KeySlide

Troy Johnson, Suman Karki, Hesham Salman, Ting Li, Xiaoying Yu

Overview

KeySlide is a fast-paced game. It is self-contained and has no external dependencies: it is built with Java using the AWT and Swing frameworks, the JDBC API, and all code is custom-written (with the exception of VerticalLabelUI -- see references). The development of the game was carried out by five individuals, which required some planning and foresight with regards to development.

Introduction

This document specifies the functional and relational components of the software system. It specifies the specifics of the design and implementation of the system, and is to be read by the designers, testers, and contributors to the software system.

The software system to be produced is KeySlide. It is self-contained. It is a game with few functions: the ability to view high scores, read instructions, and play the game. The game itself has five expected input commands from the user: the up, left, right, and down arrow keys, as well as the lack of input (when requested). When the correct input is received, the game will display a symbol asking for another input. When the incorrect input is received, the game ends and displays the score to the user.

The software system will be open-sourced and freely distributed upon completion, and will provide enjoyment to those that play it.

The rest of this document contains the overall description of the software system and the specific description of its components.

Overall Description

Product Perspective

This software system is independent and self-contained into an executable JAR file. The software system should be portable, in that it can execute on any environment with JRE 1.8. Constraints on this product: The user-interface and input methods require a traditional laptop/desktop computer. The expected inputs are key-presses from the keyboard, and navigation is handled through mouse interaction.

Product Functions

The software system can read input from the mouse and keyboard and reacts accordingly to well-timed user-input. The game should be able to keep track of the current score of the user and store high-scores in a database. The game must have an instructions pane that shows users how to play. The game must operate on a timer such that the user only has a limited amount of time to produce the proper input.

Nonfunctional Requirements

The game should be simple and intuitive to play. The transition between panels should be seamless and fast. Access to the database for retrieving and inputting high scores should be seamless and fast. The code for the game should be well documented so as to encourage collaboration.

