CSC207: Design Document

Chun Yui Terry Yeung, Tat Yin Sen, and Yan Nowaczek

November 22, 2021

1 Specification

Missile Mayhem is a arcade style game where a user moves a character to avoid incoming missiles. The goal is to survive as long as you can with the pilot having only 5 lives. The pilot is controlled with the arrow keys. Missiles are free moving and chase the pilot with built in navigation intelligence (not always perfect). The game can be paused. When the game is first launched, the user will be prompted to enter their name and will not be asked again. The game will remember users and their scores, i.e., the number of seconds survived. At the end of the game the best scores will be displayed. There is going to be a variety of missiles that have varying speeds and movement types, e.g., bouncing, circling, and accelerating.

2 Progress

Three weeks ago several members left the team and the code of the original project was lost. The above three members decided to continue from scratch. Initially we communicated through Discord and exchanged zip files outside GitHub. About two weeks ago GitHub had been reactivated and the team started using GitHub to write the code.

3 Functionality

`MainLoop` prompts every 30 milliseconds `GameState`, which keeps track of game phases, clock, and username. It activates `Canvas` that displays `JPanels` in sequence. It also activates `GameLogic` which decides which panels need to be drawn and contains `Iterator<JPanel>`. Finally, `GameState` listens to `Console` for user input. Each `JPanel` may contain additional functionality. For example, `JButtons` has its own listener, `JPanels` with `BufferedImages` of the missiles have their own navigation logic to determine their next position. Implementing `JPanels`, `JButtons`, `JLabels`, and `JTextBox` is difficult and we frequently run into problems. At this time, the game features the input screen, the pause screen, and the game screen where the user can move the pilot. The game stores the username in an external `stats.txt` file.

4 Class Diagram

We followed the SOLID principles and each class has a single responsibility with the exception of `Canvas`, which both updates and paints `JPanels`. In this diagram, colors indicate SOLID layers and boundaries. Related classes have not yet been put separate folders.

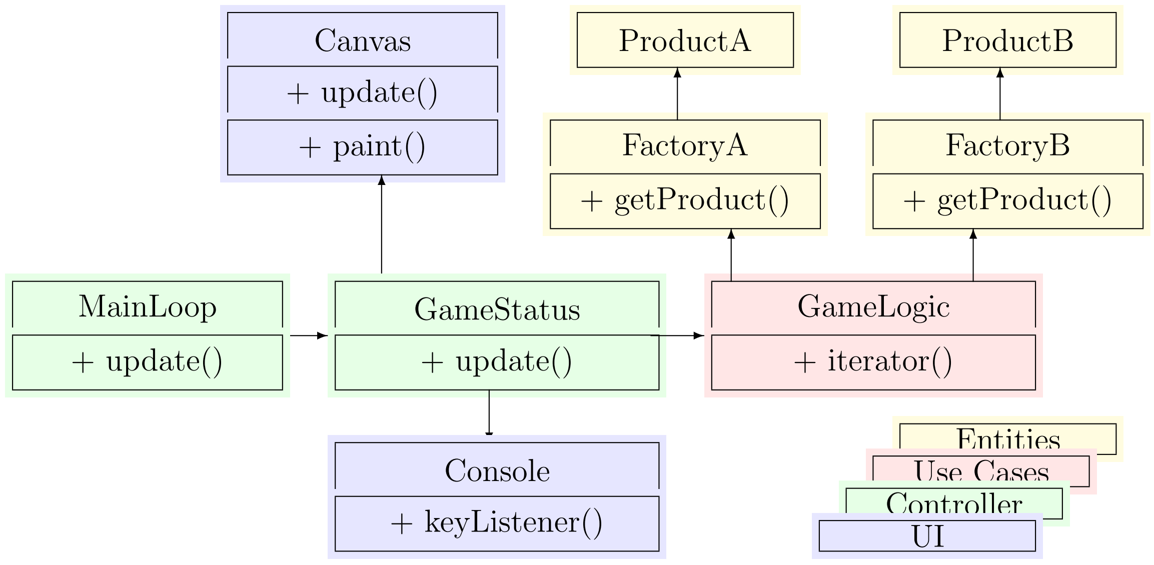


Figure 1: Missile Mayhem class diagram

5 Phase I responses

5.1 Major decisions

In the beginning of November the three remaining team members restarted the project from scratch.

