OOP Final Project

PepeClicker

Pierre Corazo Cesario
[REDACTED]

PRG2104

July 16 2021

Table of Contents

Introduction

Task

Students of PRG2104 in the April 2021 semester at Sunway University were tasked with a final project to demonstrate their inheritance and polymorphism programming knowledge. This was to be done in the Scala language with the use of the ScalaFX graphical user interface (GUI) library to create a GUI application. Suggested standalone systems were:

- (Scientific) calculator
- Library Management System
- To-do list
- A personal game
- Etc.

The requirements of the system should have at least four use cases or functions. For example:

- Check In
- Check Out
- Search Book
- View Book

Of course, since this is an Object-oriented programming (OOP) class, students should also demonstrate OOP principles when implementing the system to achieve elegant design.

Pepe the Frog

The proposed system for this project is based on the popular cookie clicker game but with styling in the form of the popular Internet meme “Pepe the Frog”. In a cookie clicker game, players simply click on a cookie on the screen and gain a cookie each time it is clicked. Accumulated cookies can then be used as currency to buy power-ups. Power-ups can change gameplay in various ways, but the classic power-ups typically increase the amount of cookies earned per click and grant cookies passively. Since the system takes styling from Pepe the Frog, various elements were changed to fit the meme and will later be discussed.

Gameplay Overview

Upon startup, users are greeted with a login page with a ‘PepeStare’ image to introduce the Pepe the Frog theme. Users can login with their username and password if their appropriate save file is present in the resources/saves folder.

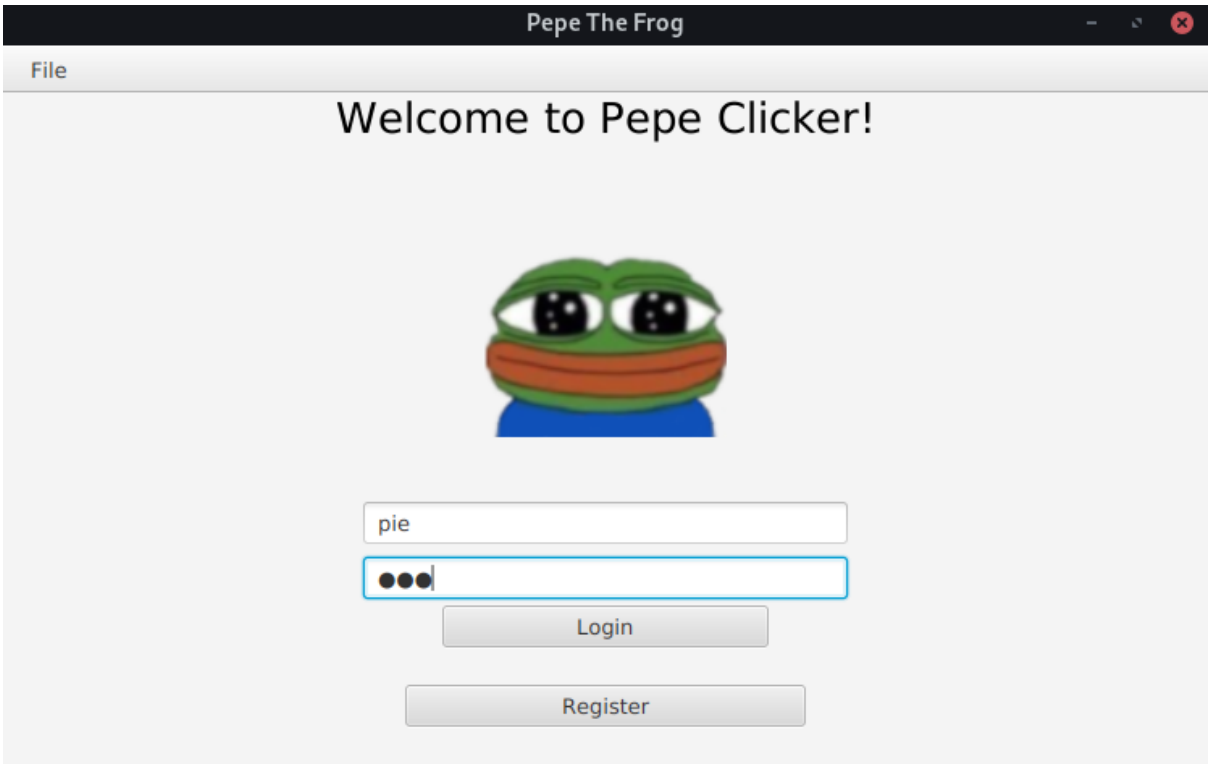

Fig. 1. Login page.

An appropriate message is displayed depending on the errors associated with failed logins.

