CSC207: Design Document

Edward, Terry and Yan

December 8, 2021

0 Content

1 Overview and Specification

Missile Mayhem is an arcade style game where a user moves a character (Pilot) to avoid incoming missiles. The user controls Pilot with arrow keys. Pilot has 11 lives. Missiles are moving on their own and chase Pilot with built in navigation intelligence. The user may change the username, toggle between light or dark themes, pause and restart, exit at any time. The game

over screen displays the current score and the top five scores. The score is the number of seconds Pilot has survived.

2 Diagram

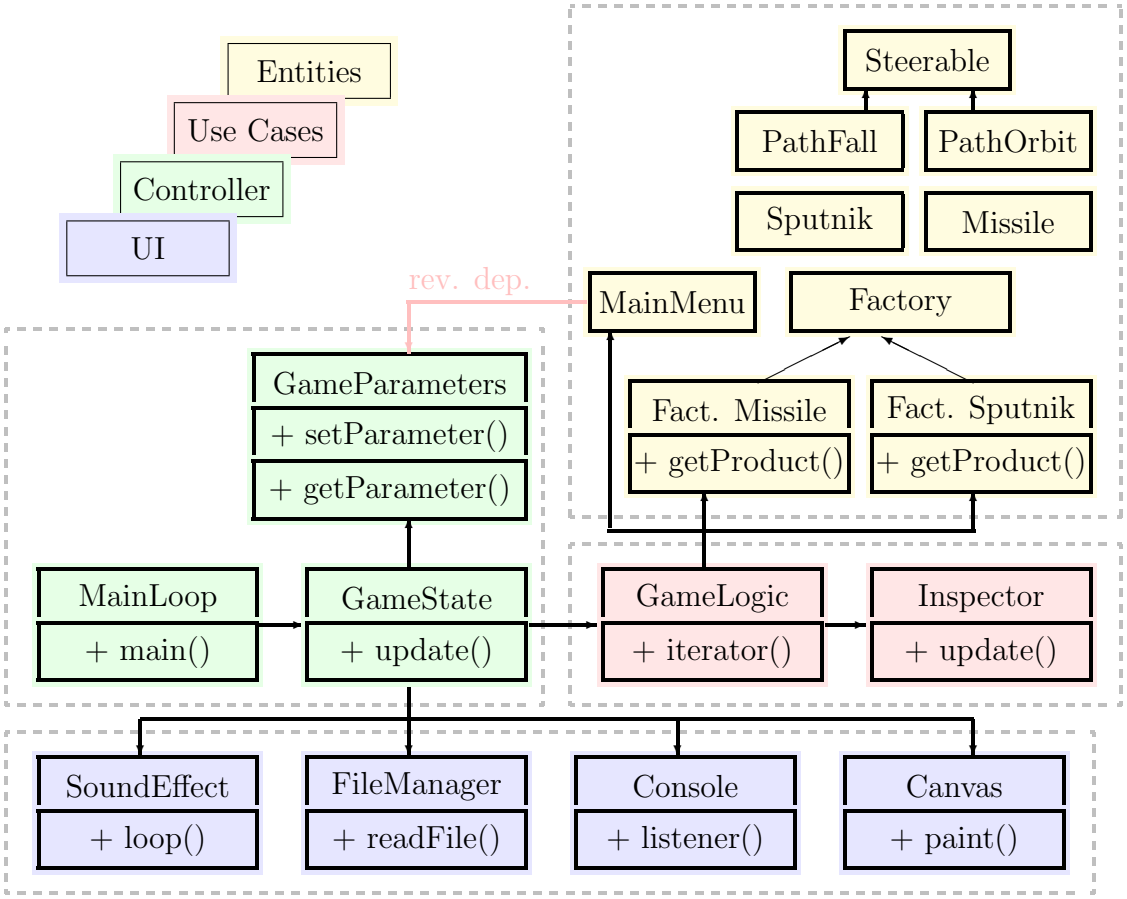

Figure 1: Missile Mayhem packages and calling dependencies.