5.2 Clean Architecture compliance

We adhere to the four layers as depicted in the class diagram above. Entities, which we call products, reside in the most inner layer. The next layer is the use cases, which are handled by `GameLogic`. `GameStatus` is in the next outer layer and the last UI layer consists of `Canvas` or Presenter and `Console`. The outer classes are dependant on the inner classes.

5.3 SOLID principles compliance

We attempt to make sure that each class has a single responsibility. For example, `Console` was initially implemented to listen to the keys and has not needed any modification. The open closed principle works well because classes like `GameStatus` and `GameLogic` are extended but not modified. The Liskov substitution principle is being followed in, for example, `Product`s. They are all passed around as `JPanel`s but in `Canvas` they can be substituted by `ProductB` and `ProductA`, which allows `Canvas` to extract additional information about the placement of the components of `JPanel`s.

5.4 Packaging strategies

We have not applied any packaging strategies yet. It is planned to organize classes by layers as specified in Clean Architecture. Only images and text files have been separated in the resource folder and the `stats.txt` file is located in the root folder.

5.5 Design patterns summary

Design patterns have been well implemented. The `Iterator<JPanel>` works really well, allowing different parts of the code to quickly traverse individual `JPanel`s and handle them according to specific individual features. The

Abstract Factory pattern has been implemented but it does not help much. The `Builder` will be a better option because `JPanel`s are actually assembles of many components. The Strategy design pattern has been implemented with respect to `ProductB`. For details, see sections below.

5.6 Progress: open questions

Controlling `JFrame` components is complicated. The information is hard to get and most methods do not work. For exmaple, we still cannot find a why to control the input text box. The images drawn by `Canvas` tend to flicker a lot and there is no clear solution for moving objects. For the input screen, we have stopped the repaint algorithm which eliminated flickering completely.

5.7 Progress: what works well

The `Iterator<JPanel>` is fantastic - simple and useful. Also, the algorithms that control the movement of missiles are working fabulously. Missiles themselves calculate the best route in order to hit the target.

5.8 Progress: work assignments and plans

Edward (Tat Yin Sen) worked on the code outside of GitHub.

Terry (Chun Yui Terry Yeung) lately cleaned and commented the code and implemented the chase algorithm.

Yan (Yan Nowaczek) lately produced graphics for buttons and implemented the `Iterator<JPanel>` and finalized this report. He has been acting as the self proclaimed development lead.

The next items to do is the completion of the user input screen, the replacement of the Factory Strategy with the Builder Strategy, and the implementation of remaining game phases.

6 Class details

Overall, the design relies on a small number of classes, each of which contains a small code that deals with a single concern. Class `Canvas` is an exception. It paints and updates components.

TODO Related classes need to be put in separate folders.

6.1 MainLoop

Class `MainLoop` is very simple. It contains only `main()` that repeatedly calls on class `GameStatus` to update itself and asks if it is time to exit the loop.

TODO Due to its simplicity there is nothing to add to this class.

6.2 GameStatus

Class `GameStatus` keeps track of the game stage, the position of the image that the user can control (target/pilot), and the clock. It calls on class `Console` to get user input and decides on the stage of the game, e.g., 0 - in progress, 9 - exit. It passes target coordinates and the clock value to class `Canvas` and calls on class `Canvas` to paint the screen.

TODO More game stages need to implement. So far there are only five phases: 0 (in progress), 5 (start), 6 (pause/menu), 7 (user input), and 9 (exit).

The clock is working but does not stop running during pauses. The position of the target is implemented and it is time to implement collisions and the game score.

6.3 Console

Class `Console` captures the integer values of keys pressed by the user.

TODO Nothing. The code inside the class is very simple. The input to JButtons is handled by the buttons. Beware that JButtons take away the focus, which needs to be return to JFrame.

6.4 Canvas

Class `Canvas` paints images (products) according to the stage of the game. Initially, this class creates a set of products which during the game, can be added to or remove from the canvas. This class also calls on products to update their positions if appropriate.

TODO Perhaps, there is a need to remove the `update()` responsibility from `Canvas`.