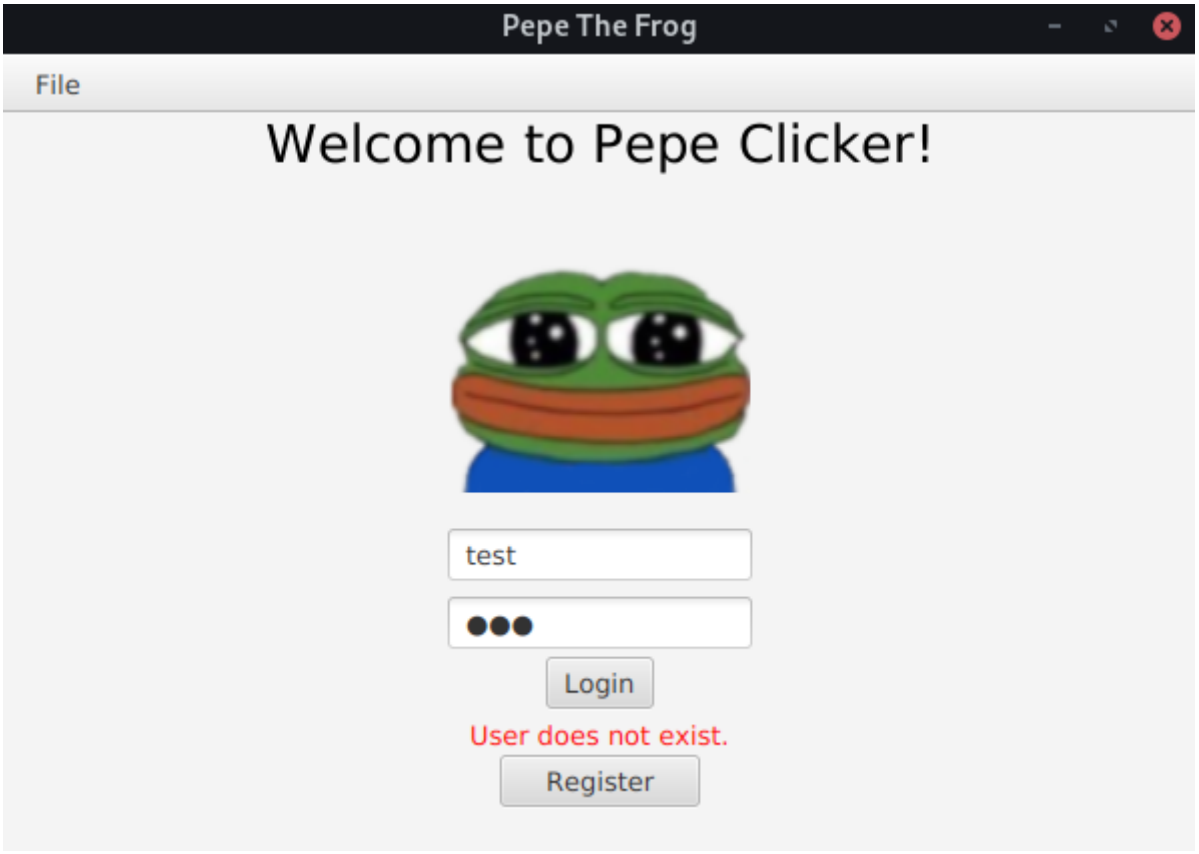

Fig. 2. Example of error message in login page.

If a username is not found, users can go to the register page by clicking on the register button, then they can return to the login page to begin playing the game.

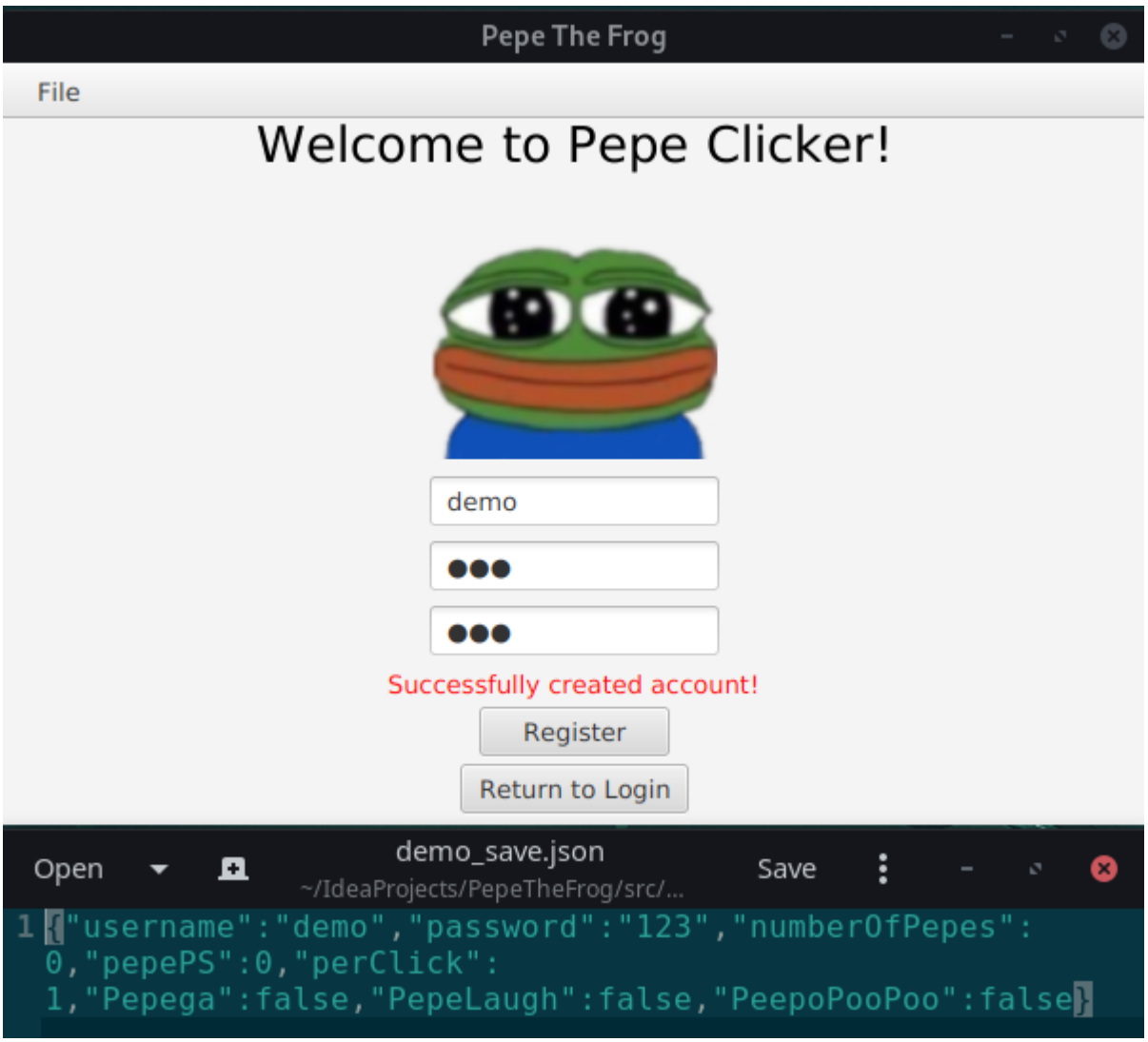

Fig. 3. Demonstration of new account and its newly created save file.