3 Walk through diagram

Package UI (files and devices)

This package consists of four classes. **SoundEffect** contains short audio clips; **FileManager** writes and reads to external text files; **Console** listens

to keystrokes; and **Canvas** paints `Entities`. All these classes communicate with **GameState**.

Package Controller

The `main()` method is included in **MainLoop**, which is the start point of the program. **MainLoop** sends every 18 milliseconds an update call to **GameState** and terminate the program when the `gameState == 9`.

GameState initializes **GameParameters** and all UI and Use Case classes, with which it communicates through strings, integers and booleans. It keeps track of `gameState`, which can be 0 (game in progress), 5 (start screen), 6 (pause), 7 (new username), 8 (final screen) and 9 (exit).

GameParameters is a board where `Entities` record game events. It provides **GameState** with information to change `gameState`.

Package UseCases

GameLogic creates arrays of `Entities` for light and dark themes and generates an iterator to scan all active `Entities`.

Inspector examines active `Entities` and inject into them a reference to **GameBoard** so that they can report their properties and events there.

Package Entities

`Entities` are subclasses of `JPanel` and come in three varieties: dynamic objects that the user can control (Pilot), dynamic objects that the user cannot control (e.g., Sputnik), and stationary screen that wait for the user's input. Dynamic objects are created in `Factories` that are subclasses of class **Factory**. Factories include in movable objects algorithms for calculating their movement. There are three classes that contain navigation algorithms: **PathOrbit** for simple chasing of a target, **PathOrbitFaulty** for a complex algorithm that includes randomly generated malfunctions, and **PathFall** for generating straight path once the object enters the visible part of the screen. All these algorithms are subclasses of interface **Steerable**.

4 SOLID

Single-responsibility principle

Classes have singular responsibilities. Class **Canvas** adds `jPanels` to the `jFram` and paints them. Class **SoundEffect** loops the audio clips. Class **GameLogic** maitains the `iterator`, making sure it contains only active `jPanels`. There are exceptions. Class `FileManager` both writes to and reads from files. It would be possible to split these responsibilities into two classes but that would result in an unnecessarily more complex code.

Open–closed principle

Classes are open for extensions. For example, it would be fairly easy to create new `Entities`, i.e., new moving objects, new stationary screens, new navigation strategies, without a need to modify the rest of the code. Similarly, it is possible to add new code to **GameState** to handle additional stages of the game. Modifications are difficult. For example, to introduce light and dark themes, it was necessary to modify the code in many classes. However, the new code makes it now easy to extend it with additional themes.

Liskov substitution principle

Class **GameLogic** treats `Entities` as the base class. It creates arrays of `jPanels` and populates the array with `Entities`, which are subclasses of `JPanel`. This works fine with class **Canvas**, which paints them as `JPanels` without knowing their specifics as subclasses of `JPanel`. This does not work too well with class **Inspector**, which uses down casting to access the methods specific to individual `Entities`. This issue could be solved with an interface that defines the methods implemented by `Entities`.

Interface segregation principle

Interface **Steerable** and its implementation in various navigation algorithms is a good example of making sure that classes do not depend on methods they do not use. However, class **Console** is in a clear violation of this principle as it depends on multiple key listener methods but uses only two. Perhaps, to address this issue, a custom listener limited to key pressed could be written.

Dependency inversion principle

Some of `jPanels` contain key listeners. This of course creates an undesirable dependency because high level `Entities` depend on the low level UI and the controller which they need to receive and report user input. This issue was resolved with reversed dependency. Class **GameParameters** is passed to `Entities` as a parameter in a method. `Entities` thus can record they states and events to **GameParameters** without a need to create this class.

5 Clean Architecture

Four layers

As shown in the diagram, classes are packaged according to the four layers of Clean Architecture. Package **Entities** represents the domain layer and includes rules such as navigation strategies and entities such as `Sputnik`. They are the most abstract and stable components that do not depend on other classes. Package **UseCases** represents the application layer and contain classes responsible for implementing the game's behavior, e.g., the order of at which `jPanels` are displayed. Here the background is always the last `jPanel` while the stack of `jPanels` can be limited sometimes to a single stationary screen. Package **Controller** represents the adapter layer. In this layer **GameState** uses string, integers and booleans to communicate with the application layer through **GameParameters** and the external components. Package **UI** represents the infrastructure layer whose classes are responsible to take the user's input, write and read to external files, and display the game events through visual window and sound effects.