6.5 GameLogic

Class `GameLogic` is where the user cases are implemented. It creates a list of `JPanel`s for painting and a list of the same size with boolean values. The second list indicates which components are to be painted for a given game stage. It implements the **Iterator Design Pattern** and its `iterator` can be used elsewhere in code to easy scan all paintable components.

TODO Not much for now. The logic will have to updated as new stages of the game are created.

6.6 ProductA, ProductB ...

Classes `ProductX`s are of type `JPanel` so that they can be added to `JFrame` and displayed. Unfortunately, `JPanel`s come in many variants and sometimes need to be handle individually by calling the name of the class, which is `ProductX`. Some `JPanel`s contain `JButton`s to request user commands such as `start` or `menu`. One `JPanel` contains a `Jlabel` to display the clock. When appropriate some `ProductX`s contain methods that calculate the image position on the screen.

TODO Apply the **Strategy Design Pattern** to the methods that calculate trajectories. These algorithms are similar to one another and quite complex, resulting in bulky classes.

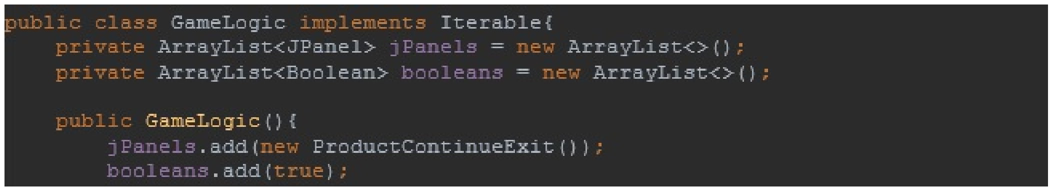
6.7 FactoryA, FactoryB ...

Class `FactoryX` follows the **Abstract Factory** design pattern - see details below. Essentially, they import images and set default x and y coordinates. This class allow the creation of multiple identical products. Note that some products such as text information are too simple and are not created in factories.

TODO To fully comply with the **Factory Design Pattern** each `FactoryX` should be a subclass of the super class `Factory`, which is not the case.

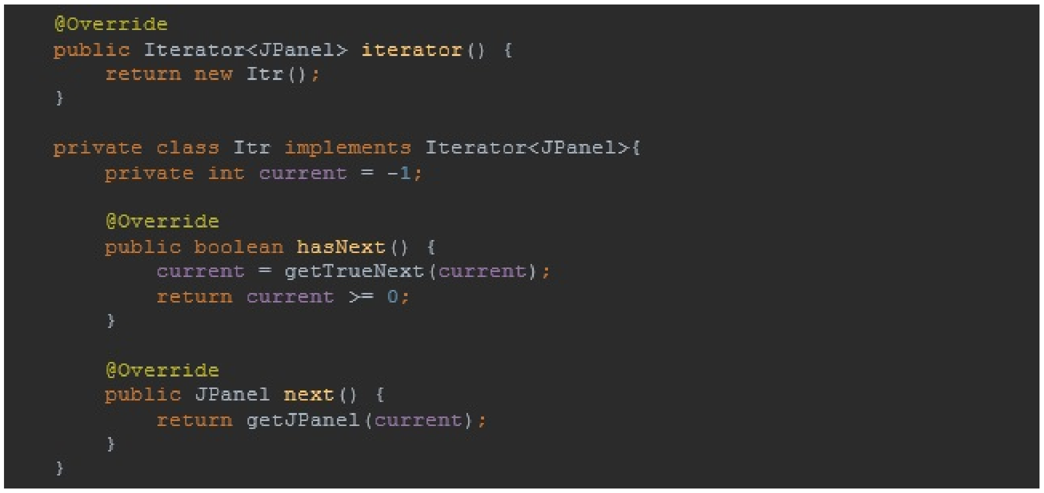
7 Iterator (fully implemented)

This design pattern allows a simple scanning of all available products anywhere in the program.



```
public class GameLogic implements Iterable{
    private ArrayList<JPanel> jPanels = new ArrayList<>();
    private ArrayList<Boolean> booleans = new ArrayList<>();

    public GameLogic(){
        jPanels.add(new ProductContinueExit());
        booleans.add(true);
```