Once logged in, the user's progress will be loaded in. Players can begin clicking on the Pepe on the screen to earn points and buy power-ups.

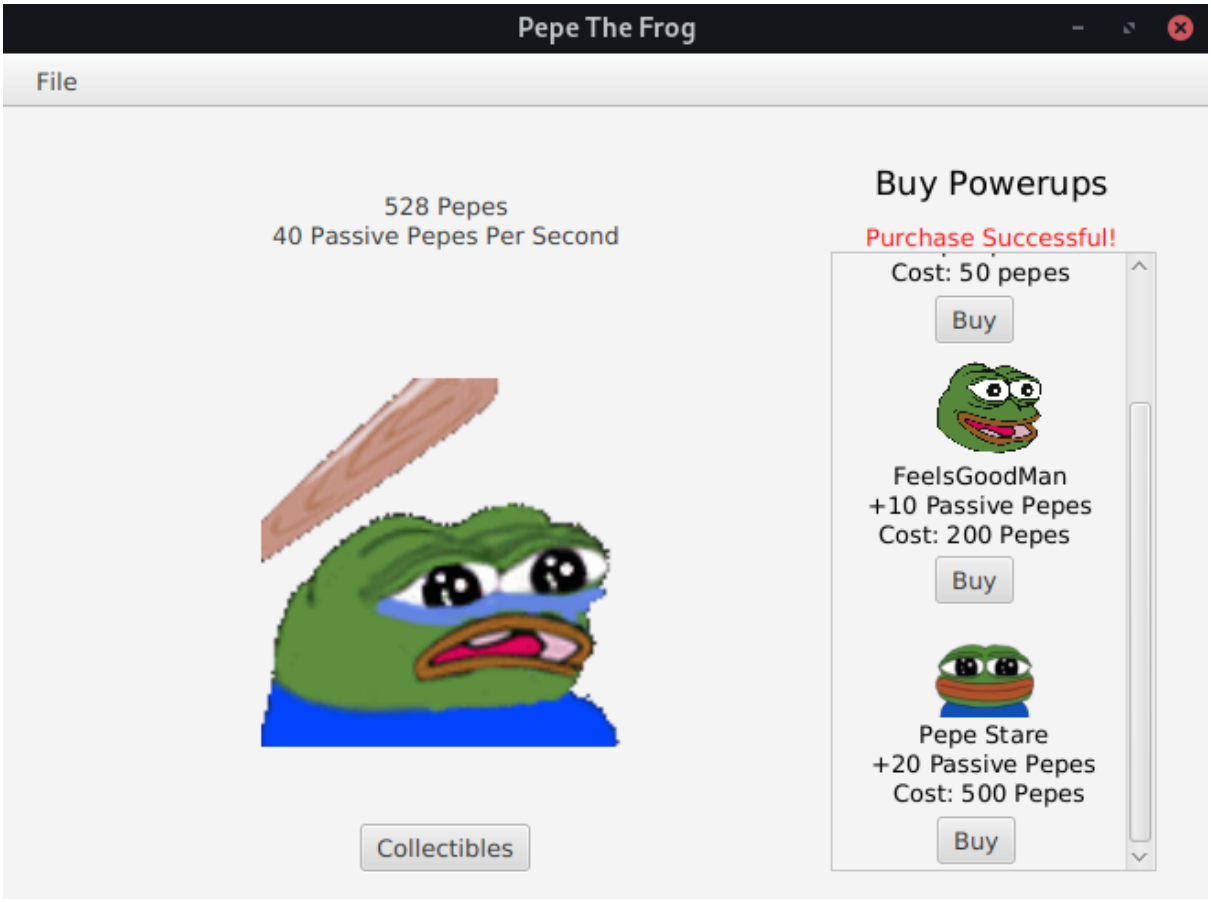

Fig. 4. Demonstration of the Game scene.

If they have accumulated enough Pepes, they can navigate to the Collectibles screen through the Menu bar to buy expensive Pepe Collectibles.

