under `Settings->CI/CD->General Pipelines`
3. Edit the Doxygen configuration file (`docs/code/doxyfile`) to reflect the name of the card game you are building.
    a. Set the `PROJECT_NAME` variable to the name of your card game (e.g. `PROJECT_NAME = "Go Fish"`)
4. A Makefile is provided for compiling your game and running the SE quality checks.
    a. Due to the game's interactive nature, you will not be able to have the CI run the `make memcheck` target. You will need to run this on the command line to check for memory leaks.

Design

1. Examine the requirements document for your assigned card game to understand how to play the game.
    a. Consider meeting as a team to play the game together to understand how the game is to be played and determine any missing information from the requirements document.
2. Create a design for the card game that uses the `Model-View-Controller` architectural design pattern.
    a. Use the following conventions to make it clear which classes fulfil the roles of "model", "view,"

and "controller":

i. Model - `<Name>Model`

ii. View - `<Name>UI`

iii. Controller - `<Name>Controller`

b. Create one (or more) UML class diagrams in the `docs/design` folder showing how you have used MVC in your application.

Implementation

1. Implement your design for the card game.
    a. The implementation is expected to provide:
        i. All of the "red" requirements (i.e. those identified as essential to the game).
        ii. Some of the "yellow" requirements (i.e. those identified as important to the game).
        iii. Some of the "green" requirements (i.e. those identified as optional to the game) **if** all of the "yellow" requirements have been completed and your team chooses to do so.
    b. Some requirements may have to change as you understand the game better.
        i. Document any such changes in a file called `Requirement Changes` placed in the `docs/design` folder. Examples of changes include:
            1. Adding missing use cases.
            2. Updating a use case to make it more clear.
            3. Updating a requirement to make it less ambiguous.
            4. Changing the "colour" of a requirement (i.e. you may find that some "red" requirements are "yellow," some "yellow" requirements are "red," and so forth).
2. You are encouraged to create unit tests to help with creating a quality product, but unit testing (including mocking) is not part of deliverables for this sprint (i.e. not graded). Unit testing and mocking will be the topic of the next sprint.
    a. If your team chooses to create unit tests:
        i. Unit tests are to be placed in a directory named `test`.
        ii. The unit tests your team creates will be removed when the code for the next sprint is distributed.

Grading

Your grade for the sprint will be determined according to the following[1]:

1. Uses the Model-View-Controller architectural design pattern as documented by the file names and UML diagram(s) (25%).
2. A working implementation of the card game (50%).
3. The quality of the software (20%). Items that will be taken into consideration include:

[1] The instructor reserved the right to adjust the relative percentages to create the most accurate student assessment as possible.

   a. The build is passing on GitLab.
      i. Program compiles.
      ii. No style violations.
      iii. No memory leaks.
      iv. No static analysis violations.
   b. The documentation of the methods.
      i. Description of method.
      ii. Description of the parameters (if any).
      iii. Description of the returned value (if any)
      iv. Indication of exception thrown by method (if any)
4. Submission of the personal and team member review (5%).

Appendices

Understanding MVC in the context of a card game

The *Controller* in MVC is responsible for coordinating the use of *Models* by the *View*. For example:

1. The `Controller` requests from the `View` information about which card the player wants.
2. The `View` asks the player for the information.
3. The player indicates the card they want via the `View`.
4. The `View` returns this information to the `Controller`.
5. The `Controller` asks a `Model` (a player's hand or the deck) about the card.
6. The `Model` (hand) responds with a "Yes/No" answer.
7. The `Controller` requests that the `View` inform the use of the result.
8. The `View` displays the appropriate message.

Pulling from the original repository

It may be necessary to update assignment files from the repository that your team forked. The following instructions are given as a guide for how to do this.

1. Create a link from your local repository to the original repository. For example, if the original repository were `http://gitlab.cs.uleth.ca/course3720/goFish.git`, the command would be:

   ```
   git remote add upstream
   http://gitlab.cs.uleth.ca/course3720/goFish.git
   ```

2. Pull in the changes from the original repository:

```
git pull upstream main
```

3. Resolve any merge conflicts as needed.

4. Commit the updates to your repository (if needed)

```
git commit –m "Update from the forked repo."
```

5. Push the changes to your team's GitLab repository:

```
git push origin main
```