FOOP Final Project Report

吃冰不揪 Team members

b06902011 陳義誠
b06902014 林立
b06902016 林紹維
b06902040 游高政

Responsibilities

b06902011 陳義誠

- **GameLoop**, **Game**, **GameView**, **Button**, **LevelUpButton**, **SummonButton**
- Controller, view, and GUI design
- Debugging and testing

b06902014 林立

- **Main**, **States**, **ImageReader**, **World**, **TowerHpBar**, **Level**
- Class diagram, meeting record
- Merge branches to main
- Testing, debugging

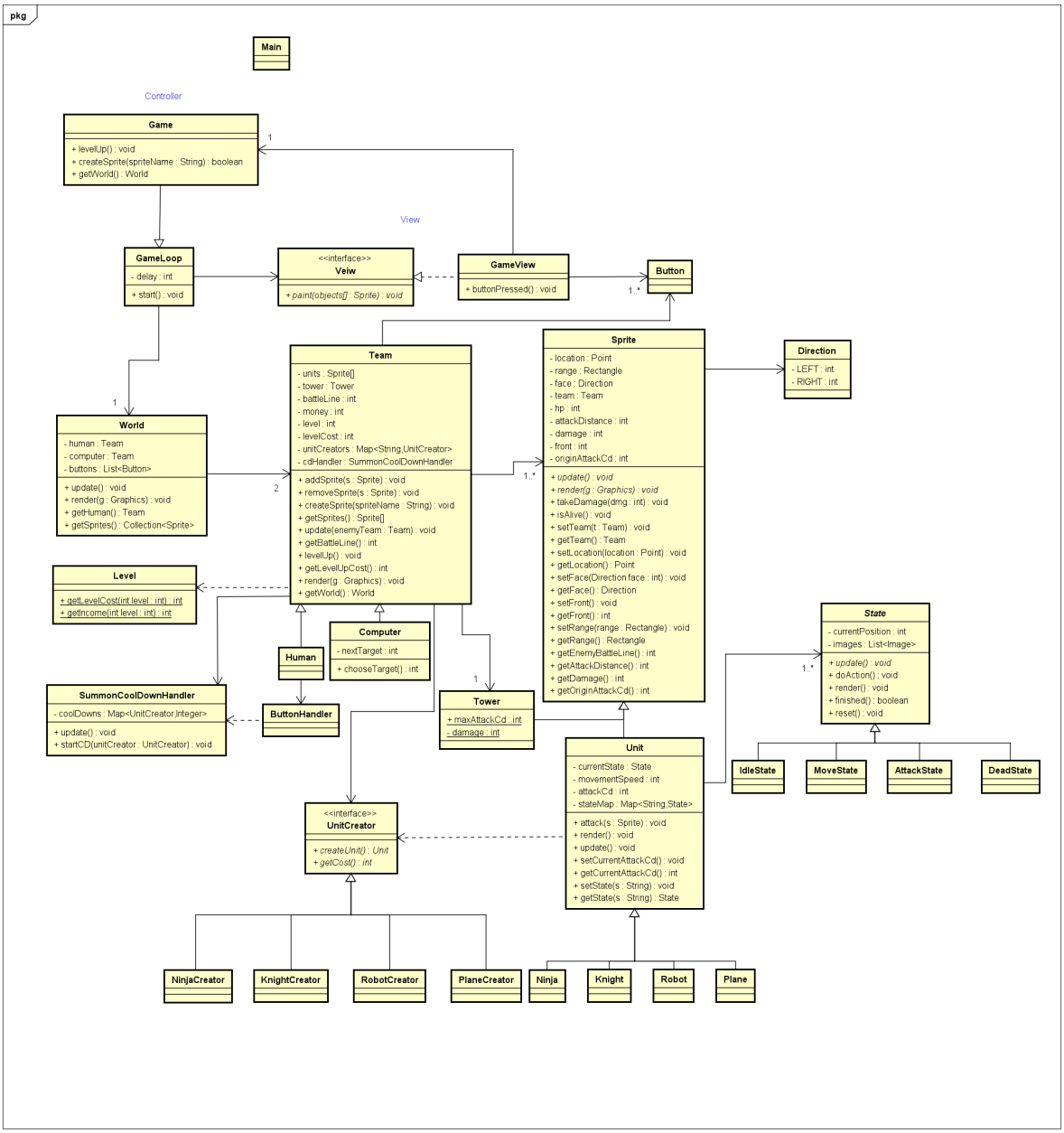
b06902016 林紹維

- **States**, **Sprite**, **Units**, **Tower**, **UnitCreator**s, **StateHandler**
- Testing, debugging
- Game balance adjustment on **Sprite**'s ability

b06902040 游高政

- **Team**, **Human**, **Computer**, **SummonCoolDownHandler**, **ButtonHandler**
- Game design and balance adjustment
- Testing, debugging

Class Relations



Main

Initialize different classes. Once the classes are all setup, tell the Game to start.

GameLoop

An abstract class defining the game flow, including start, loop (which updates **World** and renders **GameView** for each frame), game over, and restart. It also defines the interface **View** which **GameLoop** can render whatever implements via the methods.