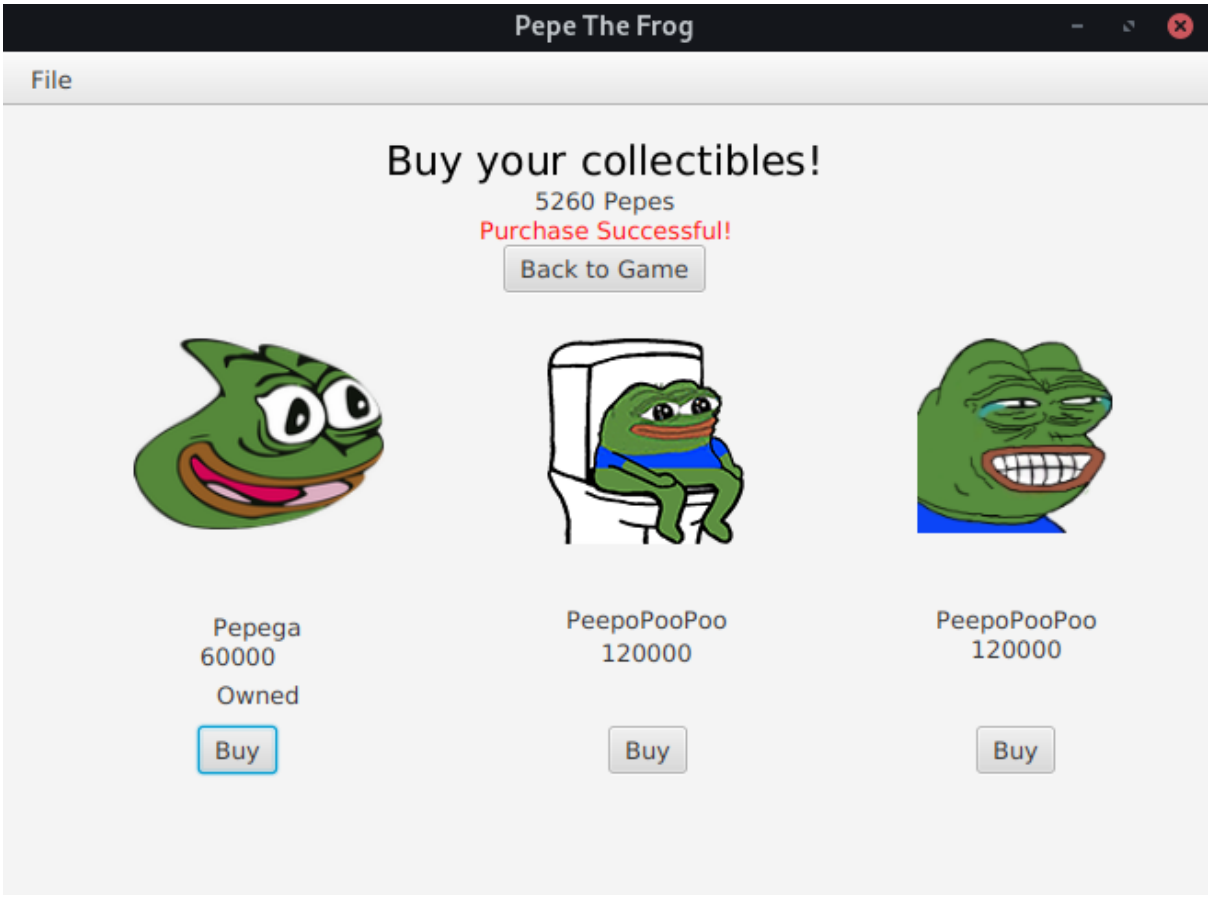

Fig. 5. Demonstration of collectibles shop.

If the user wishes to save, log out or close the application, the functions can be accessed through the Menu bar and the appropriate functions will run.

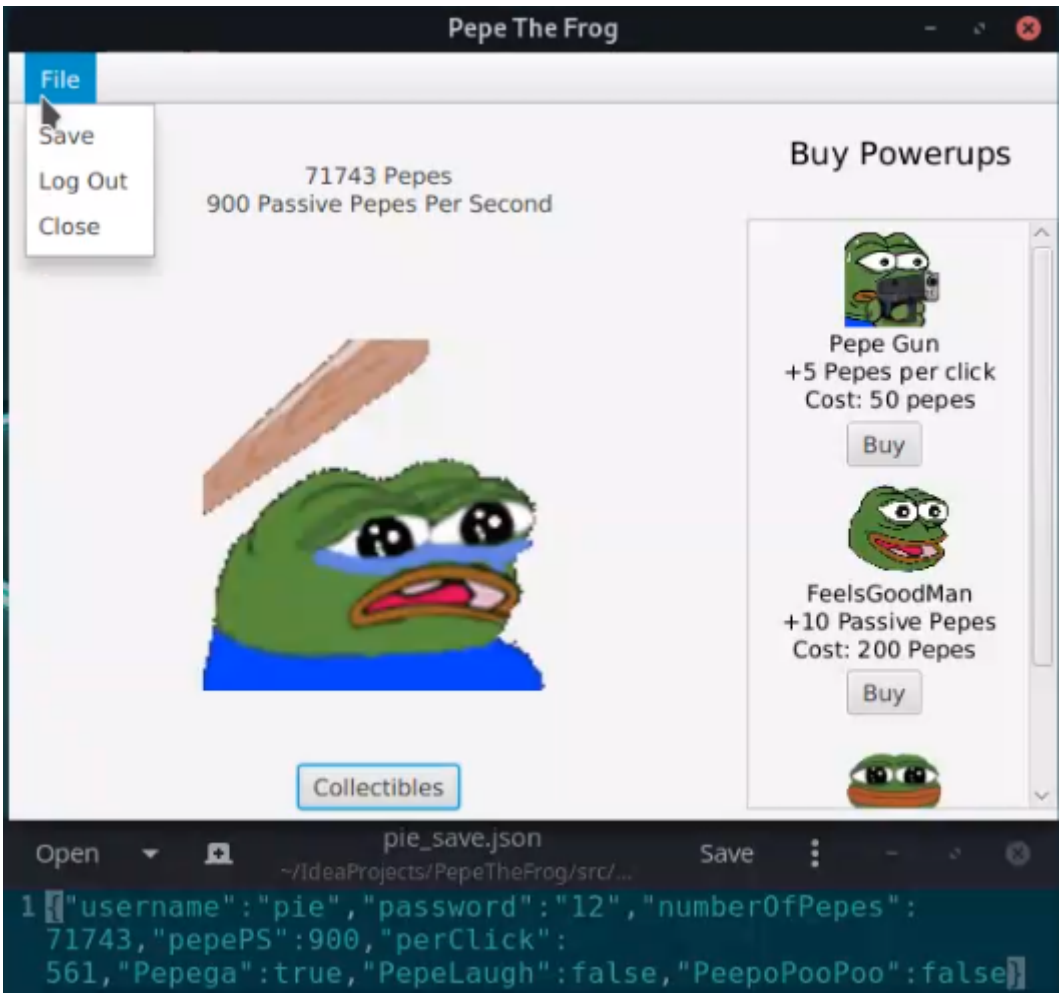

Fig. 6. Demonstration of save function in menu bar.