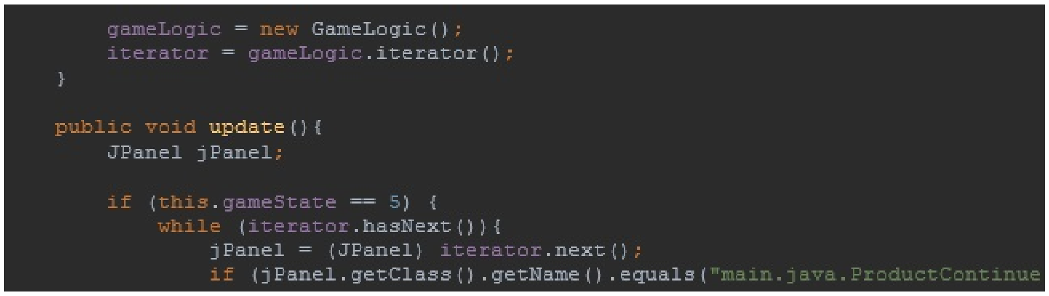
Class `GameLogic` implements `iterable` to produce `iterator` that iterates over a list of `JPanels` while skipping those that are not active as indicated in the second list of `Booleans`.



```
    @Override
    public Iterator<JPanel> iterator() {
        return new Itr();
    }

    private class Itr implements Iterator<JPanel>{
        private int current = -1;

        @Override
        public boolean hasNext() {
            current = getTrueNext(current);
            return current >= 0;
        }

        @Override
        public JPanel next() {
            return getJPanel(current);
        }
    }
```

As required, this class implements method `iterator()` that returns a class of type `iterator`, which here is named `Itr`. Class `Itr` has two methods: `hasNext()` and `next()`. The first ensures there is a next item and the second returns that next item.

Available examples for this pattern never tell you that it is important to have variable `current` to track the progress of the scan. Also these examples perform a single scan, which is not the case in this game. For multiple use of the `iterator` it is necessary to reset the `current` at the time when `hasNext()` gets false.



```
        gameLogic = new GameLogic();
        iterator = gameLogic.iterator();
    }

    public void update(){
        JPanel jPanel;

        if (this.gameState == 5) {
            while (iterator.hasNext()){
                jPanel = (JPanel) iterator.next();
                if (jPanel.getClass().getName().equals("main.java.ProductContinue
```

This design patterns allows scanning all available items anywhere in the code. The procedure involves creating a copy of class `GameLogic` and using its iterator's two methods in a loop, i.e., `while(iterator.hasNext())` then do something with `next()`. The class that uses the iterator knows nothing about the nature of the items, i.e., `JPanel`, `JButton` or `JLabel`, or how these items are stored.

8 Strategy (fully implemented)

In the original design, class `ProductB` had a long method with several helper methods to calculate the next position of the embedded image.

Private attributes included x and y coordinates, increment (speed), direction vector measured in gradients, and the coordinates of the target.

The **Strategy** design pattern asks us to place the calculation of the new position in a separate class so that `ProductB` can have different methods to calculate trajectories and a given trajectory can be reused in other `Products`.

Solution. Class `ProductB` creates a copy of `TrajectoryB` as `t`, uses its method `t.nextPosition(x, y, vector, ...)` to calculate the next position, and updates its attributes with `t.getX()`, `t.getY()`, etc.