Dependency rule

The program adheres to the principle that dependencies point inwards and that the inner layers are designed not have any knowledge regarding the outer layer. This principle required the implementation of inverted dependency to enable some `Entities` to communicate with the controller.

Abstraction principle

Packages **Entities** and **UseCases** contain game logic and game entities that are the most abstract. Package **Controller** is less abstract and contains the concrete implementation of the rules and entities. Package **UI** is the most concrete and implements details of what is presented to the user and what user input is accepted.

6 Design Patterns

Strategy

Initially movable objects included algorithms to calculate their next positions. These algorithms were similar to one another and have been moved to separate classes such as `PathOrbit` that implemented interface Steerable. The result is that, in order to create a new movable object, the factory is requested to produce a specific object with a specific navigation strategy which is passed to the factory as a parameter. The moving object does not know which navigation strategy has been assigned to it. (Also see Section 14.)

Factory

The program implements the Factory Method design pattern. Movable objects are created in factories that have only one method, i.e., `getProduct()`, and are subclasses of an abstract class **Factory**. Since the program also features light and dark themes, there is an opportunity to implement the Abstract Factory design patterns, where two factories, light and dark, produce two families of products. (Also see Section 15.)

Iterator

Class **GameLogic** assembles all jPanels into two, light and dark, arrays. The `Interator` enables other parts of the program to scan only those jPanels that are active in a given stage of the game. (Also see Section 13.)

Facade

Class **GameParameters** provides class **GameState** with a simplified interface to access to events generated by stationary `jPanels`. It is not necessary for **GameState** to know the specifics how these events are generated. Some of the events, e.g., "exit" are common to several `jPanels` while others, e.g., "username", are specific. Therefore, **GameParameters** provides a simplified interface to a set of classes.

7 Accessibility Report

1) Features addressing Universal Design principles

Equitable Use. The program uses large fonts with bright colors that contrast with the background. For example, the game time is displayed in cyan on black. The score is displayed in bright red on black or white. Some of the labels are displayed in two languages: English and Chinese. The program also uses sound effects and bright visuals (explosions) to better inform the user about the game progress. The game ends with a cheerful clip to leave all users happy.

Flexibility in Use. The program allows users to toggle between light and dark themes. There might be some users who for various reasons such as being tired or being visually impaired might prefer one of these themes.

Simple and Intuitive Use. The game requires very little technological knowledge or prior gaming experience. At minimum, the game requires the user to be able to click on the start and exit buttons. Other operations are very simple and are limited to entering the username, clicking on other buttons and reading a short info screen.

Perceptible Information. The game presents information about the game progress in several redundant forms: pictorial and auditory. Text and moving objects are displayed in bright colors on a contrasting background in order to effectively communicate with users regardless of ambient conditions or the users' sensory abilities.

Tolerance for Error. The game is free from programming errors as far as

we know through test cases. The user input text box displays the previous username and will not accept input shorter than two characters or greater than sixteen characters. If the input is not a valid character count than an error will be displayed in the window prompting the user with clear instructions. However, the input has no limit on the text length and the use of special characters.

Low Physical Effort. There is little physical effort required to operate the game. The game is not entirely engaging and there is little probability the user will suffer from fatigue. However, not everyone may be capable to operate a game designed for a desk top and a future version might be adopted to portable devices.

Size and Space for Approach and Use. The game is played on a single screen and all buttons and input fields are displayed on that screen. There is a problem, however, because a mouse is required to click on these button and the input field. This issue can be addressed with focus. When the input screen is presented, the focus should be on the the input field so that the user can start typing right away. Similarly, a default button can be put into focus so that the user will only have to press `Enter` instead of clicking. Other buttons could be navigated to through `Tabs`.