Features

The scope of the project first has to be defined before it can begin. The proposed features include the following:

- Login
- Logout
- Save progress
- Gain Pepe currency on click
- Buy power-ups
    - Boost gained Pepes per click
    - Earn passive pepes per second
- Buy collectibles

Project Requirements

The proposed features have been set and now the requirements of the project can be listed.

Requirement	**Choice**
Programming language	Scala
Integrated Development Environment (IDE)	Jetbrains IntelliJ
Graphical User Interface (GUI) designer	Scene Builder
JSON serialisation library	uPickle and OS-Lib by Lihaoyi [1], [2], [3]
Software design pattern	Model-view-controller

Tab. 1. Project requirements and their choices.

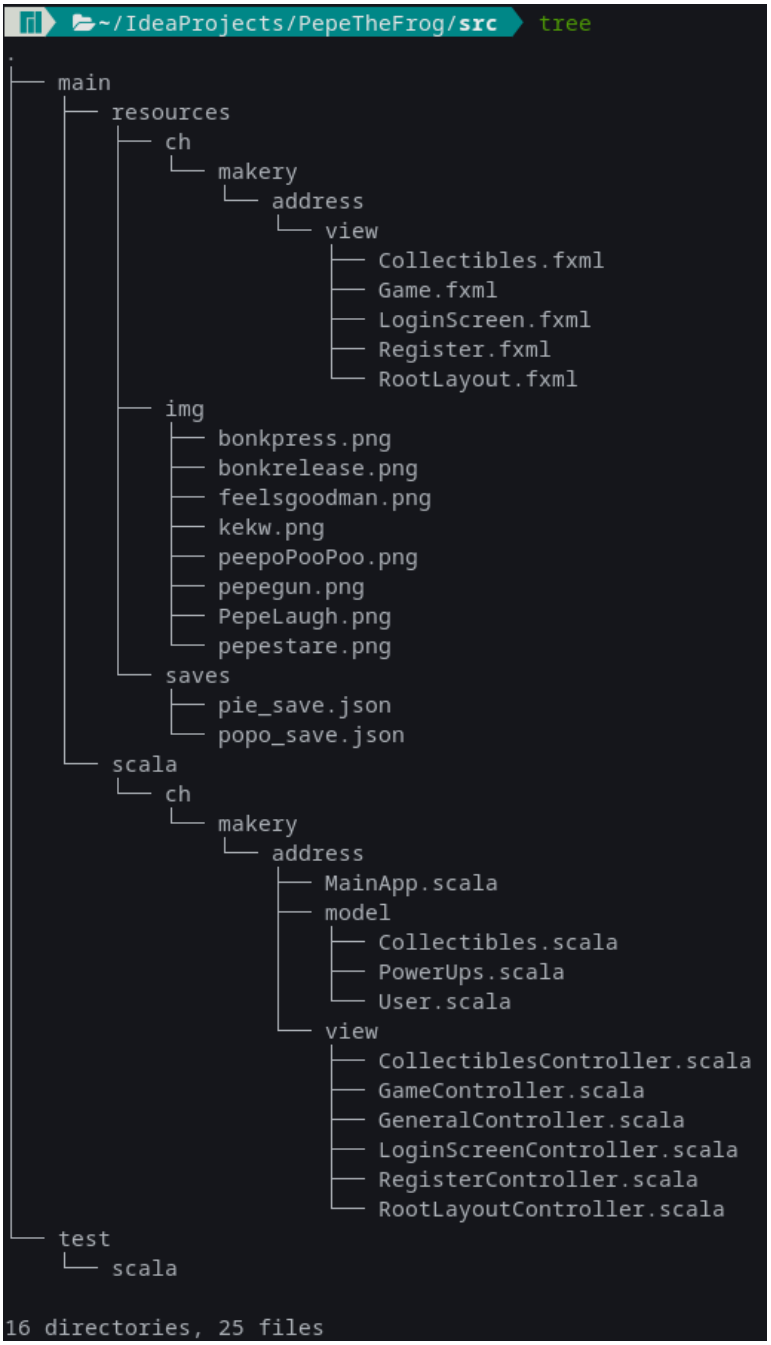

Fig. 7. Project directory structure.

Main

A standard SceneBuilder FXML MainApp skeleton was used to construct the stages and scenes in the GUI. This skeleton was taken from the OOP tutorials which, in turn, was taken from the supplied document.

```
object MainApp extends JFXApp {
  // transform path of RootLayout.fxml to URI for resource location.
  val rootResource = getClass.getResourceAsStream("view/RootLayout.fxml")
  // initialize the loader object.
  val loader = new FXMLLoader( fxml = null, NoDependencyResolver)
  // Load root layout from fxml file.
  loader.load(rootResource);
  // retrieve the root component BorderPane from the FXML
  val roots = loader.getRoot[jfxs.layout.BorderPane]

  // initialize stage
  stage = new PrimaryStage {
    title = "Pepe The Frog"
    scene = new Scene {
      root = roots
    }
  }
```

Fig. 8. Standard FXML stage setup.