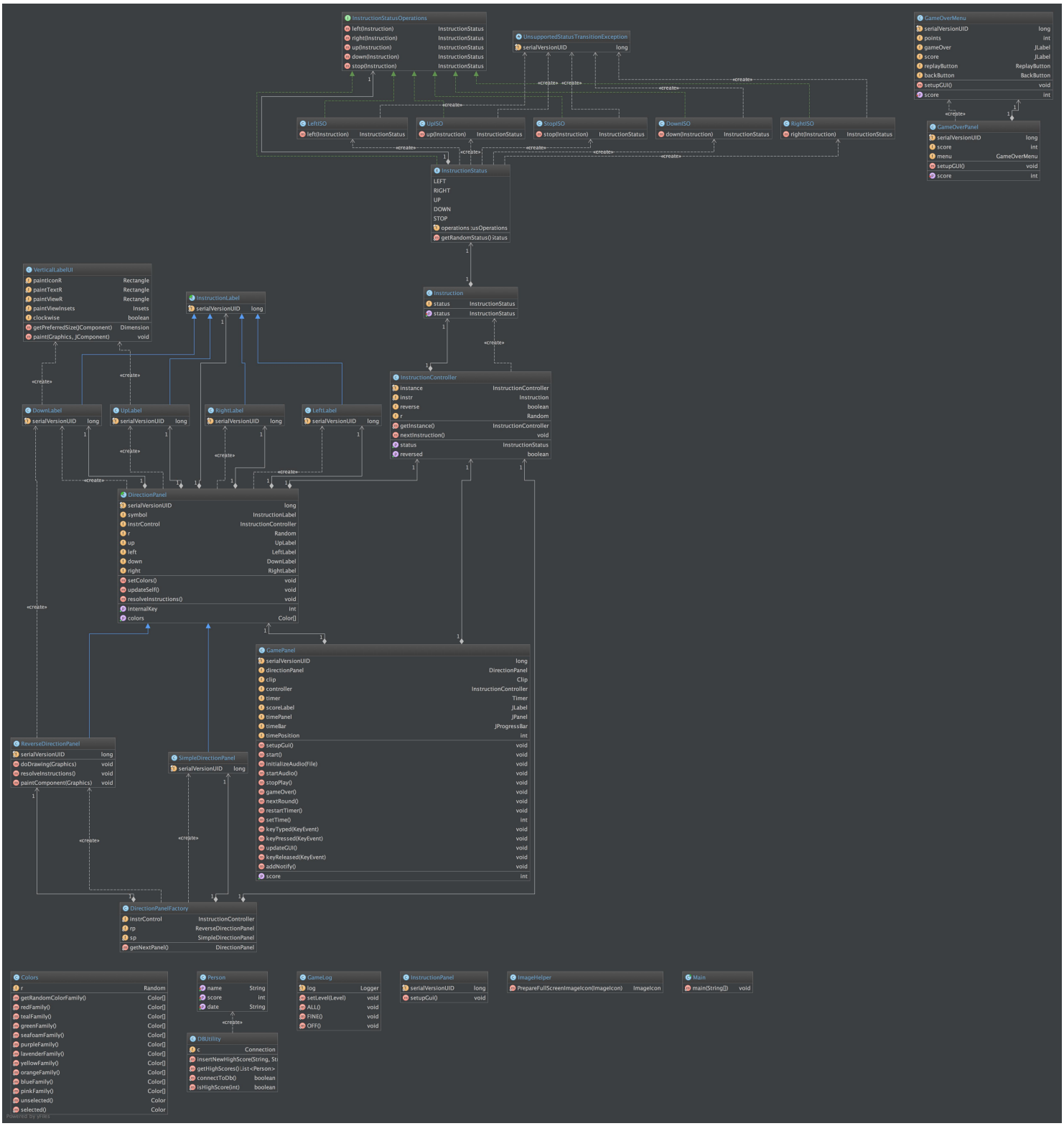
User Characteristics

The users of this software do not have to be technically inclined. The users are people with any level of computing experience or technical background. The user is not expected to enter into the program with any knowledge of its function.

Constraints

The users must also have a keyboard and mouse plugged into or integrated into their device. We are not aware of any GPU or CPU requirements. Any modern day desktop or laptop computer processor should be powerful enough to run KeySlide.

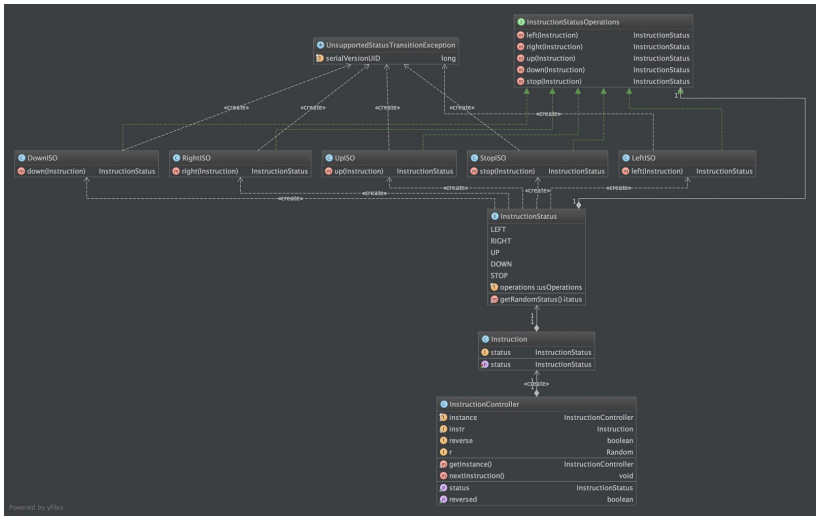
System Specification



Component Specification

Instruction*:*

The generation of new instructions is handled by a number of classes in the instruction package.



These classes follow the state design pattern, with a few key design changes. In our implementation of the state design pattern, we have opted to place an enumerable object (InstructionStatus) to instantiate a new concrete state upon call. This allows the state pattern to be used with an enumerable, such that InstructionStatus.STOP returns the Stop state. The InstructionStatusOperations interface is the “state” interface: it defines actions of each state. It takes advantage of default methods, which were introduced in Java 1.8. This means that each concrete state, such as DownISO, only overrides one method. Each state cannot switch states to itself, and will throw an UnsupportedStatusTransitionException should it happen.