2) Audience

This game could be marketed to children or adults with limited gaming experience. In addition the target audience could include people in the Emergency Room who wait long hours to see a doctor or to receive treatment. The game would create a nice distraction from their discomfort of their situation. Finally, the game may be of interest to Computer Science students interested in developing their first Java game. The components of the game are not complex and are nicely organized. It would be easy to use this game as a starting blue print.

3) Excluded Demographics

The game requires the use of a keyboard with arrow keys and a mouse. Smart phone users will not be able to play the game until the current method of input is converted to the input available on other devices. The game should

not be used by an avid gamer as its entertainment or educational value would not be appropriate to the gamer.

8 Notes on code quality and GitHub use

As of the time of writing this report, the code is completely documented internally with javadocs; warnings have been removed; and several test cases implemented. Classes with no getters, could not be tested, e.g., **MainLoop**. Similarly, classes that play audio clips or paint JPanels have no tests. GitHub has been used extensively. Pull requests and branch merging have been the primary tool. Other features like issue threads have been utilized in the final part of the project.

9 Individual contributions

These are only samples of our work.

Edward - GameParameters.java

https://github.com/CSC207-UofT/course-project-group-fifty-two-uwu/commit/e447cbb6ce1882435a64b197602d657eeb6e72e1

This class was a major improvement. It allowed us to remove `jPanels` (entities) from **GameState**. `jPanels` can now post their events to **GameParameters** which is injected into them as a parameter.

Terry - Steerable and Strategy

https://github.com/CSC207-UofT/course-project-group-fifty-two-uwu/commit/e4575923be96eb9b8cbd8f7e69f3cd48e91c3d94

Interface **Steerable** allowed us to define strategies, i.e., algorithms for navigation. These strategies are now assigned to movable objects without them knowing about these strategies. Also, it is now possible to mix strategies and movable objects.

Yan - Inspector

https://github.com/CSC207-UofT/course-project-group-fifty-two-uwu/commit/c0fd6a8691639ddb35be89dbf1e7e1c4a7dac416

Class **Inspector** allowed us to enforce the Single Responsibility Principle. It removed the interactions with `jPanels`' methods away from class **Canvas** which is now responsible for just painting `jPanels`. This was also an improvement to the Clean Architecture as these two classes belong to entirely different layers.

10 Weekly Update - December 6, 2021

Clean Architecture (Yan)

Class `GameState`, which is in the Controller Layer, no longer manipulates stationary `JPanels`, which are Entities. `JPanels` are now updated by `Inspector`, which resides in the Use Case Layer.

10.1 Accessibility (Edward and Yan)

Users can now select from two themes: `Light` or `Dark`. Several new dark `Entities` have been created. Factories have now two variations for each theme.

10.2 SoundEffects(Terry)

Class `SoundEffect` has been implemented. It plays `wav` clips when the game stage changes or when there is a collision detected.

10.3 Single responsibility Principle

Class `Canvas`, which belongs to the UI Layer, is only now responsible for gathering and displaying `JPanels`. A new class `Inspector` has been created to update the attributes of `JPanels`.

11 Weekly Update - November 29, 2021

11.1 Packaging & Dependency Injection (Yan)

All classes are now in one of four folders corresponding to four layers: Entities, Logic, Controller, and UI. This was a major effort because of unexpected dependencies created by `JPanels` (products) since they contain key listeners. A solution was found in **Dependency Injection**. At runtime a `JPanel` gains access to `GameParameters` that is passed to it as a parameter.

11.2 Interface (Edward)

To solve the issue of products with key listeners, class `GameParameters` was implemented as an interface between `GameState` and `JPanels`. It is simply a list of various parameters describing the state of the game. The advantage of this interface is that `JPanels` update `GameParameters` without a need or knowledge how these update affect the program.

11.3 Serialization (Terry)

Class `GameBoard` has been created to store and fetch game users and their scores. A text file contains a list of two fields separated by a colon. Method `write()` records the current score if higher than an existing score. Method `read()` reads the text file and creates an `ArrayList` of tuples which are of type `Tuple` and contain a string for the username and an integer for the score. Class `GameBoard` also contains a method to sort the tuples by the score. This will allow the program to display five top gamers at the end of each game session.