Below that, there are various functions for switching between scenes. The problem with all these functions is that they share much of the same code, aside from the names of their appropriate FXML files. Thus, the code was separated and turned into a function, with a String as an argument for the FXML's file path. With this change implemented, the code becomes much more readable and maintainable.

```
  // redundant code for scene switching
  def setupScene(location: String): Unit = {
    val resource = getClass.getResourceAsStream(location)
    val loader = new FXMLLoader( fxml = null, NoDependencyResolver)
    loader.load(resource);
    val roots = loader.getRoot[jfxs.layout.AnchorPane]
    MainApp.roots.setCenter(roots)
  }

  // various scenes go here
  // actions for display main game screen window
  val path: String = "view/"
  def showGame(): Unit = {
    setupScene(path + "Game.fxml")
  }

  def showCollectibles(): Unit = {
    setupScene(path + "Collectibles.fxml")
  }
```

Fig. 9. Simplified scene switcher.

To account for the passive Pepe income from the game, a *TimerTask* utilising the java.util library was made. Essentially, it runs a block code in the background indefinitely between set

intervals [4]. In this case, it increments the Pepe Counter by the Per Second value every single second. This passive task is also the cause of the main weaknesses of this program, as will be discussed in the Weaknesses section.

```
// passive pepes
val timer = new Timer()
val passivePepe = new TimerTask {
  override def run(): Unit = {
    User.pepeCounter += User.perSecond
    // no refresh labels
  }
}
timer.schedule(passivePepe, 1000L,  l1 = 1000L)
```

Fig. 10. Passive Pepe code block in MainApp.

Model

Collectibles

Collectibles are instantiated from the *NewCollectibles* class in the Collectibles object. NewCollectibles specify their name and cost which are then referenced by the *CollectiblesController* class to display the relevant information in the GUI.

```
object Collectibles {
  class NewCollectibles(val _name: String, val _cost: Int)

  val pepega = new NewCollectibles( _name = "Pepega",  _cost = 60000)
  val pepeLaugh = new NewCollectibles( _name = "PepeLaugh",  _cost = 90000)
  val peepoPooPoo = new NewCollectibles( _name = "PeepoPooPoo",  _cost = 120000)
```

Fig. 11. Collectibles objects.

PowerUps

The PowerUps object contains the purchasable powerups in the Game scene. PowerUps are instantiated from either the *Type1PU* or *Type2PU* class depending on whether they boost the Pepes per click or Pepes per second respectively. These classes are actually subclasses of the *NewCollectibles* class in the Collectibles object since they share some properties. PowerUps add an *ability* variable as opposed to Collectibles which do not have it.

```
object PowerUps {
  // powerups that boost clicks
  class Type1PU(_name: String, _cost: Int, val _ability: Int) extends NewCollectibles(_name, _cost)
  // powerups that boost passive
  class Type2PU(_name: String, _cost: Int, val _ability: Int) extends NewCollectibles(_name, _cost)

  val pepeGun = new Type1PU( _name = "Pepe Gun",  _cost = 50,  _ability = 5)
  val feelsGoodMan = new Type2PU( _name = "FeelsGoodMan",  _cost = 200,  _ability = 10)
  val pepeStare = new Type2PU( _name = "Pepe Stare",  _cost = 500,  _ability = 20)
}
```

Fig. 12. PowerUps objects.

User

The User object holds information about the currently logged in player in memory. It contains their username, password, their game progress and which collectibles they own. The values here change as they play the game and are typically reflected with the use of *refreshLabel()* in the appropriate scenes.

```
object User {
  var username: String = ""
  var pepeCounter: Int = 0
  var perClick: Int = 1
  var perSecond: Int = 0
  var pepega: Boolean = false
  var pepeLaugh: Boolean = false
  var peepoPooPoo: Boolean = false
}
```

Fig. 13. User objects.

Controllers

In all function implementations, properties like name, cost and ability are referenced by calling them in their respective classes. No values are hardcoded which means these functions

will automatically reflect changes if the properties are changed in the Model section.

```
} case "peepoPooPoo" => {
  if (User.peepoPooPoo.equals(false)) {
    if (User.pepeCounter >= Collectibles.peepoPooPoo._cost) {
      User.pepeCounter -= Collectibles.peepoPooPoo._cost
      // adds new collectible to user arraybuffer
      User.peepoPooPoo = true
      refreshLabels()
      showMessage( str = "Purchase Successful!")
    } else {
      showMessage( str = "Insufficient Pepes!")
    }
  } else {
    showMessage( str = s"${Collectibles.peepoPooPoo._name} already owned!")
  }
```

Fig. 14. Portion of *buyItem()* in *CollectiblesController* demonstrating OOP references.

GeneralController and *ActionController*

A general abstract controller class was first made since all of the controllers needed a *showMessage()* function to notify the user of any actions they were doing. This controller is aptly named *GeneralController* and does not use the @sfxml annotation since it is just a skeleton for the actual controllers to use. For the scenes where players buy items, another abstract class was made that subclasses *GeneralController*. This subclass, named *ActionController*, has abstract methods *buyItem()* and *refreshLabels()* and is intended for *GameController* and *CollectiblesController* to subclass and override the methods. *CollectiblesController* handles buying collectibles and switching to the Game scene. Similarly, *GameController* handles various actions such as: clicking on Pepe, buying power-ups and switching to the Collectibles scene, hence why both of them subclassing *ActionController* is justified.

```
// special abstract controller for game and collectibles controller
abstract class ActionController(mesg: Label) extends GeneralController(mesg) {
  def buyItem(action: ActionEvent): Unit
  def refreshLabels(): Unit
}
```

Fig. 15. *ActionController* abstract class code.

