Thief

By

Richard Pham
Oakar Kyaw
Ethan Liao
Bennett Wong
Jeremy Anunwah

Introduction

For our final project, we were prompted to create a text-based game. The game is set inside of a building which is represented by 81 squares (a square = a space) in a 9-by-9 grid. The user of the game is playing as a Thief. The objective of the user and the win condition of the game is to obtain a briefcase, which is hidden in one of nine “special” rooms. These special rooms are evenly spread out across the building. The thief has three tries to find the briefcase before he is eliminated by ninjas in the building.

There are six ninjas patrolling the building and if they move onto the same space as the Thief, then they will kill the Thief with a stab. The Thief will then be either sent back to the default position, or the game will end, depending on the current amount of lives of the Thief. The ninjas will move in a random direction, regardless of where the Thief’s current position is. The one exception is that if the Thief moves into a room that is adjacent to a room with an ninja, the ninja will automatically kill the Thief.

The Thief is equipped with a gun that can shoot in a straight line and kill ninjas. There is only one bullet inside of the gun at the start and is able to obtain more bullets throughout the building which is one of the few power-ups that the Thief can obtain.

There are three power-ups randomly distributed across the building and they can assist the Thief. The power-ups can be: an additional bullet, invincibility, and radar.

The first power-up which is the additional bullet, will add a bullet to the Thief’s gun, which can be used as stated before can kill a ninja.

The second power-up is the invincibility, which will negate the attack of ninjas for five turns. This way the Thief can get through rooms without having to be careful about the ninjas.

The last power-up is the radar, which will show the Thief the exact location of the briefcase. With this the Thief can quickly obtain the briefcase, not having to waste any turns moving in the direction away from the briefcase.

The user is also able to quit the game whenever they would like to. They are also allowed to save the game at any point of the game, then load the game up where they left off.

There are several rules and aspects to this game:
(The game revolves around the player's turn and then the ninjas' turn)

1) The player will start with 3 lives. If the player loses a life, then the player will return to the starting position (bottom-left corner of the building). Everything else will be retained (location of briefcase, enemies, power-ups, etc.) There will not be anymore power-ups that come up in the building.

2) The building is pitch black, so the user cannot see the contents of the building. On every player's turn, the player is able to "look" once in any direction. This will reveal the next two squares in the specified direction and tell the user if there is an enemy ahead or if the path is clear. This is very important so that the Thief will not accidentally move into a room that is adjacent to an ninja.

3) The user can only enter "special rooms" from the northside (the Thief can only enter the "special rooms" if they are above the room first).

Problem Description

We decided to start with the basic objects in the game since it was decided to do the bottom up implementation). So we did the class design at first which was also our first milestone.

When our team was discussing the creation and initialization of the objects, we realized that there were only going to be two entities that would be moving around as the game would be in session: the player and the ninja. Thus, we created an abstract class in order to encapsulate similar functions that the player and the ninjas would have.

In addition to this, we created another abstract class to represent the other objects that would be in the game. This included the power-ups and the
We created "objects" representing:

- The Thief
- The Grid (represents the building)
- The Enemy
- The Thief's Gun and Bullet
- The Powerups
- The Night Vision Goggles

- The Rooms
- The User Interface
- The Game Engine
- The Main

We deduced that we needed to design these objects in the following order:

1) The Thief
2) The Thief's Gun/Bullet
3) The Thief's Night Vision Goggles
4) The Enemy
5) The Rooms
6) The Powerups
7) The Grid
8) The User Interface
9) The Game Engine
10) The Main

Approach to solution

The Thief

The Thief is ultimately the character that the user is controlling inside of the game. We started with the generic variables which are the Thief's row position, Thief's column position, and a boolean that specifies whether or not the player is alive.

The constructor requires an input for row and column and an instance of a Thief would be created with the specified row and column along with the alive variable being *true*.

Since all of these variables are *private* we provided get methods that allows us to retrieve these attributes when we need to access them in other classes. We also created setPosition methods that allows us to modify the position of the Thief when the Thief is killed by the enemies. The method can also be used to reset the position of the Thief when killed..

Lastly, we created methods to move the Thief. We created four methods to move up, down, left, and right.

The Bullet

The Bullet class will extend the **InGameObject** class (see below)

The Bullet is pretty straightforward as it only serves one function, which is being a bullet. This Bullet class won't have a shoot method as the shooting function will be designed in a different class. The Bullet will have one variable called ammo which holds the number of bullets left.

The Bullet has a default constructor that will start with a default position and will contain 0 ammo. There is an additional constructor (this one we will use) that allows us to specify the position and the number of ammo.

The Bullet will have a get and set ammo method.

InGameObject

This is a class we created to prevent redundancy (copy and pasting) in code. This class is extended in the Bullet, the Powerups, and the Goggles.