12 Weekly Update - November 22, 2021

12.1 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.

12.2 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 four phases: 0 (in progress), 5 (start), 6 (pause/menu), 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.

12.3 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.

12.4 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`.

12.5 GameLogic

Class `GameLogic` is where the user cases are implemented. It creates a list of `JPanels` 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.

12.6 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.

12.7 FactoryA, FactoryB ...

Class `FactoryX` follows the **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** a super class `Factory` needs to be created. Also, `ProductX`s need to be created by method `getProduct()` rather than `subclass`es.

13 Iterator

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 `JPanel`s while skipping those that are not active as indicated in the second list of `Boolean`s.

```
@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.

14 Strategy

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.

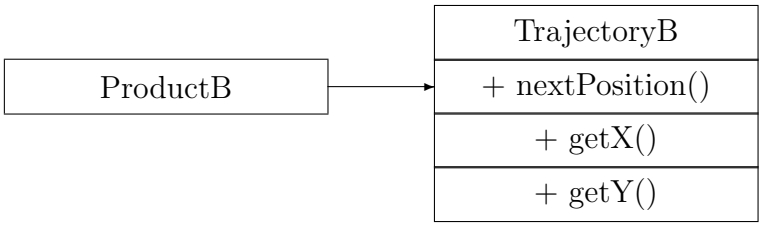

Figure 2: Strategy design pattern

15 Factory Method

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.

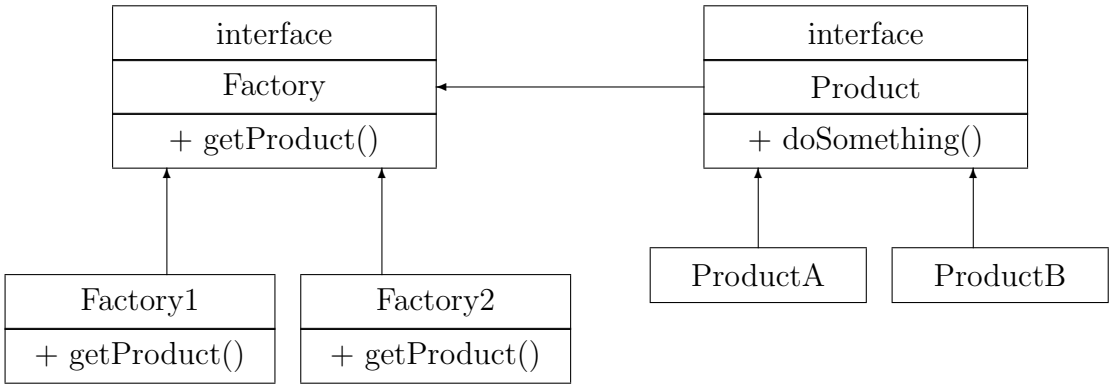

Figure 3: The factory method as a design pattern

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.

16 Bugs and fixes

16.1 Lost Focus

Problem. `JFrame` becomes unresponsive.
Solution. When `JButton` is displayed, it gets the focus, which is never re-

turned to `JFrame`. This can be fixed with just two lines.

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

16.2 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);
```

16.3 LIFO system of JPanels

Jave `Graphics` paints `JPanels` in the LIFO order. To get dark or light backgrounds it is necessary to add these backgrounds as the last `JPanel` to `JFrame`.

16.4 Maps cannot be sorted

Even when elements are put into a map in an ordered fashion, the map will rearrange them by `key` within a tree structure. The order by `value` cannot be restored. Our solution was to abandon `Map` and create a custom `ArrayList<Tuple>`.

16.5 JFrame cannot manage multiple components

Both `JFrame` and `jFrame.getContentPane()` have the same methods to add, revalidate, remove, and repaint. Experimentally, we have established a sequence that works.

```
jFrame.setVisible(true);
jFrame.getContentPane().removeAll();
jFrame.getContentPane().add(jPanel);
jFrame.getContentPane().add(jPanel);
jFrame.getContentPane().revalidate();
jFrame.revalidate();
jFrame.repaint();
```