Game

A subclass inheriting **GameLoop** defining the controller. The methods are used to react the command from the player. It usually communicates with **Human** of the model, but also has a reference of **World**.

GameView

The extended **Jframe** class implementing **View** in **GameLoop**. It is nested with the following inner classes, all extended from **JPanel**:

- **GameplayPanel** contains the components to show through the gameplay. It has the following inner classes:
    - **Background** paints the background only once.
    - **Canvas** defines the place to be rendered. When **GameLoop** tells **GameView** to render, only this class would be repaint.
    - **ButtonsPanel** contains and shows a list of **Button**s player can operate, including level up and summon **Unit**s.
- **GameOverPanel** shows the game over scene according to the result (win or lose), and offers the restart option. **GameView** switches between this and **GameplayPanel** when the game is started/restarted/over.

Button

An abstract class inheriting **JButton**. Any button operable for the player should extend from this class.

LevelUpButton

A subclass inheriting **Button** with an **ActionListener** to tell **Game** to level up **Human**.

SummonButton

A subclass inheriting **Button** with an **ActionListener** to tell **Game** to make **Human** summon specific **Unit**.

World

Keep track of the two teams: **Human** and **Computer**. Call the **Team**s to update and render themselves. And retrieve sprite information among the teams.

Level

Define the money that can be obtained given the level of the **Team**. And define the upgrade cost of each level.

State

Given a **Unit** and a **List of Image** of the state, render them respectively in different frame. Check the conditions of the associated **Unit** and decide what to do or whether to change to different state or not.

ImageReader

Given the name of the **Unit**, the **ImageReader** helps us to retrieve the **List of Images** under its folder. After making adjustments to the **Images** (for example: adjust the size of each **Image**, sort them to the correct order), the **Images** is then passed into the **State**.

Sprite

An object (either a tower or a unit), that belongs to a **Team**. Occupies a range on the canvas, has the ability to attack and is subject to damage.

Tower

Each **Team** has only one **Tower**, when the **Tower** collapses, the team loses.

Unit

Abstract class of **Ninja**, **Robot**, **Knight**, and **Plane**. Has the state attribute, and keeps information of position, cooldown, and speed.

UnitCreator

An abstract class which is the predecessor of each of the units' creator. Returns a new corresponding **Unit** according to its class.

StateHandler

Abstract class of the **State**s, defines the method of transition.

Team

An abstract class maintains what a team should have, including **Tower**, **Unit**s, money, level, etc.

Human

An extend class of **Team** shows the information of player including level and money and maintains the buttons.

Computer

An extend class of **Team** with decisions made automatically by AI.

SummonCoolDownHandler

Help the **Team** to handle the cool down of summons.

ButtonHandler

Help **Human** handle the **Button**s.

Advantages

Unit Follows OCP

Just add new extended **Unit** and **UnitCreator** classes and initialize in **Main**.main() with

```
buttons.add(new SummonButton(game, unitName));
```

(where *unitName* is also defined in **Main**.main()) would make the new **Unit** available. Since the size of **Button** is relative and not absolute, increase and decrease of available **Units** would not break the GUI either.

New State is Easy to Create

State handles the rendering of **Unit**. Adding a **State** only need to add a folder name under the **Unit**'s folder, and insert all the images of the **State**. Then the unit will automatically be rendered correctly during the newly added **State**.

Team Follows OCP

Computers with new AI and new design can be easily done.

An Easy but Challenging Game

Easy to know how to play by intuition, but challenging to win.

Disadvantages

Overlap in Different States

The update() function has a lot of overlapping code between different **State**s. But there are many behavioral variations in them. Maybe there should be more abstracted common behaviors or more delegation of variance.

The State Changing Conditions are Hard Coded

There is no Finite **State** Machine to let us initialize the state changing conditions. So each **State** should specify those conditions independently. This makes **State** not follow OCP. If we want to add more states to expand our game, the process would be more laborious.

No Audio Effects Provided

The game do not come with sounds.

States are Not OCPed

There are currently four **State**s in our setting, but each transition between **State**s are written in each of the update() methods, since we did not use finite state machine in our system.

There is only One Stage

So far, there is only one stage designed, and that adding new stages would need to change the **Computer** class in order to achieve different computer behaviors.

Window Size is not Flexible

This is due to rendering **Sprite**s with absolute positions. Besides, although all the panels in **GameView** are relative to WIDTH and HEIGHT defined in it and pre-modifying them before compiling and starting the game would make displaying the panels just fine, dynamically resizing the window while running can still break the GUI.

Other packages

None

How to Play the Game

1. Click on one of the buttons at the bottom of the screen
    - The first 4 buttons on the left summon unit
    - The 1 button on the right increases your level
    - The cost and cooldown of the buttons are showed on the buttons themselves
    - Leveling up will increase the money accumulating speed
2. Try to beat the tower on the left by summoning units/leveling up; while defending the right tower to prevent it from being destroyed
3. The money accumulating speed of computer is slightly faster than human. Use your stratagies to utilize the properties of different units and try to out smart the computer.
4. Good luck!