CollectiblesController and GameController

These controllers are for their respective scenes and include the additional functions such as incrementing and switching scenes. The following two functions that are going to be described are the more crucial components to these controllers.

buyItem()

An important implementation to note is the *buyItem()* function. Initially, the design was that each buy button for each item was assigned its own buy function. However, it was quickly noticed that this was very redundant and not OOP-like since each of the buy functions had very similar blocks of code. An attempt to refactor this was made and the final approach taken was to pass the button's *fx:id* as an argument into a single *buyItem()* function. A match expression can then be used to run the appropriate buy code depending on which button was pressed. While this is not as modular as possible, it is still an improvement over the original approach since all the buttons are directed to a single method. A similar *buyItem()* implementation was used in the *CollectiblesController* class but adapted for buying collectibles instead.

```
// runs relevant code block based on which buy button is pressed
override def buyItem(action: ActionEvent) {
  // get id of which button was pressed
  var source:Node = action.getSource.asInstanceOf[Node]
  val pu = source.getId()
  pu match { // buy relevant item based on button's fx:id
    case "pepeGun" => {
      if (User.pepeCounter >= PowerUps.pepeGun._cost) {
        User.pepeCounter -= PowerUps.pepeGun._cost
        User.perClick += PowerUps.pepeGun._ability
        refreshLabels()
        showMessage( str = "Purchase Successful!")
      } else {
        showMessage( str = "Insufficient Pepes!")
      }
    } case "fgm" => {
      if (User.pepeCounter >= PowerUps.feelsGoodMan._cost) {
```

Fig. 16. Portion of *buyItem()* function in *GameController*.

refreshLabels()

To update the labels in the scenes, such as the number of passive Pepes per second and current number of Pepes, *refreshLabels()* was made. Essentially, this function updates the

appropriate labels on the screen with the values in memory. This function was then overridden in the *GameController* and *CollectiblesController* classes since they point to different scenes.

```
// refresh current pepe count
override def refreshLabels(): Unit = {
  numberOfPepes.setText(User.pepeCounter.toString + " Pepes")
  if (User.pepega) {pepegaOwned.setText("Owned")}
  if (User.peepoPooPoo) {peepoPooPooOwned.setText("Owned")}
  if (User.pepeLaugh) {pepeLaughOwned.setText("Owned")}
}
```

Fig. 17. Overridden *refreshLabels()* function in *CollectiblesController*.

RootLayoutController, RegisterController and *LoginScreenController*

Since a login and save system was made, the values needed to be somehow stored after the program stops running. The method taught in tutorials is to implement a database but it is rather overkill for a simple clicker game like this. Thus, the chosen approach to save files is to store them into a JSON file. JSON files are formatted in such a way that they are human readable which makes them easy to work with. Seeing as the native scala.util.parsing was deprecated since Scala 2.11, a third-party library had to be used to interface with JSON files. Lihaoyi's uPickle and OS-Lib libraries were chosen to implement JSON parsing into the project and this is demonstrated in the *handleSave()*, *attemptRegister()* and *attemptLogin()* functions of the *RootLayoutController, RegisterController* and *LoginScreenController* classes respectively [1], [2], [3].

```
// attempts to login user
def attemptLogin(): Unit = {
  // checks for save file based on username
  val filename: String = unameInput.getText() + "_save.json"
  try { // try to access file
    val temp = os.read(os.pwd/"src"/"main"/"resources"/"saves"/filename)
    val parsed = ujson.read(temp)
    val uname: String = parsed("username").str
    val pw: String = parsed("password").str

    // loads values if username and password patches
    if (uname == unameInput.getText() && pw == pwInput.getText()) {
      User.username = uname
      User.pepeCounter = parsed("numberOfPepes").num.toInt
      User.perSecond = parsed("pepePS").num.toInt
      User.perClick = parsed("perClick").num.toInt
      User.pepega = parsed(Collectibles.pepega._name).bool
      User.pepeLaugh = parsed(Collectibles.pepeLaugh._name).bool
      User.peepoPooPoo = parsed(Collectibles.peepoPooPoo._name).bool
      MainApp.showGame()
```

Fig. 18. Portion of *attemptLogin()* function in *LoginScreenController*

While it is true that the user can simply edit the values in the JSON file with a text editor to add more Pepes and 'cheat', it is not too serious since using the JSON file is merely to demonstrate how Scala objects can be saved and then retrieved at a later time. If security was of utmost concern, the JSON file can be encrypted.

Applied Object-Oriented Programming Concepts

Various OOP concepts were applied throughout the program's code and include but are not limited to, code reuse, inheritance and overriding (runtime polymorphism).

Code Reuse

As mentioned in the beginning of the Controllers section and in Fig. 14, references to the Model were used and nothing was hardcoded. Furthermore, as demonstrated in various controller functions throughout all the Controller classes, if there is a block of code that is repeated and used in other sections, it is turned into a function for code reuse like in RootLayoutController's *end()* function. Another great example is the scene switcher *setup()* function in MainApp and as shown in Fig. 9.

Inheritance/Subclassing

Subclassing is also present throughout the controllers and model objects in this system. For example, *Type1PU* and *Type2PU* both extend from the *NewCollectibles* class since they share similar properties. This can be seen in Fig. 12. Furthermore, all @sfxml controllers are subclasses of either *ActionController* or *GeneralController* since many of the actual controllers shared many functions such as *showMessage()*, *buyItem()* and *refreshLabels()*. In fact, the abstract controller *ActionController* extends from *GeneralController*.

Overriding (Runtime Polymorphism)

On a similar note to the abstract classes of the controllers, abstract methods such as *buyItem()* and *refreshLabels()* are in the *ActionController* class. These methods are used by both *CollectiblesController* and *GameController* but with slight changes to them since they handle different scenes. As such, overriding those methods is a great choice to promote OOP concepts. This can be seen in Fig. 16 and Fig. 17.

