Name: Qing (Cecilia) Lu
Username: luqing
Student ID: 300363602

1. Take 3 commit URLs as examples of individual contribution.

URL	Description
https://gitlab.ecs.vuw.ac.nz/course-work/swen225/2020/groupproject/team26/chipschallenge/-/commit/77919a9de34854caaa5346870baa3422e9c41fbe	Added basic Swing objects into the GUI class: 1. Set the JMenuBar, and added JMenus and JMenuItems: • Game — New Game; • Options — End Current Game, Exit; • Level — Level 1, Level 2; • Help — Instruction. 2. Set the board canvas: • Wrote the inner class BoardCanvas that extends the class Canvas; • Created a BoardCanvas object in GUI and added it onto the left JPanel of the whole JFrame. 3. Set the sidebar: • Wrote the inner class InfoCanvas that extends the class Canvas; • Created an InfoCanvas object in GUI and added it onto the right JPanel of the whole JFrame. 4. Method *readAllLines()* is to read the *Instruction.txt* file and return its content as a String. 5. Added ActionListener to the JMenuItem "*Help*": pop up a window that shows the game instruction (content in the *Instruction.txt*).
https://gitlab.ecs.vuw.ac.nz/course-work/swen225/2020/groupproject/team26/chipschallenge/-/commit/ec2f116e920843562833245bb186485d72c80307	Added another thread to load the animation image on the board canvas before a game starts: • Firstly, I wrote the class AnimationThread for the animation image. It initializes a LoadingScreen object in its constructor. In addition, it has *start()*, *stop()*, and *run()* methods to control the status of the thread. • Secondly, I created an AnimationThread object and implemented it in the GUI class.
https://gitlab.ecs.vuw.ac.nz/course-work/swen225/2020/groupproject/team26/chipschallenge/-/commit/30ab48717a2eef1ce0b87cf7473c37748d2010bf	Managed the time countdown on the sidebar (infoCanvas): When initializing the first game after running the program, it will initialize a RunnableThread object (named timeRuunableThread, another thread for the time countdown), and start this thread in GUI. Also, the number is drawn in the first black box on the sidebar to show the time left. The time is updated every second.

2. Ranking of the contribution of each person in the team.

Name	ECS Email Address	Contribution (from 1 to 5)
Daniel Marshall	marshadani@ecs.vuw.ac.nz	5
Oscar Sykes	sykesosca@ecs.vuw.ac.nz	5
Qing Lu	luqing@myvuw.ac.nz	5
Samuel Kain	kainsamu@myvuw.ac.nz	5
Ruiyang Zhang	zhangruiy@ecs.vuw.ac.nz	4
Yun Zhou	zhouyun@myvuw.ac.nz	4

3. Reflection Essay

a. What knowledge and/or experience have I gained?

- Group work experience on Git: I got a better understanding of how to work on group projects on Git (e.g., how to create issues and assign them to team members; how to commit, merge local code with the code in the master branch and push it, etc). It's a good practice in the collaboration with colleagues on Git in a real working environment.
- Getting more familiar with Swing objects and layout management: The layout manager helps to manage look-and-feel-dependent component appearances.
- Multiple threads: I implemented 3 threads (GUI, animation image, and time countdown) in the Application package, which provides me with an impression of how multiple threads interact.

b. What were the major challenges, and how did we solve them?

- Handling multiple pressed keys with the KeyListener. I implemented key events in the k*eyTyped* method at first. But the combination of *Ctrl + any letter* didn't work. Then I tried to put the function *KeyListenerAdded* in *keyPressed* and *keyReleased* method respectively and tested it. Finally, I found that the correct place to put this method is *keyPressed*, as the *keyTyped* could get only one key's event (press and release continuously) each time (it doesn't work when more than one key is pressed and released).
- Handle the board's movement and always show a 9x9 board. We decided to do a 15x15 Board class in Maze, and keep two boards (15x15 and 9x9) for rendering. The Renderer receives Maze's board fresh each time, keep track of what square in the actual tile array (0, 0) is in the 9x9 array, and pass the starting point of the 9x9 board to Application.
- Cooperating with team member who is in another country. During our in-person meetings, we summarized what we had done and what we needed to do in a google document clearly and effectively, and shared it with team members who did not attend in time.

c. Which technologies and methods worked for me and the team, and which didn't, and why?