There is a single constructor that requires an input for the row and column.

The InGameObject holds a row and column position and a boolean that specifies if the "object" is onBoard or not.

There are methods that get and set the position values and a method to check if the object is onBoard. We also created an use method that changes the onBoard value to false (since it is "used").

Goggle

The Night Vision Goggles will extend **InGameObject** (see above).

There are variables for the column and row positions as well as a boolean that specifies if the goggle is on or off. Additionally, the Night Vision Goggles reveals two positions/spaces, thus the class must also receive a second row and column position.

The constructor will use the InGameObject constructor to set the position of the goggles, and then will set the goggles to be turned on.

There are GoggleOn and GoggleOff methods to enable the goggles to be on and off and there is a getGoggle method to obtain the isOn boolean value.

Enemy (Ninja)

The Enemy will extend the **Character** class.

There is a single constructor that requires an input for a row and column which will then create an Enemy in that row and column.

We needed to have the Enemy be able to stab a player and eliminate a life from the player so, the Enemy has one method called stab and this method will require a Thief target and then the Thief will then lose a life.

Character

The abstract Character class will implement the Serializable.

There is a single constructor for the Character class, which will require an input of row and column. These will be used to put the Character on a specific part of the board. The Character also will be given a status of being alive or not.

The Character class has methods to set/get the position row/column of the Character. There will also be methods to move the Character up, down, left, and right. Finally there is a method to find whether the Character is alive or not.

Room

This class represents the rooms of the game. The room will each take one spot on the board, while one of the will have the briefcase which is the goal of the player to obtain. There is only one important room, which is the room with the briefcase. Nothing will happen if the Thief enters a room with no briefcase.

The Room class has one variable (boolean) called hasBriefcase. This variable specifies if the room has a briefcase or not.

There is one constructor that requires an input of a row and column and will always start with no briefcase.

There are two methods: one that sets the Briefcase which will set the hasBriefcase variable to true for that room and the other checks if there is a Briefcase in that room by returning the has Briefcase variable.

Powerups

The Powerups class will extend the **InGameObject** class (see above).There are separate classes for all of the power-ups: the Radar, the Bullet, and Invincibility.

The Radar class will have a constructor which takes in an input of row and column, which will be used to set the Radar in the building. The Radar will be given a switch, which sets the Radar on/off. The Radar has a method to turn on the radar when the power-up has been collected by the Thief, and also other methods that obtains the current status of the Radar.

Grid

Decided that Grid needs the following: to create rooms, to create ninjas, to create power ups, to randomize the location of the briefcase every time a new Grid is created, to randomize the location of the power ups every time a new Grid is created, and to randomize the location of the ninjas every time a new Grid is created. Moreover, we decided that we would keep all the instances of the in-game objects and characters in the Grid class. This would make it easier to save and load the game, as we would only have to save the Grid class rather than a multitude of classes.

Discussion of Implementation

While many other groups decided to work with 2D arrays in order to print the Grid, our group decided to refrain from this and use the toString() method in order to print the grid. Because of this, instead of having the array indexes reference the location of the entities and objects, we assigned column and row values to the various objects on the Grid. This allowed us to update the location of the player and enemies without having to go through and sort through a 2D array.

```
public String toString(boolean debugMode) {
    this.debugMode = debugMode;
    String result = "";
    for(int i=0;i<GRID_SIZE_ROW;i++){
        for(int j=0;j<GRID_SIZE_COLUMN;j++){
            result+="[";
            if(debugMode){
                if(user.getPositionRow()==i && user.getPositionColumn()==j){
                    result+="P";
                }else if(checkIsRoom(i,j)){
                    if(checkIsBriefcaseRoom(i,j)) {
                        result+="C";
                    }
                    else {
                        result+="R";
                    }
                }else if(bullet.getPositionRow()==i && bullet.getPositionColumn()==j && bullet.isOnBoard()){
                    result+="B";
                }else if(invinc.getPositionRow()==i && invinc.getPositionColumn()==j && invinc.isOnBoard()){
                    result+="I";
                }else if(radar.getPositionRow()==i && radar.getPositionColumn()==j && radar.isOnBoard()){
                    result+="A";
                }else if(checkIsEnemy(i,j)){
                    result+="N";
                }else{
                    result+=" ";
                }
```

Also, when creating the Grid, we had to make sure that objects would not be placed in the same position as other objects. Thus, we had to create separate methods for checking the positions of the various entities and objects, such as the ninjas, rooms, and the power-ups. Thus, the Grid became the main class that would hold all the instances of the various objects and entities of the game.

```
public void randomizeEnemies(){
    int randomNumber;
    int row;
    int column;
    for(int i = 0; i < ninjas.length; i++){
        randomNumber = rand.nextInt(81)+1;
        row = (randomNumber-1)/9;
        column = (randomNumber-1)%9;
        while(checkRandomAvailability(row,column)){

            randomNumber = rand.nextInt(81) + 1;
            row = (randomNumber-1)/9;
            column = (randomNumber-1)%9;

        }
        ninjas[i].setPositionRow(row);
        ninjas[i].setPositionColumn(column);
    }
}
```

The next thing we had to tackle was the randomization of the positions of the ninjas, power-ups, and briefcase location. We took a random integer from 1 to 81 and from there, we split the integer into the row and column values. The row would be the random integer generated subtracted by 1 and divided by the max amount of rows(9). Then, the column value would be the random integer generated subtracted by one and the remainder of dividing by the max amount of columns (9).