Also pictured in the diagram is the InstructionController. The InstructionController is not part of the state design pattern but is a wrapper around instruction. The InstructionController allows for retrieval of the next instruction state as well as the reversed state property.

Driver

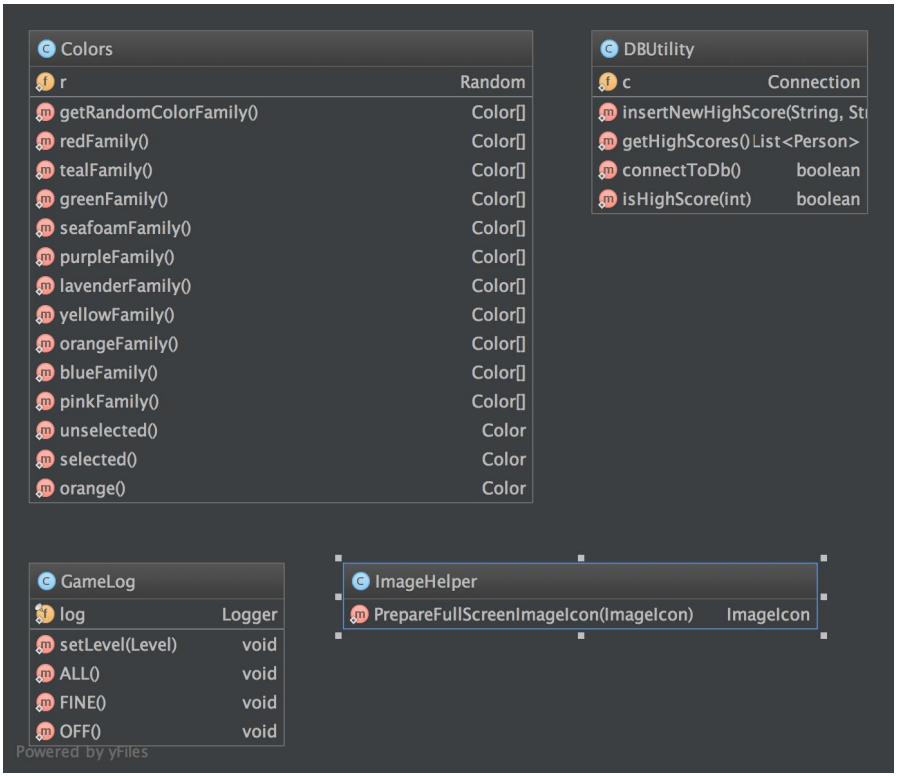
This is the main class of the game. Its only function is to create the JFrame and set it to visible.

Person.java

This class defines the person’s characteristics for easy entry into the database and retrieval from the database. A person has a name, score, and date associated with their high score.

Utilities

This is a collection of unrelated classes that help provide some functionality elsewhere in the program.



Colors.java:
This class provides easy access to all Color objects used throughout the program. The [color]Family methods return a color palette of that color, for use in the game screen. The selected and unselected methods return the color of selection for navigation buttons.

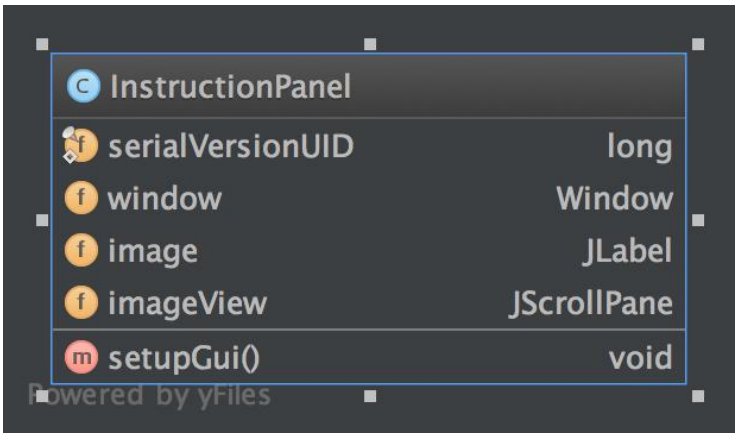
GameLog.java:
The GameLog is an object that is used throughout the program to log state changes in the game. It is a global logger class that is used for debugging. It is a wrapper for the built-in Logger class, and contains some functions to set the level of the logger and quickly shift to commonly used levels.