- Class Diagram worked for me because it gave me the first idea of how to design the program structure better and clearer, and managed the relationships between classes more easily.
- CRC cards worked for me because they clearly represented class details in an OO system. CRC models are an effective tool for detailed design: adding and allocating fields and methods in AnimationThread, RunnableThread, GUI and its inner classes.
- Dependency graph worked for me and the team because it provided a visualization of who we need to talk to. It helped to think about what information was (or would be potentially) required by other parts (what functions shall I do; what getters and setters shall I implement).
- Code contract worked for me and the team: e.g., we used *assert as* the precondition and postcondition to make sure some requirements are satisfied (such as the returned value is not null, or a boolean value is true); otherwise, it threw an exception, which helped to debug.
- We didn't code and generate State Diagram on *Umple (*instead, we used IntelliJ and Eclipse). But I used *SmartDraw* to create a UML class diagram to see the behaviour of our program.

d. Discuss how you used one particular design pattern or code contract in my module. What were the pros and cons of using the design pattern or contract?

One of the design patterns that I used in the Application module is Model-View-Controller (MVC) pattern: Model is the Board object; View is the *Renderer object* that draws the game interface on the board canvas as a view for the user; Controller is GUI's *Listeners* (*keyListener* and *mouseListener*). The *Controller* accepts input from the user and converts it to commands to manipulate *Model* (*board.move()*), which then updates the *View* (*renderer.redraw()*).

Pros	Cons
- Faster development of the application: easy to assign tasks to other team members and work in parallel; - One-way data flow improves the traceability: View (renderer) could only be updated by the changes of Model (board); - Easier to debug: multiple levels are written in the application, which supports debugging level by level.	- The isolated development process by multiple authors may lead to delay in our modules' development; - MVC pattern is very complex, as we had to spend more time discussing a better way of splitting the whole logic of the program into 3 parts; - The controller manages too much logic, making it more difficult to maintain the code.

e. What would I do differently if I had to do this project again?

- Add a "Start Game" button on the sidebar to make the game more user-friendly. The board in this project followed the appearance of the board shown in the handout — only a menu bar on the top; no buttons on the board. Adding a button on the sidebar will allow the user to start and end a game by simply clicking the button rather than clicking twice on the menu bar.
- Change the font of the time displayed to improve the game interface. I'd like to make the countdown time a digital clock, which looks nicer: 
- Make the JFrame of the board resizable. It will allow the user to drag the game window to change its size; and the size of the tiles and all the pictures on the board will change and suit the real-time size of the window. However, cooperation with Renderer would be necessary, as it may need Renderer to draw the board more smoothly.

f. What should the team do differently if we had to do this project again?

- Each team member creates an individual branch. What we did in this project is:
  - Firstly, committing what we have done;
  - Then, merging our local code with the master branch (accept their changes if there is any);
  - Lastly, pushing the updated code (code only in our own package) onto the master branch.

  Although what we have done does not override or destroy other team members' code, we would like to make individual branches the next time we do a group project. This requires us to better cooperate to deal with the merge requests.
- Use CRC cards to structure the whole program. CRC cards help to assign tasks (what fields and functions a class needs to include) and represent the classes clearly. These advantages prevent conflict risks, especially when a function could be implemented in either of the classes.
- Improve Code Contracts and make checks clearer. We have had code contracts in our project, e.g., *assert* for "String::contains" as a postcondition. We can employ more contract elements to take advantage of contract enforcements, such as build-time checks for static analysis.