Problems Encountered

@sfxml and Inheritance

The *CollectiblesController* and *GameController* classes needed functions with the same names - *refreshLabels()* and *buyItem()*. It made sense to use inheritance and overriding to adhere to OOP concepts. In this case, *GameController* would inherit from *CollectiblesController* since *GameController* needs a few more functions. However, it seemed that performing two levels of inheritance with *@sfxml* annotations was not permitted by the library. Initially, it was thought that there were issues with the syntax for all of the constructor inheritances, but the syntax was valid. After much time trying to have it work in this way, the chosen solution was to instead have another abstract class named *ActionController* subclass the *GeneralController* abstract class. Now, by having *GameController* and *CollectiblesController* subclass *ActionController*, the final outcome is the same as was first intended and still follows OOP concepts. Since *ActionController* has

abstract methods for *refreshLabels()* and *buyItem()*, *GameController* and *CollectiblesController* can each override the functions. The rest of the controllers subclass *GeneralController* since they do not need *GameController*'s special functions.

getClass.getResourceAsStream() and Images

As noted by the lecturer during the tutorials, changing the *getClass.getResource()* function to *getClass.getResourceAsStream()* allows for chosen FXMLs to be streamed through even if the project is compiled into a .jar file. However, for this project, some modifications needed to be made to accommodate for this change. There are inherently many images in this GUI application and as such, they need to be displayed properly. With the directory structure shown in Fig. 1, the default path given by SceneBuilder for images will look something like this: '../../../../img/pepestare.png'. Using this path will render the image correctly in SceneBuilder and allow the image to be shown if *getClass.getResource()* is used. However, if *getClass.getResourceAsStream()* is used, an *IllegalArgumentException: Invalid URL or resource not found* error will be thrown if the code is compiled. One can infer that this is due to how SBT projects look into the *src/main/resources* for resources by default, thus conflicting with SceneBuilder's chosen path. This is easily fixed by removing the various periods and slashes at the beginning of the path. A working path would be: '/img/pepestare.png'.

Strengths

The system displays its strengths through its ability to login, logout, register and store user progress even after the program is closed. As demonstrated in the Gameplay Overview section, users are able to create new accounts, login into existing accounts and save their gameplay while they are in the application. All of this is thanks to the use of the uJson and OS-Lib libraries to save Scala objects into JSON objects [1], [2], [3].

Weaknesses

There are two main weaknesses to the system:

- The application has to be closed through the close button in the menu.
- The current number of Pepes does not update every second.

Both of these weaknesses have to do with the Timer Task implemented in *MainApp* as shown in Fig. 8. The application needs to be closed using the menu option 'Close' as it will properly end the Task. If the usual 'X' window button is used, the GUI will close but the console will still be running in the background.

The other drawback is that with this method of updating the Pepe Counter, the *refreshLabels()* function cannot be called in the same code block since errors will be thrown. Thus, the passive Pepes earned will still register in memory, but will not show in the program until a function that runs the *refreshLabel()* function is executed.

UML Diagram

The UML diagram is a good reference to understand how the classes in a system are interconnected. Below is the UML Diagram for this Pepe the Frog GUI application.

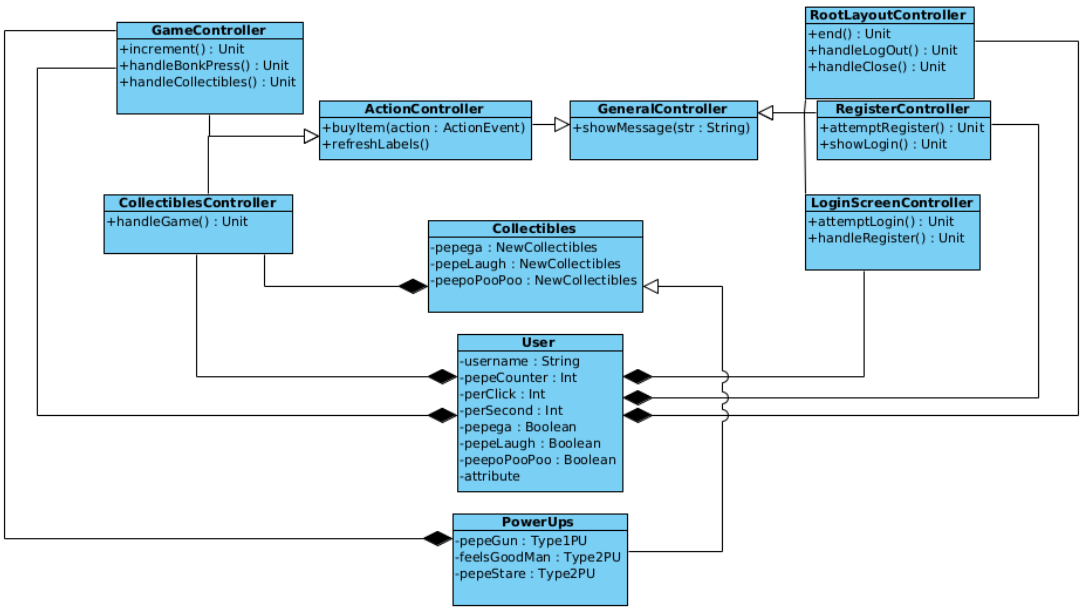

Fig. 19. UML Diagram of Pepe the Frog system.

References

[1] Lihao Yi uJson

https://www.lihaoyi.com/post/HowtoworkwithJSONinScala.html#modifying-json

[2] Lihao Yi OS-Lib https://github.com/com-lihaoyi/os-lib

[3] https://mungingdata.com/scala/read-write-json/

[4] https://stackoverflow.com/questions/25351186/run-a-function-periodically-in-scala