ImageHelper.java:
This class contains only one method: PrepareFullScreenImageIcon. This method takes an ImageIcon as a parameter and changes its size so that it fits our 1280x720 screen resolution.

DBUtility.java:
This class contains methods for retrieving values from the database and storing values into the database. Also contained within the class are some methods for operationing on data pulled from the database, such as checking if a score is in the top 10 scores currently in the database.

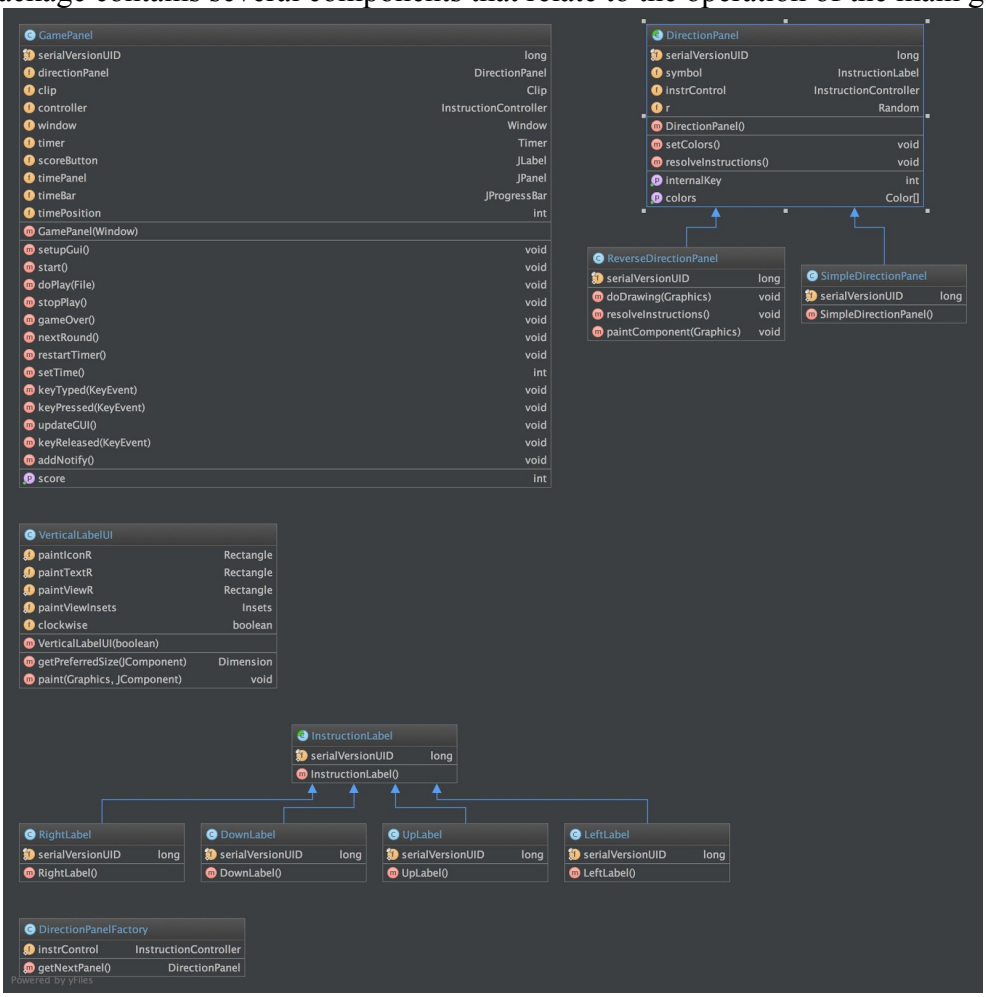
***views**:*
This package contains all graphical elements for the program, such as JLabels and JPanels.

views.directions:



Contains the InstructionPanel. The InstructionPanel simply displays a full-screen image of the instructions for the game. The image of the instructions is stored in the Assets folder.

views.game:
This package contains several components that relate to the operation of the main game.