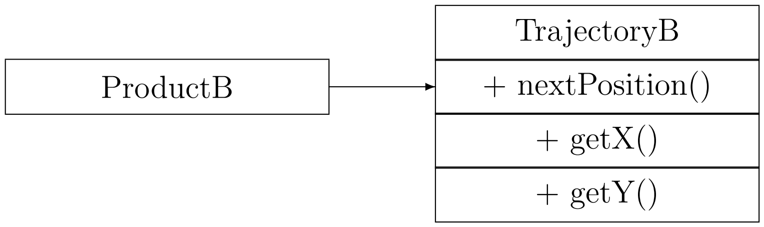


Figure 2: Strategy design pattern

9 Abstract Factory (party implemented)

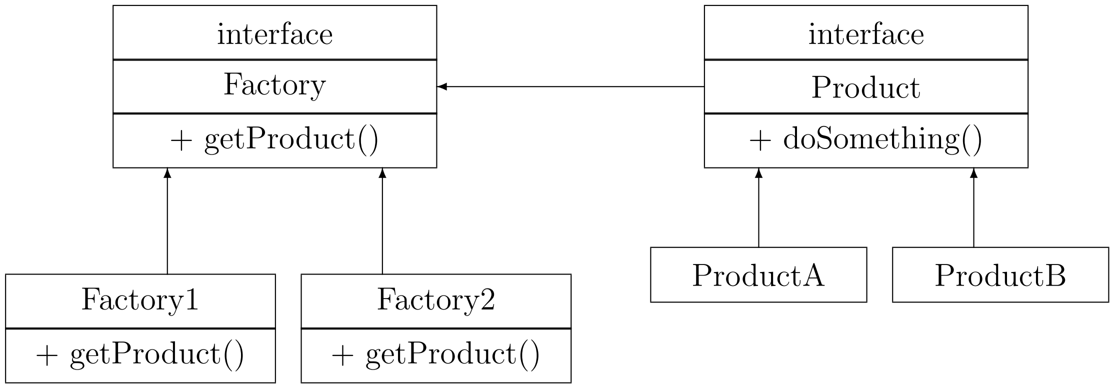


Figure 3: The factory method as a design pattern

The **Factory Method** is a design pattern for creating products (objects) of similar types.

According to this method the products are created by subclasses of an interface that describes generic behaviour of the subclasses such as getProduct.

The main() method in a separate class will actually return the product. The method examines the input ad determines which type of product needs to be created. Through a switch, it instantiates one of the subclasses that implement product creation. Finally, it returns whatever one of this subclasses returns.

The advantages of the Factory Method include the ability to modify how products of specific types are created by making changes to individual subclasses that create products without making changes to the main() method. It also possible and easy to add new product creating subclasses.

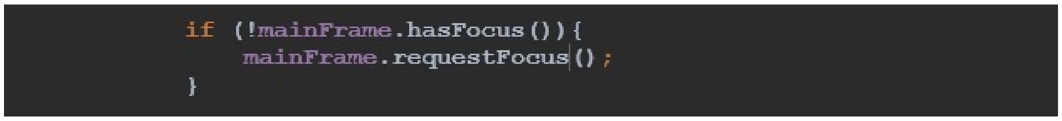
This method often features an interface for the product itself. This interface is then implemented by the product creating subclasses.

The anti-pattern, that is the code that benefits from the Factory Method, can be spotted the presence of multiple classes each of which creates products of similar types.

10 Bugs and fixes

10.1 Lost Focus

Problem. `JFrame` becomes unresponsive.
Solution. When `JButton` is displayed, it gets the focus, which is never returned to `JFrame`. This can be fixed with just two lines.

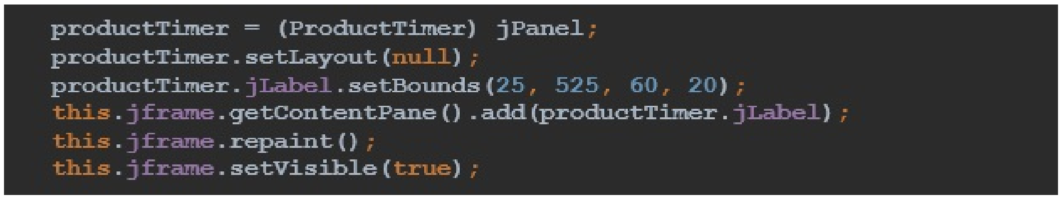


```
if (!mainFrame.hasFocus()){
    mainFrame.requestFocus();
}
```

10.2 Ignored Coordinates

Problem. It is difficult to control the position of `JButton` and JLabel.
Solution. At run time, access the components of `JPanel` through downcasting. Make sure to set the layout to `null` because without this statement changes will not be recognized. Use `setBounds` to give the components their

new positions. Then, add to `JFrame` only the components. This process needs to be repeated for each individual component.



```
productTimer = (ProductTimer) jPanel;
productTimer.setLayout(null);
productTimer.jLabel.setBounds(25, 525, 60, 20);
this.jframe.getContentPane().add(productTimer.jLabel);
this.jframe.repaint();
this.jframe.setVisible(true);
```