Additionally, we realized we needed to make another class for the Night Vision Goggles due to the fact that the the object also needed to be assigned and received positions. However, rather than the positions refer to its placement on the Grid, the positions would refer to spaces where the player looked in.

Testing data

Test	Description	Pass/Fail
Move	The player will move in the direction specified by the user through his or her input. If the user presses ‘w’, the player moves up. If the player presses ‘s’, the player moves down. If the player presses ‘a’, the player moves to the left. If the player presses ‘d’, the player moves to the right.	Pass
Look	The user will press ‘g’ in order to use the night vision goggles. The player will then be asked what direction he or she would like to look in. If the user presses ‘w’, the top two squares above the player will be revealed. If the user presses ‘a’, then the two squares to the left of the player will be revealed. If the user presses ‘s’, then the two squares below the player will be revealed. If the user presses ‘d’, then the two squares to the right of the player will be revealed. If the user has already used the night vision goggles this turn, then the game will not allow the user to look in another direction.	Pass
Shoot	The user will press ‘e’ in order to shoot his or her gun. The player will then be prompted for the direction to shoot in. If the user	Pass

	presses ‘w’, the player will shoot a bullet in the ‘up’ direction. If the user presses ‘a’, the player will shoot a bullet in the ‘left’ direction. If the user presses ‘s’, the player will shoot a bullet in the ‘down’ direction. If the user presses ‘d’, the player will shoot a bullet in the ‘right’ direction. If the player does not have a bullet, then the gun will not shoot. If the player shoots in the direction of a ninja, then the ninja will die. If the direction the player shoots in has two ninjas, then the bullet will kill the ninja that is closer to the player.	
Ninja movement	If the user inputs a command to move or successfully shoots his or her gun, then the ninjas will all move to a randomly selected position . If the ninja is next to a wall or a power-up, then the ninja will not move in those positions. If the player is next to a ninja, then the ninja will kill the player.	Pass
Pick up Invincibility	If the user picks up the Invincibility power-up, the player will be invincible for 5 turns. If the player is invincible, then ninjas will not attack the player and ignore the player.	Pass
Pick up Bullet	If the user has a bullet, the Bullet power-up does nothing. If the player does not have a bullet, the player receives a bullet.	Pass

Pick up Radar	If the user picks up the Radar power-up, the player will be shown the room the briefcase is located in.	Pass
Pick up Briefcase	The player picks up the briefcase. The user wins the game, and the game ends. The user will then be returned back to the main menu.	Pass
Check the room	The user can only check the room from the top of the room.If the user tries to check from any other directions other than top, an error message will be printed.	Pass
Debug mode	If debug mode is turned on, then the user will be shown the positions of every entity and object on the grid. This will include which room the briefcase is located in.	Pass
Lose game	The user will lose the game once the user lose all of three lives. The game session will then end, and the user will then be returned to the main menu.	Pass
Ninja kills the player	If a ninja kills the player, then the player will die and his or her position will be reset to the starting position. Additionally, the player will lose one life.	Pass
Save game	Save the position of all the ninjas, the players, the briefcase and the power-ups to a file. The status of debug mode will be saved, too.	Pass
Load game	Load the position of all the ninjas,	Pass

	the players, the briefcase and the power-ups from a saved file. The status of debug mode will be loaded. too.	

Conclusion

The assignment was overall straightforward. Things we learned were how Serializable worked and how the saving and loading the game worked. We also learned the importance of extending classes and having a superclass since they prevent redundancy (copy and pasting) in our code and classes.

From this final project, we learned the importance of Object Oriented Programming in a team/group setting. If there was a group member who coded one class holding all of the variables, constructors, and methods, the code would be congested and hard to read by the other group members.

Separating the instances and objects into classes not only made it easier for the individual to understand, but it allowed easiness for us to collaborate since we could focus on one class at a time and simply reference to other classes. It also made it easier for us to see what was changed and what was added. If it was all in one class, we would have to scroll up and down and it would have caused a lot of confusion and would have most likely gave us headaches.

Suggestions for Improvements

1) Clean up code to make it more readable
2) Implement AI movement for ninjas
3) Implement GUI