DirectionPanel.java:
Initially, this class was the only implementation of DirectionPanel, but later became an abstract class when the reverse direction was added. It contains two public methods: resolveInstructions() and setColors(). setColors() randomly sets the colors for the panel based on the colors available in the Colors.java utility class. resolveInstructions() reads the instruction from the InstructionController, and accordingly adds the correct JLabel to the DirectionPanel.

SimpleDirectionPanel.java:
Implements DirectionPanel with no overrides or changes.

ReverseDirectionPanel.java:
Implements DirectionPanel with some changes. Has a doDrawing method that paints the background with horizontal stripes, such that the color of the background alternates between the base color (center value of the color array returned from Colors.java) and the darkest color (final value of the color array). To paint the background as defined by doDrawing, the ReverseDirectionPanel must override paintComponent method. ReverseDirectionPanel also overrides resolveInstructions such that the internal key of each instruction is reversed.

DirectionPanelFactory:
This class returns the appropriate type of direction panel based on the value of the reverse boolean inside of InstructionController.

InstructionLabel:
This abstract class defines all of the shared properties between instruction labels. Shared properties are font size and font type.

LeftLabel:
Extends InstructionLabel. Sets the default text to "<"

RightLabel:
Extends InstructionLabel. Sets the default text to ">"

UpLabel:
Extends InstructionLabel. Sets the default text to "<". Has the default UI set to VerticalLabelUI

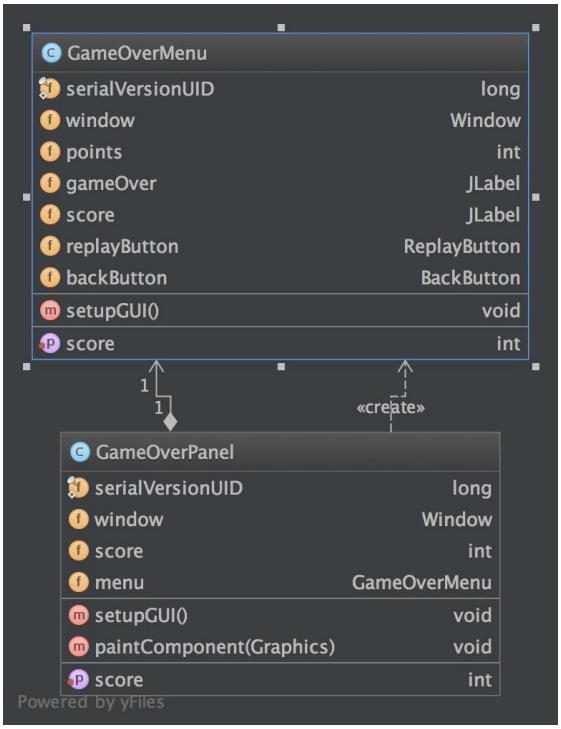
DownLabel:
Extends InstructionLabel. Sets the default text to ">". Has the default UI set to VerticalLabelUI.

VerticalLabelUI:
Allows the label to be rotated 90 degrees.

GamePanel:
This class is the primary controller for the game. It contains many functions and handles the stopping and playing of audio, restarting the timer, going to the next round, starting and ending the game, and listening for key-presses.

views.gameover:

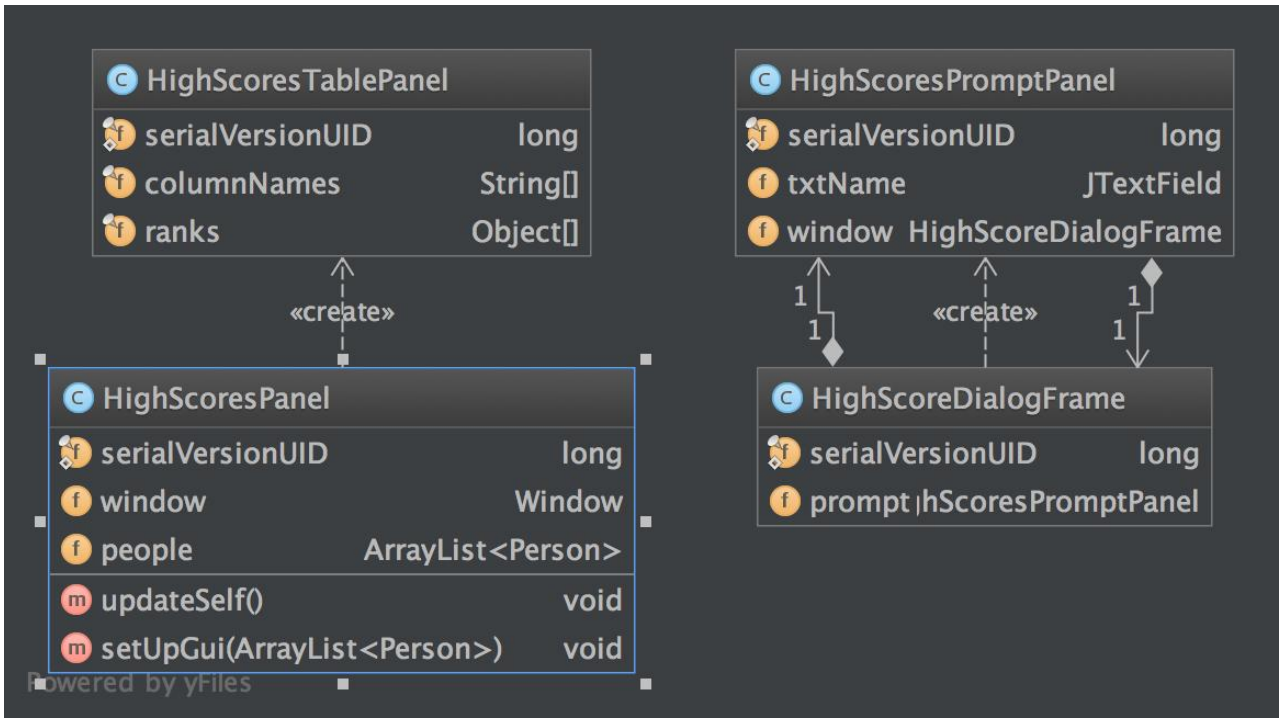
This package contains two classes: GameOverMenu and GameOverPanel. It defines the components that are related to showing the game over screen.



GameOverPanel:
This class houses the menu for the game over screen. It overrides the paintComponent method to display the background from a picture.

GameOverMenu:
This class contains all the informational text to be displayed when the user loses. It has navigation buttons to allow the user to play again or go to the home screen.

views.highscores:
The highscores package contains four classes, all of which are related to displaying the high scores or recording new high scores.



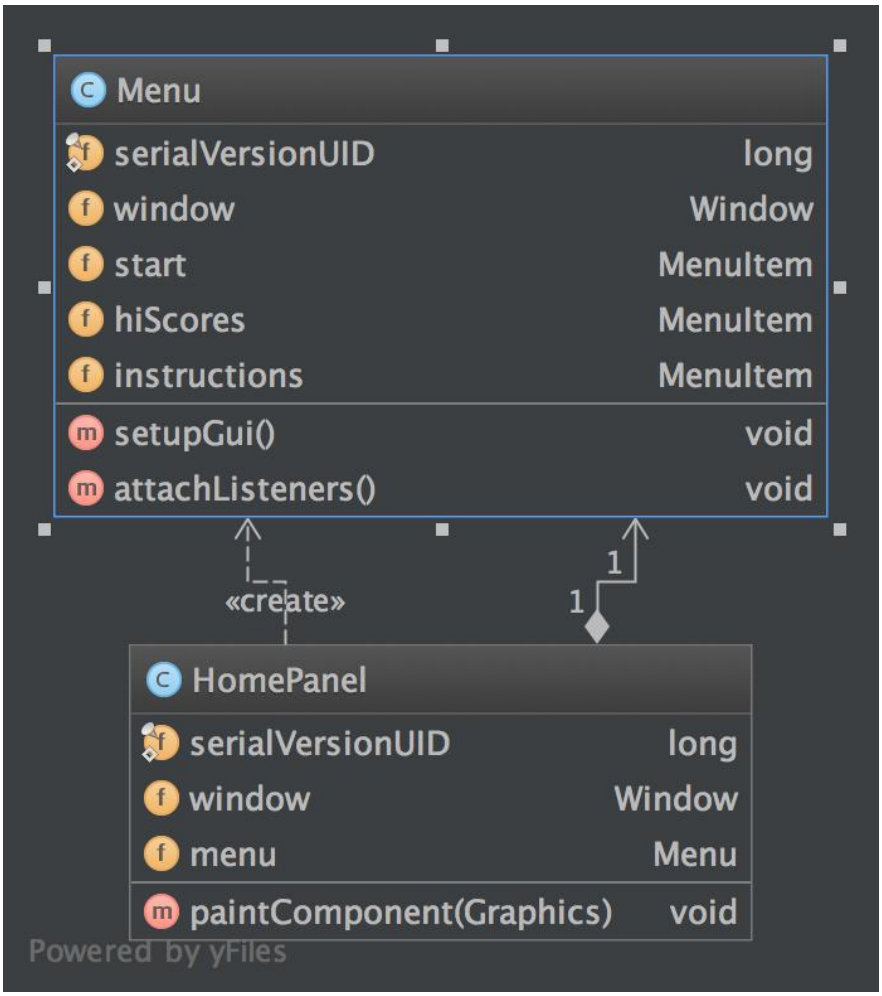
HighScoreTablePanel:
This class sets up the view for the high scores section of the game.

HighScoresPanel:
The discrete view is to be used for the high scores section of the game

HighScoresPromptPanel:
The panel which prompts the user for the username so they can enter the score into the database when they achieve a top ten score.

HighScoreDialogFrame:
The window that pops up upon new high score.

views.home:
This package contains the views for the home screen: the Menu class, and the HomePanel.

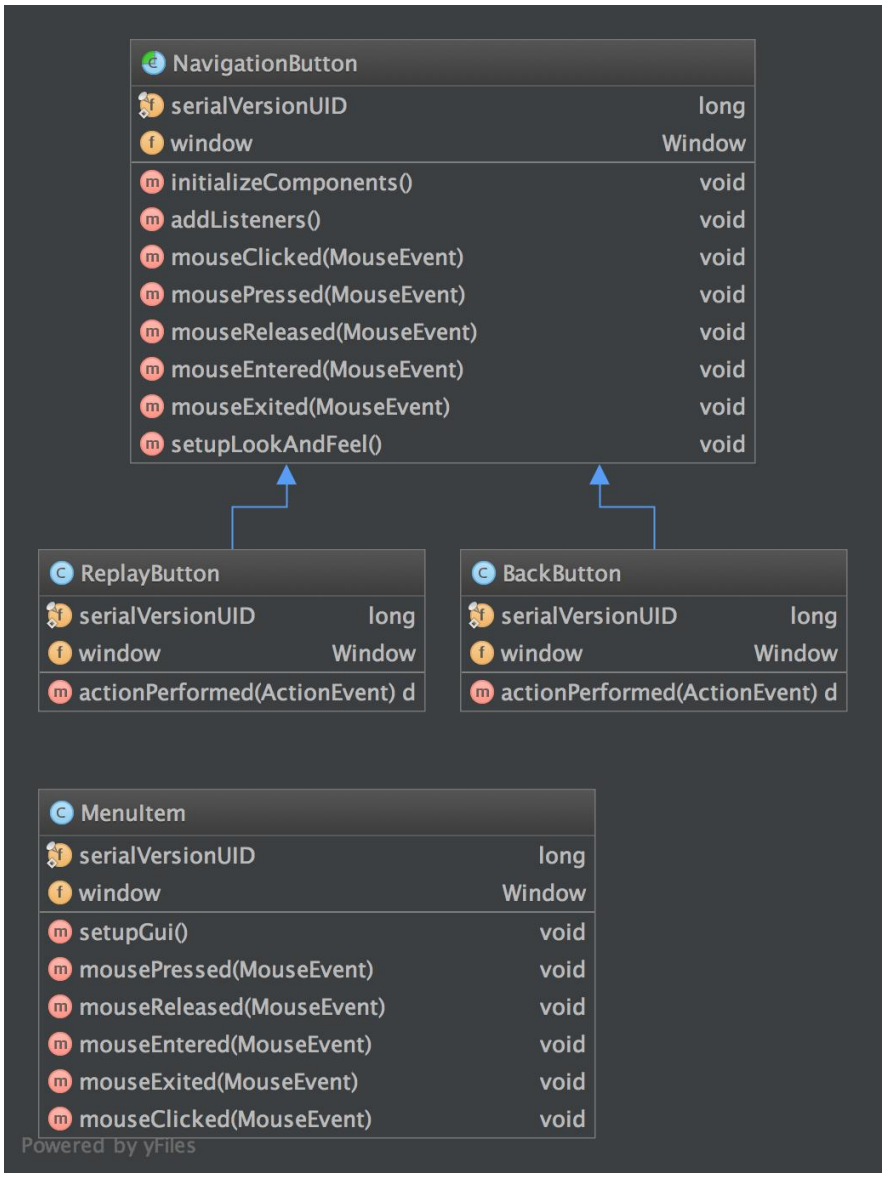


Menu.java:

Contains a number of buttons that allow for navigation to the various parts of the game. The three buttons allow navigation to the game page, to the high scores page, and to the instructions page.

***views.navbuttons*:**

This package contains all the navigation buttons.



NavigationButton:
This is the abstract class that contains all shared features between BackButton and ReplayButton. This class implements MouseListener for button "hover" effects, and sets up the font and style of each button.

ReplayButton:
Extends NavigationButton. Switches back to the game panel on click.

BackButton:
Extends NavigationButton. Switches to the home panel on click.

Database Requirements and Constraints:

hiscores	
name	text
score	int
Date	date

We used a sqlite database to score all the high score data. We used sqlite because sqlite is easily portable with the game and is lightweight and simple to use.. The schema isn't too complex as all we need to keep track of is a person's username, score, and date they achieved that score on. A simple schema diagram can be seen above indicating the field names and data types used for each field in the SQLite database. We use the JDBC API for connecting to the database and inserting and pulling values from the database.

Maintainability:
The code should be very maintainable and easily extendable. The provided documentation should guide new contributors through the software system.

Portability:
This software system should be able to run anywhere that the Java runtime is available, assuming adequate hardware.

Testing
Test-driven development was the method of implementation that we used. In doing so, we briefly tested components after writing them. This allowed for a quick verification of our work and sped up the development process. Tests were written on an as-needed basis: not all classes were extensively tested -- in fact, most weren't. The reason why testing was relatively minimal (especially for a TDD implementation) is because we had extensive logging throughout our classes. Most methods of most classes had entering, exiting, warning, severe, fine, and informational logs to aid in the debugging process.

This project was also developed using Git. When a member of the team attempted to merge a feature branch into the master repository, the pull request was approved by another member of the team. This was to make sure that code added to the project at least had two pairs of eyes examining it and thus minimizing errors. Before approving of pull requests, the code was integration tested on an as-needed basis.

Conclusion
Developing KeySlide helped renew, broaden, and strengthen our knowledge across many disciplines such as Java, Sqlite Databases, design patterns, and several other fields. The end result of our work is what we believe to be a fun game that is easily extensible. We

have open sourced the game and anyone wishing to review or edit the code may do so by downloading the source code from http://www.github.com/hesham8/keyslide.

References

[1]. Palfinger G. Modeling object states and behaviors using a state action manager[J]. Crossroads, 2007, 13(4): 8-8.

[2]. Cooper J W. The design patterns Java companion[J]. 1998.

[3]. Hannemann J, Kiczales G. Design pattern implementation in Java and AspectJ[C]//ACM Sigplan Notices. ACM, 2002, 37(11): 161-173.

[4]. Hinkle, Greg. “Vertical Label UI: Label.” Vertical Label UI: Label. Sapient, Jan. 2002. Web. 27 Apr. 2015.